Ejercicios de programación orientada a objetos con Java y UML

Leonardo Bermón Angarita

UNIVERSIDAD NACIONAL DE COLOMBIA · 1867 ·

Leonardo Bermón Angarita

Profesor asociado del Departamento de Informática y Computación de la Universidad Nacional de Colombia - Sede Manizales. Ingeniero de Sistemas, magíster en Informática de la Universidad Industrial de Santander y doctor en Ingeniería Informática de la Universidad Carlos III de Madrid.

Se desempeñó como director del Departamento de Informática y Computación y coordinador del programa curricular de Administración de Sistemas Informáticos de la Universidad Nacional de Colombia - Sede Manizales del 2011 al 2018. Forma parte del Grupo de Investigación en Aplicaciones y Herramientas Web de la Universidad Nacional de Colombia.

Sus áreas de investigación están orientadas principalmente a la ingeniería de *software*: procesos *software*, modelos de *software* basados en UML y programación orientada a objetos.

Ha trabajo en varios proyectos relacionados con el desarrollo de aplicaciones móviles para el aprendizaje de la lógica computacional, la educación especial, la gestión del conocimiento y las capacidades dinámicas.

Ciencias de Gestión

Ejercicios de programación orientada a objetos con Java y UML

Ejercicios de programación orientada a objetos con Java y UML

Leonardo Bermón Angarita

UNIVERSIDAD
NACIONAL
DE COLOMBIA

Bogotá, D. C., 2021

Primera edición, 2021
ISBN 978-958-794-575-1 (digital)

Colección Ciencias de Gestión

Edición
Editorial Universidad Nacional de Colombia
direditorial@unal.edu.co
www.editorial.unal.edu.co

Dayán Viviana Cuesta Pinzón
Coordinación editorial
Anyeli Rivera Tancón
Corrección de estilo
Henry Ramírez Fajardo
Diseño de la colección
Martha Elena Echeverry Perico
Diagramación

Editado en Bogotá, D. C., Colombia.

Catalogación en la publicación Universidad Nacional de Colombia

Bermón Angarita, Leonardo, 1971-
Ejercicios de programación orientada a objetos con Java y UML / Leonardo Bermón Angarita. -- Primera edición. -- Bogotá: Vicerrectoría de Investigación. Editorial Universidad Nacional de Colombia; Manizales: Universidad Nacional de Colombia. Facultad de Administración. Departamento de Informática y Computación, 2021.
1 CD-ROM (688 páginas): ilustraciones en blanco y negro, diagramas. -- (Colección Ciencias de Gestión).

Incluye referencias bibliográficas e índice temático
ISBN 978-958-794-575-1 (e-book)

1. Programación orientada a objetos (computación) -- Problemas, ejercicios, etc. 2. Java (Lenguaje de programación de computadores) -- Problemas, ejercicios, etc. 3. UML (Computación) -- Problemas, ejercicios, etc. 4. Algoritmos (Computadores) -- Problemas, ejercicios, etc. I. Título II. Serie

CDD-23 005.117 / 2021

▶ Contenido

▶ Lista de figuras

▶ Lista de tablas

Dedico este libro a Lyda,
cuyo amor me guía y motiva todos los días

▶ Introducción

El paradigma orientado a objetos (OO) es un enfoque del desarrollo de *software* que emergió en los años 60 del siglo pasado, se desarrolló en los años 80 y se difundió ampliamente a partir de la década de los 90 (Seidl, Scholz, Huemer y Kappel, 2015). Desde entonces, esta forma de analizar, diseñar y construir sistemas *software* se ha convertido en la forma tradicional de afrontar la complejidad, entender los requisitos de los programas y proponer soluciones a las problemáticas planteadas en el ámbito informático. Sus conceptos fundamentales han sido incorporados en muchos lenguajes de programación existentes.

La evolución es una característica que siempre está presente en el desarrollo de *software*. Por ello, para afrontar con éxito el cambio y mantenimiento constante inherente al *software*, el propósito inicial del paradigma OO era facilitar el reuso y la modificación de los productos de *software*.

Por consiguiente, el paradigma OO trata de obtener un tratamiento balanceado e integrado de los aspectos estáticos (relacionados con la estructura o componentes principales de los programas) y dinámicos (que hacen énfasis en la relación y comunicación entre componentes) de un sistema *software*. El punto de partida del paradigma OO es el concepto de objeto que encapsula dichos aspectos estáticos y dinámicos (Black, 2013). Se ha definido un nuevo conjunto de conceptos en el ámbito del modelado, diseño e implementación de *software* basado en el concepto de objetos.

El paradigma se ha consolidado incluyendo sus conceptos a lo largo del ciclo de vida de desarrollo del *software* (Pressman y Maxim, 2014). Por lo tanto, el concepto de objeto ha sido incorporado en áreas de conocimiento como la ingeniería de *software*; el análisis orientado a objetos (para crear modelos de los requisitos funcionales del programa a desarrollar, logrando comprender los aspectos y conceptos principales que aborda el programa); el diseño orientado a objetos (modelando desde distintos puntos de vista la interacción de

diversos objetos para resolver un problema de *software*); la programación orientada a objetos (desarrollo de código con base en objetos que poseen atributos y métodos, y se comunican entre sí) y las pruebas orientadas a objetos (para probar las clases desarrolladas y las interacciones entre las clases, hasta validar que se cumplan los requisitos de los programas).

Este paradigma es visto generalmente por estudiantes de informática en cursos de programación de primeros años o semestres. Algunas universidades los abordan después de un curso introductorio de algoritmos y de programación estructurada. Otras universidades abordan este paradigma en forma simultánea con la programación estructurada inicial. Indudablemente, estos conceptos de programación son de vital importancia en las etapas iniciales de formación.

Dicho paradigma OO incluye una terminología muy amplia basada en los conceptos fundamentales de objetos, clases, atributos, constructores y métodos. El paradigma tiene una fuerte base epistemológica, ya que está basado en la forma en que los seres humanos estudiamos, analizamos y comprendemos el mundo en que vivimos.

En primer lugar, se identifican objetos concretos (una casa, una mascota, un familiar, etc.); luego, se realizan generalizaciones sobre dichos objetos que serán los tipos o clases (las casas serán instancias de vivienda; las mascotas serán instancias de animal; y los familiares serán instancias de personas, etc.); posteriormente, se identifican atributos para dichas clases (una vivienda tendrá una dirección y teléfono; un animal tendrá un nombre y clasificación; una persona tendrá un nombre, apellidos y un número de documento de identidad) y, finalmente, acciones o métodos que pueden realizar dichas clases (una vivienda se puede comprar o vender; un animal puede alimentarse y emitir sonidos; una persona puede comer, trabajar y dormir). Por ello, el paradigma OO es fácil de asimilar y entender por programadores novatos, ya que utiliza y amplía la forma en que se modelan los "objetos" del mundo estudiado.

Luego de abordar estos conceptos básicos iniciales, los cursos avanzan para estudiar temas más complejos como la herencia, el polimorfismo y la ligadura dinámica. La herencia permitirá que los elementos propios de un objeto (atributos y métodos) puedan ser heredados

por sus hijos (una casa heredará los atributos y métodos de los inmuebles de la ciudad; y los animales y personas heredarán los atributos y métodos de los seres vivos). El polimorfismo otorgará la propiedad que un objeto pueda variar su comportamiento (por ejemplo, una persona podrá desempeñarse como un trabajador con múltiples roles simultáneos, pero sigue siendo la misma persona) y la ligadura dinámica permitirá que el objeto ejecute un único comportamiento en un momento dado (por ejemplo, una persona será un padre, un trabajador asalariado o un comprador en un momento dado).

También se deben estudiar otros tipos de relaciones entre clases como: las asociaciones, las agregaciones y las composiciones. Así como para comprender el mundo real se estudian los objetos descomponiéndolos en sus partes —las viviendas tienen distintos espacios físicos y los seres vivos poseen un cuerpo con diferentes sistemas, aparatos y órganos— de igual manera en un programa OO, los objetos estarán conformados por partes (que a su vez son objetos) relacionadas entre sí.

Finalmente, los cursos pueden incluir otros aspectos de la programación estructurada convencional, pero desde el punto de vista de los objetos como:

- Estructuras de almacenamiento que agrupan objetos de una misma clase (*arrays*, matrices, colecciones, etc.).
- Archivos que permiten almacenar en forma permanente la información contenida en los objetos.
- Manejo de excepciones que permite un tratamiento adecuado de los posibles errores que pueden ocurrir cuando se ejecuta un programa. A su vez, dichas excepciones serán tratadas como objetos.
- Hasta llegar a la construcción de la interfaz gráfica de usuario (GUI) que representa la interacción del usuario con el programa utilizando un modo visual basado en componentes gráficos que serán también objetos.

Los cursos avanzados de programación orientada a objetos incluyen temas más complejos como hilos (diferentes tareas que se ejecutan en forma simultánea en un mismo programa), programación gráfica

(para generar representaciones espaciales en dos o tres dimensiones) y programación en red (para que los programas se comuniquen con otros a través de una red de computadores).

Java —como lenguaje de programación orientado a objetos— es un lenguaje muy robusto, con una colección muy completa de paquetes y clases útiles para desarrollar cualquier tipo de programa. Este lenguaje también es estudiado y utilizado ampliamente a nivel mundial; aunque hay otros lenguajes OO como: Phyton y C++, que tienen las mismas fortalezas en su implementación del paradigma OO.

Desde su lanzamiento en 1995, Java se ha consolidado como uno de los lenguajes de programación más utilizados a nivel mundial. Las aplicaciones que utiliza abarcan diversas áreas y entornos de funcionamiento como dispositivos móviles, sistemas embebidos (combinando *hardware* y *software*), navegadores web, servidores y aplicaciones de escritorio.

Entre las aplicaciones más famosas desarrolladas con Java se encuentran:

- El *software* que controla el Spirit, (vehículo explorador de Marte de la NASA).
- El *software* Integrated Genome Browser para visualizar datos de secuenciación del genoma humano.
- BioJava, una biblioteca de código abierto para la bioinformática, es decir, el procesamiento de datos biológicos.
- El motor de búsqueda de Wikipedia.
- Minecraft, uno de los videojuegos más vendidos y conocidos del mundo.

Este libro de texto contiene material de apoyo complementario para cursos de pregrado de programación orientada a objetos. Sin embargo, no aborda profundamente los conceptos del paradigma orientado a objetos; para ello, el lector debe consultar bibliografía especializada. En su lugar, muestra un breve resumen de los conceptos OO y propone una selección de ejercicios que ayuden al lector a asimilar, aplicar y reforzar dichos conceptos para lograr que los estudiantes asimilen conceptos del paradigma OO utilizando Java como lenguaje de programación.

Este libro es una inducción a los conceptos básicos de diseño para los estudiantes. Para ello, se utiliza el lenguaje unificado de modelado (*UML,* por sus siglas en inglés, *Unified Modeling Language*) como notación visual para representar la estructura estática de los programas planteados (Booch, Rumbaugh y Jacobson, 2017). Cada ejercicio incluye la representación del código Java en un diagrama de clases UML. A su vez, los objetos creados en el método *main* de prueba se representan en un diagrama de objetos UML. Algunos ejercicios pueden incluir diagramas de máquina de estados UML y diagramas de actividad UML. Finalmente, este libro también pretende facilitar la comprensión de pequeños programas orientados a objetos, el funcionamiento de distintas estructuras de control, de almacenamiento y clases propias de la API del lenguaje de programación Java y, así, ofrecer a los lectores los conocimientos necesarios para analizar y resolver problemas aplicando el modelado y la programación orientada a objetos.

En varios libros de texto de programación orientada a objetos con Java se presentan enunciados de programas a ser desarrollados a lo largo de sus capítulos correspondientes. Sin embargo, algunos enunciados son muy abstractos y no están basados en una especificación legible, práctica y fácil de entender. Este libro trata de afrontar estos inconvenientes presentando una especificación clara de los problemas a ser resueltos y mostrando una solución que pondrá en práctica los conceptos OO presentados.

A su vez, la mayoría de los libros de texto están orientados a presentar el código y la solución correspondiente de un programa, sin afrontar el análisis y diseño de la solución. Este libro presenta modelos basados en UML que ayudan a tener una visión general desde el punto de vista de la estructura estática de las soluciones.

Cada ejercicio propuesto está compuesto de los siguientes elementos:

- Conceptos teóricos: hay una breve descripción de los conceptos teóricos OO a aplicar durante la realización del ejercicio de programación.
- Objetivos de aprendizaje: enuncia los objetivos que el lector logrará con la realización del ejercicio de programación propuesto.

- Enunciado del ejercicio: descripción del ejercicio de programación a realizar. Puede estar conformado por una o varias clases. Para cada clase se identifican sus requerimientos a nivel de atributos y métodos.
- Instrucciones Java utilizadas en el ejercicio: una tabla que presenta las principales instrucciones, clases y métodos utilizados en la resolución del ejercicio.
- Solución propuesta: se presenta la solución realizada en Java de cada enunciado propuesto. El código se encuentra documentado utilizando la notación *Javadoc*, el cual es una herramienta de Oracle que se ha convertido en un estándar de la industria para documentar clases de Java.
- Diagrama de clases: hay un diagrama de clases UML para la solución propuesta.
- Explicación del diagrama de clases: se presenta una descripción más detallada de los elementos UML utilizados en el diagrama de clases de la solución del ejercicio.
- Diagrama de objetos: hay un diagrama de objetos UML para la solución propuesta. Los objetos instanciados en el método *main* de la solución son representados utilizando este diagrama.
- Ejercicios propuestos: se presentan los enunciados de varios ejercicios para que el lector refuerce los conceptos vistos o amplíe la funcionalidad del programa propuesto.

Algunos ejercicios incluirán diagramas UML adicionales si así lo requieren, como diagramas de actividad o diagramas de máquinas de estado con su correspondiente explicación.

Este libro está organizado en nueve partes que ponen en práctica numerosos conceptos básicos y avanzados de la programación orientada a objetos utilizando Java.

En el primer capítulo del libro se presenta una serie de ejercicios que describen y aplican las estructuras básicas de programación de Java. En concreto, se presentan ejercicios que utilizan estructuras condicionales (*if-else*), estructuras repetitivas (*while*, *do-while* y *for*) y *arrays*. Estos conceptos, aunque no son propiamente orientados a objetos, forman las estructuras básicas que controlan el flujo de ejecución de los programas y serán utilizadas ampliamente en los diferentes ejercicios.

En el segundo capítulo hay una selección de ejercicios básicos de programación OO relacionados con la definición de clases e instanciación de objetos. Se revisan y aplican conceptos relacionados con atributos, constructores y métodos. Además, se estudian los conceptos de sobrecarga de métodos y constructores, diferentes tipos de métodos y la visibilidad de los elementos.

En el tercer capítulo se presenta un conjunto de ejercicios de programación para entender y aplicar clases específicas como *String*, *System* y los diferentes *wrappers* de Java. También, se estudian los elementos estáticos y diversas estructuras de almacenamiento como *arrays* y vectores.

El conjunto de ejercicios del cuarto capítulo aborda los conceptos relacionados con la herencia y el polimorfismo. Para ello, los ejercicios propuestos permitirán definir jerarquías de herencia, invocar constructores y métodos heredados y polimórficos. Asímismo, se estudian las clases y métodos abstractos, y se presentan sus similitudes y diferencias con las interfaces.

En el quinto capítulo se presentan ejercicios para aplicar los conceptos de relaciones entre clases, fundamentalmente las relaciones de asociación y luego las relaciones especializadas de agregación y composición.

En el sexto capítulo, una colección de ejercicios permitirá entender los conceptos de genericidad por medio del desarrollo de clases y métodos genéricos. De igual manera, se estudian las excepciones como mecanismo para el tratamiento y gestión de errores. Por último, en esta parte se aborda la lectura y escritura de archivos para el almacenamiento persistente de información de los objetos.

En el séptimo capítulo se presentan algunas clases útiles que pueden utilizarse para mejorar el desarrollo de programas que incluyan manejo de fechas y horas (*LocalDate*), división de *Strings* (*StringTokenizer*), formatos numéricos (*NumberFormat*) y generación de números aleatorios (*Math.random*).

En el octavo capítulo se encuentran los ejercicios relacionados con el aprendizaje de conceptos sobre la construcción de interfaces gráficas de usuario. Para ello, se aplican diversos contenedores y componentes *swing* que incluye Java. Además de generar la interfaz gráfica utilizando

el paquete *swing*, se estudian y aplican los conceptos para la gestión de los eventos que ocurren cuando el usuario interacciona con la interfaz.

En el noveno capítulo hay ejercicios relacionados con el desarrollo de programas basados en *JavaFX*. El capítulo incluye la generación de interfaces gráficas de usuario utilizando este paquete, la construcción de diversos componentes gráficos como: figuras 2D, 3D, aplicación de transformaciones y animaciones a dichas figuras y generación de gráficas.

Por último, se presentan una serie de anexos que permiten al lector consultar conceptos teóricos relacionados con la sintaxis del lenguaje de programación Java, los diagrama UML utilizados en este libro y herramientas de desarrollo para programar en Java.

Los diagramas UML presentados en este libro fueron realizados con la herramienta StarUML® (*http://www.staruml.io*), una herramienta extensamente utilizada tanto en entornos académicos como industriales y que es compatible con el metamodelo estándar UML 2.X soportando la construcción de numerosos diagramas UML.

Capítulo 1
Estructuras básicas de programación

El propósito general de este capítulo es conocer las estructuras básicas de programación que tiene el lenguaje de programación Java para especificar el flujo de control de los programas. Estas estructuras básicas forman el núcleo del lenguaje y serán ampliamente utilizadas en el resto de los ejercicios del libro.

En este primer capítulo se presentan cinco ejercicios sobre estructuras básicas de programación utilizando Java.

- **Ejercicio 1.1. Estructura condicional *if-else***
  La estructura condicional *if-else* permite ejecutar diferentes porciones de código basadas en una condición booleana. El primer ejercicio trata sobre la utilización de *if* anidados.

- **Ejercicio 1.2. Estructura repetitiva *while***
  En el segundo ejercicio se abordan las estructuras repetitivas, iniciando con la estructura *while* que permite repetir una instrucción o bloque de instrucciones siempre que una condición particular sea verdadera. El segundo ejercicio coloca en práctica la aplicación de la estructura *while*.

- **Ejercicio 1.3. Estructura repetitiva *do-while***
  Otra estructura repetitiva es el *do-while*, que repite una instrucción o bloque de instrucciones siempre y cuando se cumpla cierta condición booleana. Se diferencia del

while en que la instrucción o bloque se ejecuta al menos una vez. El tercer ejercicio aborda la utilización de la estructura *do-while*.

- **Ejercicio 1.4. Estructura repetitiva *for***
  En la misma línea, otra estructura repetitiva muy utilizada es el *for*, el cual repite una instrucción o un bloque de instrucciones varias veces hasta que una condición se cumpla. El cuarto ejercicio propone un algoritmo que utiliza dicha estructura.

- **Ejercicio 1.5. *Arrays***
  Por último, el concepto de *array* es extensamente aplicado en numerosas soluciones de algoritmos para agrupar una colección de elementos de un mismo tipo. Por ello, se presenta un ejercicio que aplica este concepto.

Los diagramas UML utilizados en este capítulo son los diagramas de actividad. Las soluciones presentadas están más orientadas a la programación estructurada que a la programación orientada a objetos. Por ello, se utilizan los diagramas de actividad UML, para modelar el flujo de control de los algoritmos y podrían reemplazar a los típicos diagramas de flujo.

Ejercicio 1.1. Estructura condicional *if-else*

La declaración *if* es la más básica de todas las declaraciones de flujo de control en Java. Esta sentencia le indica al programa que ejecute una determinada sección de código solo si una condición en particular se evalúa como verdadera (Vozmediano, 2017). El formato de la instrucción es el siguiente:

```
if (condición) {
        bloque de instrucciones
}
```

Si la condición se evalúa como falsa, el control salta al final de la declaración *if*.

Otro formato de esta instrucción es utilizando *if-else*, el cual proporciona una ruta secundaria cuando la condición se evalúa como falsa. El formato es el siguiente:

```
if (condición) {
```

```
        bloque de instrucciones
} else {
        bloque de instrucciones
}
```

Además, la estructura *if-else* se puede anidar, es decir, que se puede usar una declaración *if* o *else-if* dentro de otra declaración *if* o *else-if*. El formato es el siguiente:

```
if (condición) {
        bloque de instrucciones
} else if (condición) {
        bloque de instrucciones
} else {
        bloque de instrucciones
}
```

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Entender el concepto de flujo de control condicional.
- Definir estructuras condicionales utilizando Java.
- Definir estructuras condicionales anidadas.

Enunciado: índice de masa corporal

Se desea desarrollar un programa que calcule el índice de masa corporal de una persona. Para ello, se requiere definir el peso de la persona (en kilogramos) y su estatura (en metros). El índice de masa corporal (IMC) se calcula utilizando la siguiente fórmula:

$$IMC = \frac{peso}{estatura^2}$$

Luego, a partir del IMC obtenido se pueden calcular si una persona tiene un peso normal, inferior o superior al normal u obesidad. Para generar estos resultados el IMC calculado debe estar en los rangos de la tabla 1.1.

Tabla 1.1. Cálculo de IMC

IMC	Resultado	IMC	Resultado
< 16	Delgadez severa	[25-30)	Sobrepeso
[16-17)	Delgadez moderada	[30-35)	Obesidad leve
[17-18.5)	Delgadez leve	[35-40)	Obesidad moderada
[18.5-25)	Peso normal	>=40	Obesidad mórbida

Instrucciones Java del ejercicio

Tabla 1.2. Instrucciones Java del ejercicio 1.1.

Instrucción	Descripción	Formato
main	Método que define el punto de inicio de un programa. Es un método que no retorna ningún valor.	*public static void main(String args[])*
System.out.println	Método que imprime en pantalla el argumento que se le pasa como parámetro y luego realiza un salto de línea.	*System.out.println*(argumento);
Math.pow	Método que devuelve el valor del primer argumento elevado a la potencia del segundo argumento.	*Math.pow*(argumento1, argumento2);

Solución

Clase: IndiceIMC

```
/**
* Esta clase denominada IndiceIMC calcula el índice de masa corporal de
* una persona y con base en el resultado indica si tiene un peso normal,
* inferior o superior al normal u obesidad.
* @version 1.0/2020
*/
public class IndiceIMC {
```

```
    /**
     * Método main
     */
    public static void main(String args[]) {
        int masa = 91; // Masa en kilogramos
        double estatura = 1.77; // Estatura en metros
        double IMC = masa/Math.pow(estatura, 2); /* Calcular el índice
            de masa corporal */
        System.out.println("La persona tiene una masa = " + masa + "
            kilogramos y estatura = " + estatura + " metros"); /* Mediante
            varios if-else anidados se evalúan diferentes rangos del IMC */
        if (IMC < 16) {
            System.out.println("La persona tiene delgadez severa.");
        } else if (IMC < 17) {
            System.out.println("La persona tiene delgadez moderada.");
        } else if (IMC < 18.5) {
            System.out.println("La persona tiene delgadez leve.");
        } else if (IMC < 25) {
            System.out.println("La persona tiene peso normal.");
        } else if (IMC < 30) {
            System.out.println("La persona tiene sobrepeso.");
        } else if (IMC < 35) {
            System.out.println("La persona tiene obesidad leve.");
        } else if (IMC < 40) {
            System.out.println("La persona tiene obesidad media.");
        } else {
            System.out.println("La persona tiene obesidad mórbida.");
        }
    }
}
```

Diagrama de actividad

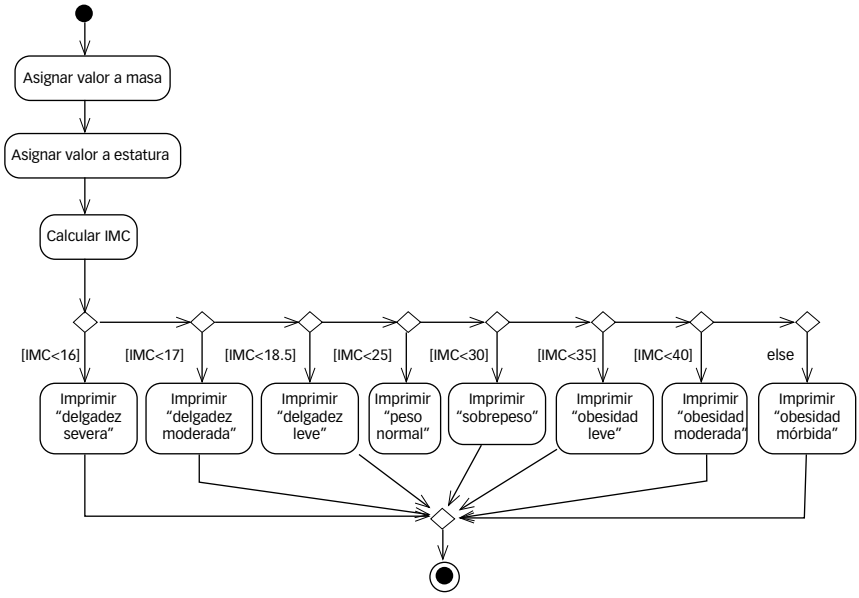


Figura 1.1. Diagrama de actividad del ejercicio 1.1.

Explicación del diagrama de actividad

El diagrama de actividad UML muestra el algoritmo del programa desarrollado para calcular el índice de masa corporal y mostrar un mensaje al usuario de acuerdo con el IMC obtenido.

El diagrama posee una actividad inicial representada por medio de un círculo negro al inicio del programa.

Las instrucciones del programa son "actividades" que se representan en UML con un rectángulo con los bordes redondeados. Mediante flechas se establece el flujo de control del programa y por medio de rombos se definen comportamientos condicionales que, si se cumplen, se bifurcarán por la ruta indicada en el flujo de control del programa.

El programa inicia con la asignación de valores a las variables masa y estatura. Luego, se calcula el IMC. Después, de acuerdo con el IMC calculado, se establece una serie de *ifs* (representados por medio de rombos que muestran bifurcaciones con expresiones booleanas encerradas entre []), si una de dichas condiciones se cumple, se imprime un texto indicando el estado de la persona.

Por lo tanto, el programa está conformado por una serie de *ifs* anidados, donde si un *if* no se cumple, se pasa a evaluar el siguiente y así sucesivamente, hasta finalizar el programa, el cual se representa como una actividad final con círculos concéntricos negro y blanco.

Ejecución del programa

```
La persona tiene una masa = 91 kilogramos y estatura = 1.77 metros
La persona tiene sobrepeso.
```

Figura 1.2. Ejecución del programa del ejercicio 1.1.

Ejercicios propuestos

- Hacer un programa que calcule las raíces de una ecuación cuadrática.
- Hacer un programa que, dado el número de un mes, presente el nombre del mes y determine la cantidad de días que tiene.
- Hacer un programa que determine si un año es bisiesto o no.

Ejercicio 1.2. Estructura repetitiva *while*

La sentencia *while* ejecuta continuamente un bloque de instrucciones mientras una condición es verdadera (Altadill-Izurra y Pérez-Martínez, 2017). El formato de la expresión es:

```
while (condición) {
        bloque de instrucciones
}
```

La instrucción *while* evalúa la condición que debe devolver un valor booleano. Si la expresión se evalúa como verdadera, la instrucción *while* ejecuta el bloque de instrucciones. La instrucción *while* continúa evaluando la expresión y ejecutando el bloque de instrucciones hasta que la condición se evalúe como falsa.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Comprender el significado y aplicación de la sentencia *while*.
- Desarrollar algoritmos que implementen dicha sentencia.

Enunciado: número de Amstrong

Se quiere desarrollar un programa que determine si un número es un número de Amstrong. Un número de Amstrong es aquel que es igual a la suma de sus dígitos elevados a la potencia de su número de cifras.

Por ejemplo, el número 371 es un número que cumple dicha característica ya que tiene tres cifras y:

$$371 = 3^3 + 7^3 + 1^3 = 27 + 343 + 1 = 371$$

Instrucciones Java del ejercicio

Tabla 1.3. Instrucciones Java del ejercicio 1.2.

Instrucción	Descripción	Formato
Math.floor	Devuelve el máximo entero menor o igual a un número pasado como parámetro.	*Math.pow*(x);
Math.log10	Devuelve el logaritmo en base 10 de un número pasado como parámetro.	*Math.log10*(x);
%	Operador resto de la división entera.	número % número
Math.pow	Método que devuelve el valor del primer argumento elevado a la potencia del segundo argumento.	*Math.pow*(argumento1, argumento2);

Solución

Clase: NúmeroAmstrong

```
/**
* Esta clase denominada NúmeroAmstrong determina si un número
* entero cumple el requerimiento que la suma de sus dígitos elevado a su
* cantidad de dígitos es el mismo número.
* @version 1.0/2020
*/
```

```
public class NúmeroAmstrong {

    /**
    * Método main
    */
    public static void main(String args[]) {
        int númeroOriginal, últimoDigito; /* Variables para el número
            original y su último dígito */
        double dígitos; // Cantidad de dígitos que tiene el número
        double suma = 0; /* Variable que sumará los dígitos elevados a su
            cantidad de dígitos */
        int número = 371; /* Número a determinar si es un número de
            Amstrong */

        númeroOriginal = número; /* Copia el valor del número para su
            procesamiento */
        dígitos = Math.floor(Math.log10(número)) + 1; /* Calcula el total
            de dígitos del número */
        // Calcula la suma de potencia de digitos
        while (número > 0) {
            últimoDigito = número % 10; // Extrae el último dígito
            // Calcula la suma de potencias del último dígito
            suma = suma + Math.pow(últimoDigito, dígitos);
            número = número / 10; // Elimina el último dígito
        }

        */ Verifica si es un número de Armstrong si la suma obtenida es
            igual al número */
        if (númeroOriginal == suma) {
            System.out.println(númeroOriginal + " es un número de
                Amstrong");
        } else {
            System.out.println(númeroOriginal + " no es un número de
                Amstrong");
        }
    }
}
```

Diagrama de actividad

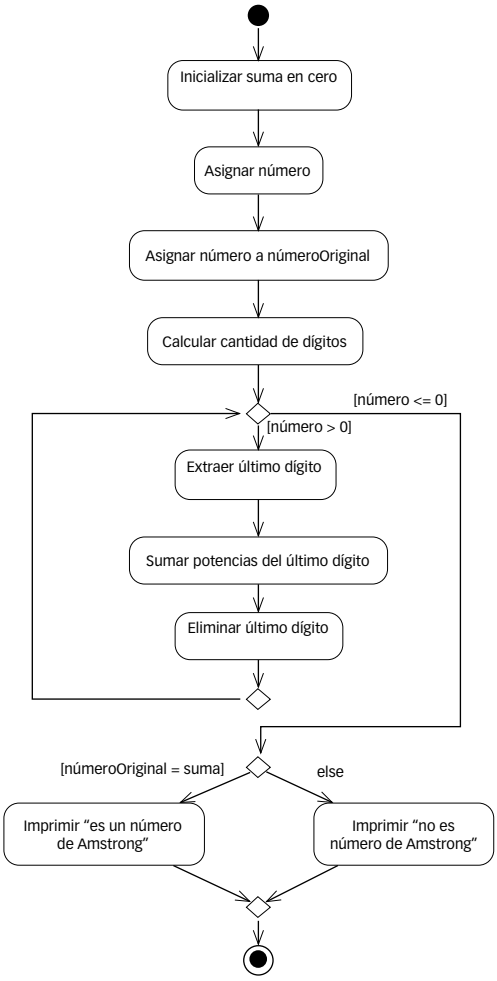


Figura 1.3. Diagrama de actividad del ejercicio 1.2.

Explicación del diagrama de actividad

La figura 1.3 muestra, como un diagrama de actividad UML, el algoritmo de la solución del ejercicio para determinar si un número dado es un número de Amstrong.

El diagrama posee una actividad inicial representada por medio de un círculo negro al inicio del programa.

Las diferentes instrucciones del programa se modelan como "actividades", las cuales se representan en UML como rectángulos con los bordes redondeados. El programa identifica como actividades las instrucciones de inicializar variables, asignar valores a las variables y calcular la cantidad de dígitos.

El ciclo *while* está representado por medio de una pareja de rombos que marcan el inicio y fin del ciclo. Por lo tanto, mientras el número sea mayor que cero, se ejecutarán en forma iterativa las actividades: extraer último dígito, sumar potencias del último dígito y eliminar último dígito. Cuando el número sea menor que cero, finaliza el ciclo y se evalúa en un *if* si la suma obtenida es el número original. Si se cumple está condición, se imprime que es un número de Amstrong. En caso contrario, se imprime el mensaje correspondiente. Con estas impresiones finaliza el programa, el cual se representa por medio de un círculo blanco y negro concéntrico.

Ejecución del programa

```
371 es un número de Amstrong
```

Figura 1.4. Ejecución del programa del ejercicio 1.2.

Ejercicios propuestos

- Escribir un programa que dado un número entero positivo *n*, calcule la suma de la siguiente serie: 1 + 1/2 + 1/3 + 1/4 + 1/5 + … + 1/n
- Escribir un programa que calcule los primeros *n* números de Fibonacci. Los números de Fibonacci comienzan con 0 y 1, y cada término siguiente es la suma de los anteriores: 0, 1, 2, 3, 5, 8, 13, 21, …

Ejercicio 1.3. Estructura repetitiva *do-while*

La sentencia *do-while* también es una estructura repetitiva que se ejecuta siempre y cuando se cumpla cierta condición booleana. La diferencia entre *do-while* y *while* es que la estructura *do-while* evalúa su expresión en la parte inferior del bucle en lugar de la parte superior (Bloch, 2017). Por lo tanto, si se utiliza la estructura *do-while*, el bloque de instrucciones se ejecutará por lo menos una vez. El formato de la instrucción es la siguiente:

do {
bloque de instrucciones
} while (condición)

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Entender el concepto de la sentencia *do-while*.
- Definir estructuras repetitivas utilizando la sentencia *do-while*.
- Diferenciar la estructura repetitiva *while* y *do-while*.

Enunciado: número perfecto

Se quiere desarrollar un programa que determine si un número es un número perfecto. Un número perfecto es aquel que es igual a la suma de sus divisores positivos.

Por ejemplo, el número 28 es un número perfecto ya que sus divisores son: 1, 2, 4, 7 y 14, y la suma de estos números es 28.

Instrucciones Java del ejercicio

Tabla 1.4. Instrucciones Java del ejercicio 1.3.

Instrucción	Descripción	Formato
%	Operador resto de la división entera.	número % número
++	Operador de incremento, utilizado para incrementar un valor en 1. Tiene dos variedades: preincremento (el valor se incrementa primero y luego se calcula el resultado) y posincremento (el valor se usa por primera vez para calcular el resultado y luego se incrementa).	Preincremento: variable++; Posincremento: ++variable;

Solución

Clase: NúmeroPerfecto

```
/**
 * Esta clase denominada NúmeroPerfecto determina si la suma de los
 * divisores de un número es el mismo número.
 * @version 1.0/2020
 */
```

```
public class NúmeroPerfecto {

    /**
    * Método main
    */
    public static void main(String args[]) {
        int suma = 0; // Variable que sumará los divisores del número
        int número = 496; // Número a determinar si es perfecto o no
        int i = 1; /* Variable utilizada para determinar los divisores del
            número */
        // Calcula la suma de todos los divisores
        do {
            // Si i es un divisor del número, se va acumulando
            if (número % i == 0) {
                suma = suma + i;
            }
            i++;
        } while (i <= número / 2); /* No existen divisores mayores a la
            mitad del número */
        // Verifica si la suma de los divisores del número es igual al número
        if (suma == número) {
            System.out.println(número + " es un número perfecto");
        } else {
            System.out.println(número + " no es un número perfecto");
        }
    }
}
```

Diagrama de actividad

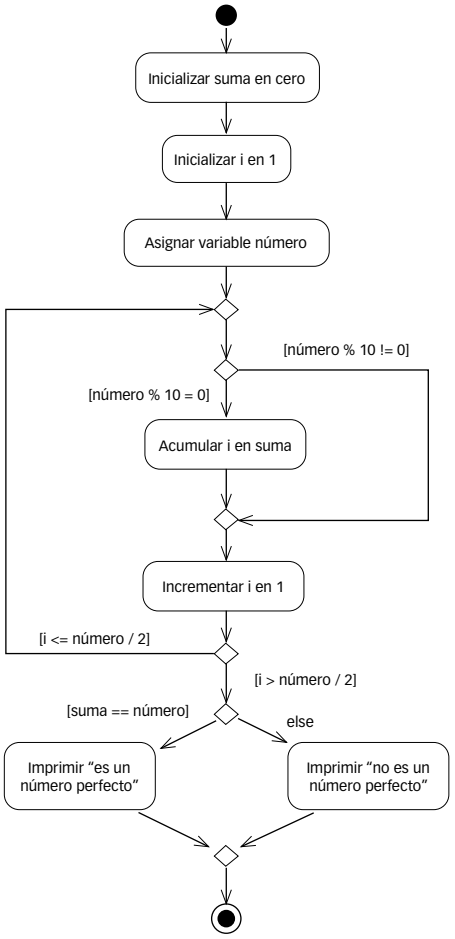


Figura 1.5. Diagrama de actividad del ejercicio 1.3.

Explicación del diagrama de actividad

El diagrama de actividad UML presentado muestra un modelo del flujo de control del algoritmo para determinar si un número es perfecto o no.

El diagrama posee una actividad inicial representada por medio de un círculo negro al inicio del programa.

Las diferentes instrucciones del programa se modelan como "actividades", las cuales se representan en UML como rectángulos con los bordes

redondeados. El programa identifica como actividades las instrucciones de inicializar variables y asignar valores a las variables.

El ciclo *do-while* está representado por medio de una pareja de rombos que marcan el inicio y fin del ciclo. Por lo tanto, mientras la condición (variable i < número /2) se cumpla, se ejecutarán en forma iterativa la actividad condicional de acumular i en la variable suma, la cual se ejecuta si el residuo entre el número y 10 es cero. Se puede observar que este ciclo se ejecutará por lo menos una vez en el programa.

Se evalúa si la suma obtenida es igual al número al terminar el ciclo *do-while*, si se cumple esta condición se imprime o no el mensaje indicando si el número es perfecto o no. Con estas impresiones finaliza el programa, el cual se representa por medio de un círculo blanco y negro concéntrico.

Ejecución del programa

```
496 es un número perfecto
```

Figura 1.6. Ejecución del programa del ejercicio 1.3.

Ejercicios propuestos

- Escribir un programa que, dado un número, determine cuántos dígitos tiene.
- Escribir un programa que, dadas 5 notas finales, determine cuántas notas fueron mayores o iguales a 3.0.

Ejercicio 1.4. Estructura repetitiva *for*

La instrucción *for* proporciona una forma compacta de iterar sobre un rango de valores. Este tipo de ciclo se repite hasta que se cumple una condición particular (Cadenhead, 2017). La forma general de la declaración *for* es:

```
for (inicialización; terminación; incremento) {
        bloque de instrucciones
}
```

La expresión de inicialización da inicio al ciclo y se ejecuta una vez, cuando comienza el ciclo. El ciclo termina cuando la expresión de terminación se

evalúa como falsa. La expresión de incremento se invoca después de cada iteración a través del ciclo e incrementa o disminuye un determinado valor.

El ciclo *for* se utiliza con bastante frecuencia en iteraciones simples en las que se repite un bloque de instrucciones un cierto número de veces y luego se detiene, pero se pueden utilizar ciclos *for* para casi cualquier tipo de ciclo.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Comprender el significado y aplicación de la sentencia *for*.
- Desarrollar algoritmos que implementen dicha sentencia.

Enunciado: números amigos

Se quiere desarrollar un programa que determine si dos números son amigos. Dos números enteros positivos se consideran amigos si la suma de los divisores de uno es igual al otro número y viceversa.

Por ejemplo, los números 220 y 284 son amigos. Los divisores del número 220 son: 1, 2, 4, 5, 10, 11, 20, 22, 44, 55 y 110, y suman 284. Los divisores de 284 son: 1, 2, 4, 71 y 142, que suman 220.

Instrucciones Java del ejercicio

Tabla 1.5. Instrucciones Java del ejercicio 1.4.

Instrucción	Descripción	Formato
%	Operador resto de la división entera.	número % número

Solución

Clase: NúmerosAmigos

```
/**
 * Esta clase denominada Números amigos determina si una pareja de
 * números son amigos. Dos números enteros positivos son amigos si la
 * suma de los divisores propios de un número es igual al otro número
 * y viceversa.
 * @version 1.0/2020
 */
```

```
public class NúmerosAmigos {

    /**
    * Método main
    */
    public static void main(String[] args) {
        int suma = 0; // Variable que sumará los divisiores de un número
        int número1 = 220; // Definición del primer número
        int número2 = 284; // Definición del segundo número
        // Suma todos los divisores del número 1
        for(int i = 1; i < número1; i++) {
            if (número1 % i == 0) {
                suma = suma + i;
            }
        }
        // Si la suma de los divisores del número 1 es igual al número 2
        if (suma == número2) {
            suma = 0;
            // Suma los divisores del número 2
            for(int i = 1; i < número2; i++) {
                if (número2 % i == 0) {
                suma= suma + i;
            }
        }
        // Si la suma de los divisores de ambos números son iguales
        if (suma == número1) {
            System.out.println(número1 + " y " + número2 + " son ami-
                gos");
        } else {
            System.out.println(número1 + " y " + número2 + " no son
                amigos");
            }
        } else {
            System.out.println(número1 + " y " + número2 + " no son
                amigos");
        }
    }
}
```

Diagrama de actividad

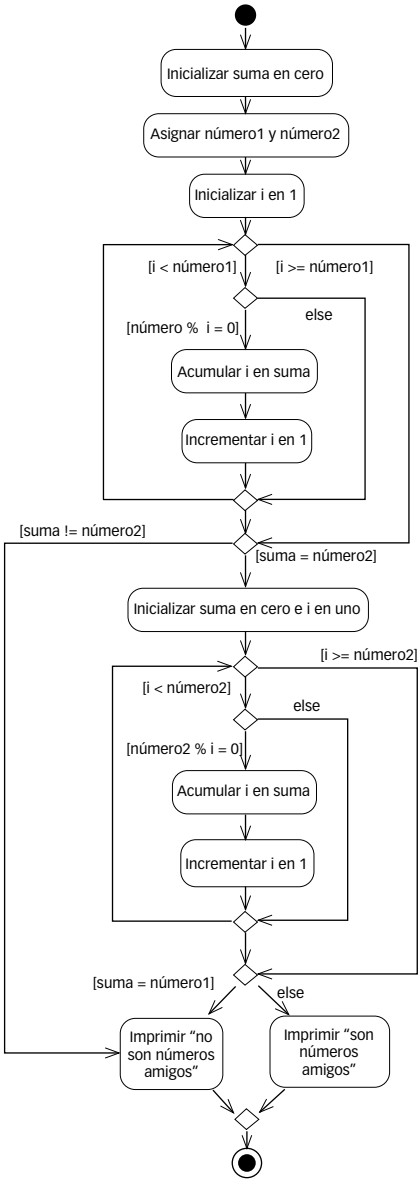


Figura 1.7. Diagrama de actividad del ejercicio 1.4.

Explicación del diagrama de actividad

El diagrama de actividad UML presentado muestra un modelo del flujo de control del algoritmo para determinar si un par de números se consideran amigos o no.

El diagrama posee una actividad inicial representada por medio de un círculo negro al inicio del programa.

Las diferentes instrucciones del programa se modelan como "actividades", las cuales se representan en UML como rectángulos con los bordes redondeados. El programa identifica como actividades las instrucciones de inicializar la variable suma, asignar valores a las variables número1 y número2 e inicializar i en 1.

Los ciclos *for* se representan por medio de una pareja de rombos que marcan el inicio y fin del ciclo. Por lo tanto, la actividad condicional (si el residuo entre número1 e i es cero) se ejecutará en un ciclo desde i = 1 hasta que sea igual a número1. Si la condición se cumple, se acumula el valor de i en la variable suma.

Al final este ciclo *for*, se evalúa si la suma obtenida es igual al número 2. Si dicha condición se cumple, se ingresa a un segundo ciclo, que se repite desde i igual a uno hasta que i sea igual a número2. Durante dicho ciclo se acumula el valor de i en suma. Al finalizar el ciclo, se evalúa si la suma obtenida es igual al número2. De acuerdo con esta condición, se imprime o no el mensaje indicando si los números son amigos o no. Con estas impresiones finaliza el programa, el cual se representa por medio de un círculo blanco y negro concéntrico.

Ejecución del programa

```
220 y 284 son amigos
```

Figura 1.8. Ejecución del programa del ejercicio 1.4.

Ejercicios propuestos

- Desarrollar un programa que calcule el factorial de un número entero positivo. El factorial de un número es el producto de todos los números enteros positivos desde 1 hasta el número en cuestión.

- Desarrollar un programa de determine el máximo común divisor y el mínimo común múltiplo de un número.

Ejercicio 1.5. *Arrays*

Un *array* es un objeto que contiene un número fijo de valores de un solo tipo. La longitud de un *array* se establece cuando se crea el *array*. Después de su creación, su longitud es fija y no puede cambiar (Clark, 2017).

Cada elemento de un *array* se accede por medio de su índice numérico. Los índices numéricos comienzan con 0. Por ejemplo, el séptimo elemento se accedería a través del índice 6 como se observa en la figura 1.9.

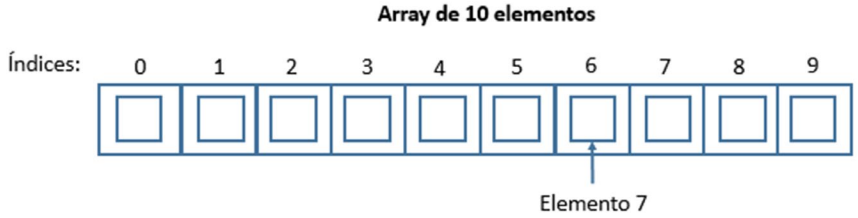


Figura 1.9. Estructura de un *array* con sus índices y elementos

Una *array* se declara de la siguiente forma:

tipoDato [] nombreArray;

El *array* se crea utilizando el operador *new:*

nombreArray = new int[tamaño];

donde tamaño es un valor entero que representa el tamaño máximo del *array*.

Los siguientes formatos de instrucciones asignan valores a cada elemento del *array*:

```
nombreArray[0] = valor; // Inicializa el primer elemento
nombreArray[1] = valor; // Inicializa el segundo elemento
nombreArray[2] = valor; // Inicializa el tercer elemento
```

Una forma alternativa y abreviada de inicializar un *array* es proporcionar sus elementos entre llaves y separados por comas. Por ejemplo, un *array* de valores enteros:

```
int[] unArray = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
```

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Entender y aplicar el concepto de *array*.
- Desarrollar algoritmos que implementen en su solución el concepto de *array*.

Enunciado: elementos duplicados

Se desea desarrollar un programa que, dado un *array* de números enteros, determine cuáles son sus elementos que se encuentran duplicados.

Instrucciones Java del ejercicio

Tabla 1.6. Instrucciones Java del ejercicio 1.5.

Instrucción	Descripción	Formato
length	Método que determina la longitud o tamaño de un *array*.	nombreArray.length()
&&	Operador AND lógico, devuelve verdadero cuando ambas condiciones son verdaderas; de otra manera, falso.	expresión1 && expresión2
!=	Operador diferente o no igual a. Devuelve verdadero si el valor del lado izquierdo no es igual al lado derecho.	variable1 != variable2

Solución

Clase: ElementosDuplicados

```
/**
 * Esta clase denominada ElementosDuplicados permite detectar cuáles
 * son los elementos duplicados en un array.
 * @version 1.0/2020
 */
```

```
public class ElementosDuplicados {

    /**
    * Método main
    */
    public static void main(String[] args) {
        int[] array = {1, 2, 3, 3, 4, 4, 5, 2}; /* Definición de un array de
            datos int */
        System.out.println("Elementos del array");
        // Imprime los elementos del array
        for (int i = 0; i < array.length; i++) {
            System.out.println("Elemento [" + i + "] = " + array[i]);
        }
        for (int i = 0; i < array.length - 1; i++) { /* Primer ciclo que recorre
            el array */
            for (int j = i+1; j < array.length; j++) { /* Segundo ciclo que
                recorre el array */
                /* Evalúa si los elementos son iguales y están en posiciones
                    diferentes */
                if ((array[i] == array[j]) && (i != j)) {
                System.out.println("Elemento duplicado: " + array[j]);
                }
            }
        }
    }
}
```

Diagrama de actividad

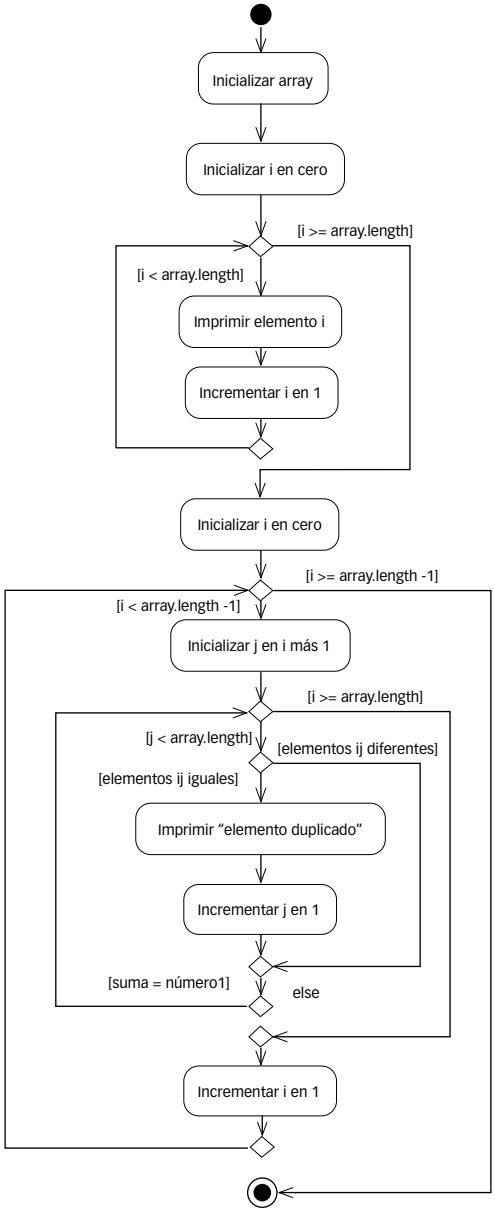


Figura 1.10. Diagrama de actividad del ejercicio 1.5.

Explicación del diagrama de actividad

El diagrama de actividad UML muestra un modelo del flujo de control del algoritmo para determinar los elementos duplicados de un *array* de valores enteros.

El diagrama posee una actividad inicial representada por medio de un círculo negro al inicio del programa.

Las diferentes instrucciones del programa se modelan como "actividades", las cuales se representan en UML como rectángulos con los bordes redondeados. El programa identifica como actividades las instrucciones de inicializar el *array* e inicializar la variable i.

El ciclo *for* que imprime los valores del *array* se representan por medio de una pareja de rombos que marcan el inicio y fin del ciclo.

Luego se presentan dos ciclos *for* anidados. El primero recorre el *array* desde la posición i = 0 hasta la longitud del *array* -1, y el segundo recorre el *array* desde j = i+1 hasta la longitud del *array*. En estos ciclos se evalúa que, si los elementos son iguales, se imprima el mensaje que el elemento está duplicado.

Cuando finalizan estos dos ciclos *for* anidados termina el programa, el cual se representa por medio de un círculo blanco y negro concéntrico.

Ejecución del programa

```
Elementos del array
Elemento [0] = 1
Elemento [1] = 2
Elemento [2] = 3
Elemento [3] = 3
Elemento [4] = 4
Elemento [5] = 4
Elemento [6] = 5
Elemento [7] = 2
Elemento duplicado: 2
Elemento duplicado: 3
Elemento duplicado: 4
```

Figura 1.11. Ejecución del programa del ejercicio 1.5.

Ejercicios propuestos

- Desarrollar un programa que determine el elemento mayor y menor de un *array* de enteros.
- Desarrollar un programa que, dado un número entero, busque dicho número en un *array*.
- Desarrollar un programa que busque elementos comunes en dos *arrays* de enteros.

Capítulo 2
Clases y objetos

El propósito general de este capítulo es hacer una introducción a los conceptos de clases y objetos que son el núcleo de la programación orientada a objetos en Java. Para ello, se presentan once ejercicios que cubren las nociones básicas del paradigma orientado a objetos que permitirán desarrollar programas, basados en una única clase, pero que incluyen conceptos fundamentales orientados a objetos.

- **Ejercicio 2.1. Definición de clases**

  La programación orientada a objetos proporciona varios conceptos y características para hacer que la creación y el uso de objetos sean fáciles y flexibles. Los conceptos más importantes son la clase y, sus instancias, los objetos. El primer ejercicio aborda la definición de clases y la creación de objetos.

- **Ejercicio 2.2. Definición de atributos de una clase con tipos primitivos de datos**

  Las clases se caracterizan por poseer atributos que las describen. Dichos atributos tienen asociado un cierto tipo primitivo de dato. El segundo ejercicio aplica la definición de atributos para una clase utilizando tipos primitivos de datos.

- **Ejercicio 2.3. Estado de un objeto**
  La colección de atributos con sus valores respectivos para un objeto en particular define el estado del objeto en un momento dado. La consulta del estado de un objeto puede hacerse utilizando el método *get*. A su vez, la modificación del estado del objeto puede realizarse a través del método *set*. El tercer ejercicio propone un problema para comprender el concepto de estado de un objeto.

- **Ejercicio 2.4. Definición de métodos con y sin valores de retorno**
  El segundo elemento más importante de una clase, además de sus atributos, son sus métodos. Los métodos permiten agregar diferentes niveles de comportamiento a los objetos. Estos métodos pueden tener o no un valor de retorno. El cuarto ejercicio aplica el concepto de definición de métodos con y sin valor de retorno.

- **Ejercicio 2.5. Definición de métodos con parámetros**
  Los métodos de una clase pueden contener una lista de parámetros, la cual es un conjunto de declaraciones de variables, separadas por comas, dentro de paréntesis. Estos parámetros se convierten en variables locales en el cuerpo del método. El objetivo del quinto ejercicio es comprender la definición de métodos con parámetros.

- **Ejercicio 2.6. Objetos como parámetros**
  Entre los parámetros de un método se pueden incluir, además de tipos primitivos de datos, instancias concretas de clases, es decir, objetos. El comportamiento de los objetos pasados como parámetros difiere de los tipos primitivos de datos pasados como parámetros. El sexto ejercicio hace énfasis en dichas diferencias.

- **Ejercicio 2.7. Métodos de acceso**
  Tanto los atributos como los métodos de una clase tienen un cierto ámbito de aplicación que permite que sean consultados y modificados por otras clases de acuerdo con los métodos de acceso definidos. El séptimo ejercicio pretende abarcar los diferentes métodos de acceso que se pueden asignar a los atributos y métodos de una clase.

- **Ejercicio 2.8. Asignación de objetos**
  Los objetos creados durante la ejecución de los programas pueden ser asignados a diferentes variables. Sin embargo, hay que considerar

que dichas variables no son objetos independientes, ya que comparten el mismo objeto. Para entender estos conceptos se plantea el octavo ejercicio.

- **Ejercicio 2.9. Variables locales dentro de un método**
  Cada método de una clase tiene un ámbito de ejecución delimitado tanto por sus parámetros como por las variables que se definen en el cuerpo del método. El noveno ejercicio pone en práctica la definición de variables locales en métodos.

- **Ejercicio 2.10. Sobrecarga de métodos**
  Los métodos de una clase pueden tener el mismo nombre, siempre que cada método tenga una signatura diferente, es decir, valores de retorno o lista de parámetros distintos. A esto se le llama sobrecarga de métodos; el décimo ejercicio está orientado a entender esta característica de la programación orientada a objetos.

- **Ejercicio 2.11. Sobrecarga de constructores**
  Los constructores, métodos especiales que permiten crear objetos, también se pueden sobrecargar. Por consiguiente, se pueden tener varios constructores con el mismo nombre y entre ellos se pueden invocar. El último ejercicio de este capítulo plantea un problema para afianzar dicho concepto.

Los diagramas UML utilizados en este capítulo son de clase y de objetos. Los diagramas de clase modelan la estructura estática de los programas. Los diagramas de clase de los ejercicios presentados generalmente están conformados por una o varias clases e incorporan notaciones para identificar tipos de atributos y métodos. Los diagramas de objeto identifican los objetos creados en el método *main* del programa.

Ejercicio 2.1. Definición de clases

Las clases son modelos del mundo real que capturan la estructura y comportamiento compartidos por una colección de objetos de un mismo tipo (Seidl *et al.*, 2015). Una clase está conformada por sus atributos y métodos. Una clase se define en Java como:

```
class NombreClase {
        lista de atributos
        lista de constructores
        lista de métodos
}
```

Un objeto se considera la instancia de una clase. Para crear un objeto se debe invocar a su constructor, el cual coincide con el nombre de la clase y se debe utilizar la palabra reservada *new*.

```
Clase objeto = new Clase();
```

Los constructores, además de permitir la instancia de objetos, realizan la inicialización de los atributos del objeto. Esto se logra pasando los valores de los atributos como parámetros en la invocación del constructor:

```
Clase nombreClase {
        tipo atributo;
        nombreClase(int parámetro, …) { // Constructor
            this.atributo = parámetro;
            …
        }
}
```

De otro lado, la palabra *this* se utiliza para referirse a los atributos de la clase y en particular, para diferenciar cuando los parámetros del constructor tienen el mismo nombre que los atributos.

El operador . (punto) permite acceder a los distintos atributos y métodos de una clase. El formato de la operación punto es:

```
objeto.atributo;
objeto.metódo();
```

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Entender los conceptos: clase, objeto, constructor y la referencia *this*.
- Comprender el significado de los atributos de un objeto.
- Realizar la instanciación de objetos.

Enunciado: clase Persona

Se requiere un programa que modele el concepto de una persona. Una persona posee nombre, apellido, número de documento de identidad y año de nacimiento. La clase debe tener un constructor que inicialice los valores de sus respectivos atributos.

La clase debe incluir los siguientes métodos:

- Definir un método que imprima en pantalla los valores de los atributos del objeto.
- En un método *main* se deben crear dos personas y mostrar los valores de sus atributos en pantalla.

Instrucciones Java del ejercicio

Tabla 2.1. Instrucciones Java del ejercicio 2.1.

Instrucción	Descripción	Formato
this	Palabra clave que se puede usar dentro de un método o constructor una clase. Funciona como una referencia al objeto actual.	*this*.atributo = parámetro;
void	Palabra clave que especifica que un método no tiene un valor de retorno.	*void* nombreMétodo() { }

Solución

Clase: Persona

```
/**
 * Esta clase define objetos de tipo Persona con un nombre, apellidos,
 * número de documento de identidad y año de nacimiento.
 * @version 1.2/2020
 */
public class Persona {

    String nombre; // Atributo que identifica el nombre de una persona
    String apellidos; // Atributo que identifica los apellidos de una persona
```

```
/* Atributo que identifica el número de documento de identidad de
    una persona */
String númeroDocumentoIdentidad;
int añoNacimiento; /* Atributo que identifica el año de nacimiento
    de una persona */

/**
* Constructor de la clase Persona
* @param nombre Parámetro que define el nombre de la persona
* @param apellidos Parámetro que define los apellidos de la persona
* @param númeroDocumentoIdentidad Parámetro que define el
* número del documento de identidad de la persona
* @param añoNacimiento Parámetro que define el año de nacimiento
* de la persona
*/
Persona(String nombre, String apellidos, String númeroDocumento
    Identidad, int añoNacimiento) {
    this.nombre = nombre;
    this.apellidos = apellidos;
    this.númeroDocumentoIdentidad=númeroDocumentoIdentidad;
    this.añoNacimiento = añoNacimiento;
}

/**
* Método que imprime en pantalla los datos de una persona
*/

void imprimir() {
    System.out.println("Nombre = " + nombre);
    System.out.println("Apellidos = " + apellidos);
    System.out.println("Número de documento de identidad = " +
        númeroDocumentoIdentidad);
    System.out.println("Año de nacimiento = " + añoNacimiento);
    System.out.println();
}

/**
* Método main que crea dos personas e imprime sus datos en pantalla
*/
```

```
        public static void main(String args[]) {
            Persona p1 = new Persona("Pedro","Pérez","1053121010",1998);
            Persona p2 = new Persona("Luis","León","1053223344",2001);
            p1.imprimir();
            p2.imprimir();
        }
    }
```

Diagrama de clases

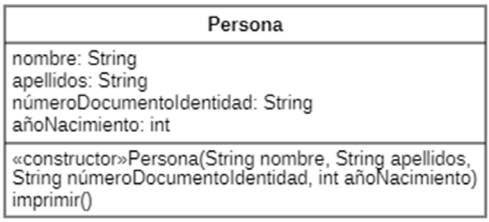


Figura 2.1. Diagrama de clases del ejercicio 2.1.

Explicación del diagrama de clases

Se ha definido una sola clase denominada Persona. El nombre de la clase se ubica en el primer compartimiento de la clase. En el segundo compartimiento se han definido los cuatro atributos junto con su tipo (nombre, apellidos y añoNacimiento de tipo *int* y númeroDocumentoIdentidad de tipo *String*). En el tercer compartimiento se han definido dos métodos, comenzando con el constructor, el cual tiene la etiqueta <<constructor>> como un estereotipo UML (brinda información adicional y personalizada) para su correcta identificación. El otro método es imprimir, el cual no contiene parámetros ni valor de retorno.

Diagrama de objetos

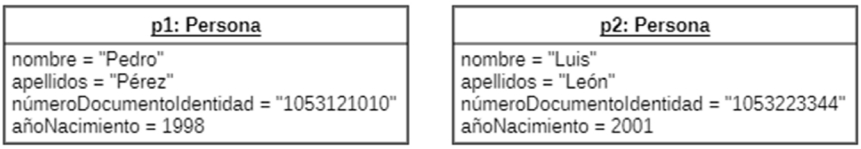


Figura 2.2. Diagrama de objetos del ejercicio 2.1.

Ejecución del programa

```
Nombre = Pedro
Apellidos = Pérez
Número de documento de identidad = 1053121010
Año de nacimiento = 1998

Nombre = Luis
Apellidos = León
Número de documento de identidad = 1053223344
Año de nacimiento = 2001
```

Figura 2.3. Ejecución del programa del ejercicio 2.1.

Ejercicios propuestos

- Agregar dos nuevos atributos a la clase Persona. Un atributo que represente el país de nacimiento de la persona (de tipo *String*) y otro que identifique el género de la persona, el cual debe representarse como un *char* con valores 'H' o 'M'.
- Modificar el constructor de la clase Persona para que inicialice estos dos nuevos atributos.
- Modificar el método imprimir de la clase Persona para que muestre en pantalla los valores de los nuevos atributos.

Ejercicio 2.2. Definición de atributos de una clase con tipos primitivos de datos

Las clases contienen una colección de atributos, que representan propiedades de los objetos. Dichos atributos tienen un tipo, el cual puede ser un tipo primitivo de dato de Java u objeto.

En la tabla 2.2 se presentan los tipos primitivos de datos en Java (Arroyo-Díaz, 2019a).

Tabla 2.2. Tipos primitivos de datos en Java

Tipo	Tamaño	Valores
byte	1 ***byte***	Valor entero entre -128 y 127
short	2 ***bytes***	Valor entero entre -32 768 y 32 767
int	4 ***bytes***	Valor entre 2 147 483 648 y 2 147 483 647
long	8 ***bytes***	Valor entre -9 223 372 036 854 775 808 y 9 223 372 036 854 775 807
float	4 ***bytes***	De -3.402823E38 a -1.401298E-45 y de 1.401298E-45 a 3.402823E38
double	8 ***bytes***	De -1.79769313486232 a -4.94065645841247E-324 y de 4.94065645841247E-324 a 1.79769313486232E308
Char	2 ***bytes***	Caracteres Unicode
boolean	1 ***bytes***	True y False

También existen los datos enumerados que representan un grupo de constantes con valores predefinidos. Se debe utilizar la palabra clave *enum* y separar las constantes con una coma. Los elementos enumerados deben estar en letras mayúsculas. El formato de los datos enumerados es:

enum variable {ELEMENTO1, ELEMENTO2, ELEMENTO3}

En este caso, la variable es un valor enumerado que puede asumir los valores ELEMENTO1, ELEMENTO2 o ELEMENTO3.

Objetivos de aprendizaje

Al finalizar este ejercicio el lector tendrá la capacidad para:

- Definir atributos de una clase con un tipo de primitivo de dato.
- Definir los valores iniciales de los atributos de una clase.
- Definir atributos de tipo enumerado.

Enunciado: clase Planeta

Se requiere un programa que modele el concepto de un planeta del sistema solar. Un planeta tiene los siguientes atributos:

- Un nombre de tipo *Sring* con valor inicial de *null*.
- Cantidad de satélites de tipo *int* con valor inicial de cero.
- Masa en kilogramos de tipo *double* con valor inicial de cero.

- Volumen en kilómetros cúbicos de tipo *double* con valor inicial de cero.
- Diámetro en kilómetros de tipo *int* con valor inicial de cero.
- Distancia media al Sol en millones de kilómetros, de tipo *int* con valor inicial de cero.
- Tipo de planeta de acuerdo con su tamaño, de tipo enumerado con los siguientes valores posibles: GASEOSO, TERRESTRE y ENANO.
- Observable a simple vista, de tipo booleano con valor inicial *false*.

La clase debe incluir los siguientes métodos:

- La clase debe tener un constructor que inicialice los valores de sus respectivos atributos.
- Definir un método que imprima en pantalla los valores de los atributos de un planeta.
- Calcular la densidad un planeta, como el cociente entre su masa y su volumen.
- Determinar si un planeta del sistema solar se considera exterior. Un planeta exterior está situado más allá del cinturón de asteroides. El cinturón de asteroides se encuentra entre 2.1 y 3.4 UA. Una unidad astronómica (UA) es la distancia entre la Tierra y el Sol= 149 597 870 Km.
- En un método *main* se deben crear dos planetas y mostrar los valores de sus atributos en pantalla. Además, se debe imprimir la densidad de cada planeta y si el planeta es un planeta exterior del sistema solar.

Instrucciones Java del ejercicio

Tabla 2.3. Instrucciones Java del ejercicio 2.2.

Instrucción	Descripción	Formato
return	Finaliza la ejecución de un método y puede utilizarse para devolver el valor de un método.	*return*; *return* variable;

Solución

Clase: Planeta

```
/**
 * Esta clase define objetos de tipo Planeta con un nombre, cantidad de
 * satélites, masa, volumen, diámetro, distancia al sol, tipo de planeta y si es
 * observable o no.
 * @version 1.2/2020
 */
public class Planeta {
    // Atributo que define el nombre de un planeta
    String nombre = null;
    // Atributo que define la cantidad de satélites que tiene un planeta
    int cantidadSatélites = 0;
    // Atributo que define la masa de un planeta
    double masa = 0;
    // Atributo que define el volumen de un planeta
    double volumen = 0;
    // Atributo que define el diámetro de un planeta
    int diámetro = 0;
    // Atributo que define la distancia al sol de un planeta
    int distanciaSol = 0;
    // Atributo que define el tipo de planeta como un valor enumerado
    enum tipoPlaneta {GASEOSO, TERRESTRE, ENANO}
    // Atributo que define el tipo de planeta
    tipoPlaneta tipo;
    // Atributo que define si el planeta es observable o no
    boolean esObservable = false;

    /**
    * Constructor de la clase Planeta
    * @param nombre Parámetro que define el nombre del planeta
    * @param cantidadSatélites Parámetro que define la cantidad de
    * satélites del planeta
    * @param masa Parámetro que define la masa del planeta (en
    * kilogramos)
    * @param volumen Parámetro que define el volumen del planeta
    * (en kilómetros cúbicos)
    * @param diámetro Parámetro que define el diámetro del planeta
    * (en kilómetros)
```

```
 * @param distanciaSol Parámetro que define la distancia media del
 * planeta al sol (en kilómetros)
 * @param tipo Parámetro que define el tipo de planeta (puede ser
 * GASEOSO, TERRESTRE o ENANO)
 * @param esObservable Parámetro que define si el planeta es
 * observable o no
 */

Planeta(String nombre, int cantidadSatélites, double masa, double
    volumen, int diámetro, int distanciaSol, tipoPlaneta tipo, boolean
    esObservable) {
    this.nombre = nombre;
    this.cantidadSatélites = cantidadSatélites;
    this.masa = masa;
    this.volumen = volumen;
    this.diámetro = diámetro;
    this.distanciaSol = distanciaSol;
    this.tipo = tipo;
    this.esObservable = esObservable;
}

/**
* Método que imprime en pantalla los datos de un planeta
*/
void imprimir() {
    System.out.println("Nombre del planeta = " + nombre);
    System.out.println("Cantidad de satélites = " + cantidadSatélites);
    System.out.println("Masa del planeta = " + masa);
    System.out.println("Volumen del planeta = " + volumen);
    System.out.println("Diámetro del planeta = " + diámetro);
    System.out.println("Distancia al sol = " + distanciaSol);
    System.out.println("Tipo de planeta = " + tipo);
    System.out.println("Es observable = " + esObservable);
}

/**
* Método que calcula y devuelve la densidad de un planeta
* @return La densidad calculada del planeta
*/

double calcularDensidad() {
    return masa/volumen;
}
```

```
    /**
    * Método que determina y devuelve si un planeta es exterior o no
    * @return Valor booleano que indica si el planeta es exterior o no
    */
    boolean esPlanetaExterior(){
        float límite = (float) (149597870 * 3.4);
        /* Un planeta exterior está situado más allá del cinturón de
            asteroides */
        /* El cinturón se encuentra entre 2,1 y 3,4 UA (UA =
            149.597.870 Km) */
        if (distanciaSol > límite) {
            return true;
        } else {
            return false;
        }
    }
    /**
    * Método main que crea dos planetas, imprime sus datos en pantalla,
    * determina su densidad y si son planetas exteriores
    */
    public static void main(String args[]) {
        Planeta p1 = new Planeta("Tierra",1,5.9736E24,1.0832
            1E12,12742,150000000,tipoPlaneta.TERRESTRE,true);
        p1.imprimir();
        System.out.println("Densidad del planeta = " +
            p1.calcularDensidad());
        System.out.println("Es planeta exterior = " +
            p1.esPlanetaExterior());
        System.out.println();
        Planeta p2 = new Planeta("Júpiter",79,1.899E27,1.431
            3E15,139820,750000000,tipoPlaneta.gaseoso,true);
        p2.imprimir();
        System.out.println("Densidad del planeta = " +
            p2.calcularDensidad());
        System.out.println("Es planeta exterior = " +
            p2.esPlanetaExterior());
    }
}
```

Diagrama de clases

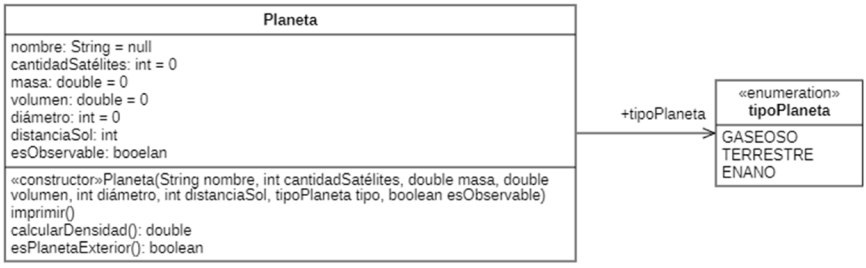


Figura 2.4. Diagrama de clases del ejercicio 2.4.

Explicación del diagrama de clases

Se ha definido una clase denominada Planeta con sus respectivos atributos que representan el nombre del planeta, cantidad de satélites que tiene, su masa, volumen, diámetro, distancia al Sol y si es observable o no. Para cada atributo, además de su tipo, se ha agregado su correspondiente valor inicial o por defecto. También se han definido sus métodos respectivos: un constructor que inicializa los valores de sus atributos, un método para imprimir los valores de sus atributos en pantalla, otro para calcular la densidad (que devuelve un valor de tipo *double*) y un último método para determinar si es un planeta exterior (que devuelve un valor booleano).

El atributo para identificar el tipo de planeta se expresa como una asociación entre la clase Planeta y la clase TipoPlaneta cuyo nombre es el nombre del atributo. En UML, las variables *enum* se identifican con el estereotipo <<enumeration>>, que representan un conjunto de valores constantes identificados como atributos en su segundo compartimiento.

Diagrama de objetos

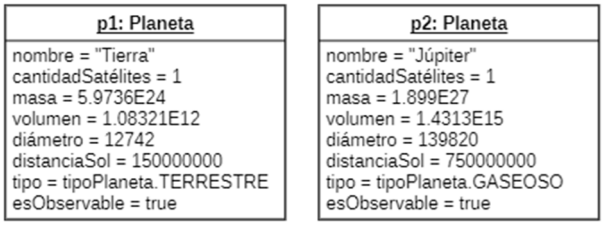


Figura 2.5. Diagrama de objetos del ejercicio 2.2.

Ejecución del programa

```
Nombre del planeta = Tierra
Cantidad de satélites = 1
Masa del planeta = 5.9736E24
Volumen del planeta = 1.08321E12
Diámetro del planeta = 12742
Distancia al sol = 150000000
Tipo de planeta = TERRESTRE
Es observable = true
Densidad del planeta = 5.514720137369484E12
Es planeta exterior = false

Nombre del planeta = Júpiter
Cantidad de satélites = 79
Masa del planeta = 1.899E27
Volumen del planeta = 1.4313E15
Diámetro del planeta = 139820
Distancia al sol = 750000000
Tipo de planeta = GASEOSO
Es observable = true
Densidad del planeta = 1.3267658771745964E12
Es planeta exterior = true
```

Figura 2.6. Ejecución del programa del ejercicio 2.2.

Ejercicios propuestos

- Agregar dos atributos a la clase Planeta. El primero debe representar el periodo orbital del planeta (en años). El segundo atributo representa el periodo de rotación (en días).
- Modificar el constructor de la clase para que inicialice los valores de estos dos nuevos atributos.
- Modificar el método imprimir para que muestre en pantalla los valores de los nuevos atributos.

Ejercicio 2.3. Estado de un objeto

Se denomina estado de un objeto al conjunto de pares *{atributo = valor de un objeto}*. El estado de un objeto puede cambiar a lo largo de su vida a medida que se vayan ejecutando sus métodos (Arroyo-Díaz, 2019a).

Las clases tienen dos tipos de métodos: *get* y *set*.

- Los métodos *get* permiten obtener el valor de un atributo de un objeto. Los métodos *get* se definen con el siguiente formato:

getNombreAtributo

- Los métodos *set* permiten asignar o cambiar el valor de un atributo de un objeto y, por lo tanto, cambian su estado. Los métodos *set* se definen con el siguiente formato:

setNombreAtributo(tipo parámetro)

Los métodos *get* y *set* tienen una estructura que se repite para cada uno de los atributos. Por ello, los entornos de desarrollo actuales permiten generar automáticamente dichos métodos.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Entender el concepto de estado de un objeto.
- Definir métodos *get* y *set* de una clase.

Enunciado: clase Automóvil

Se requiere un programa que modele el concepto de un automóvil. Un automóvil tiene los siguientes atributos:

- Marca: el nombre del fabricante.
- Modelo: año de fabricación.
- Motor: volumen en litros del cilindraje del motor de un automóvil.
- Tipo de combustible: valor enumerado con los posibles valores de gasolina, bioetanol, diésel, biodiésel, gas natural.
- Tipo de automóvil: valor enumerado con los posibles valores de carro de ciudad, subcompacto, compacto, familiar, ejecutivo, SUV.
- Número de puertas: cantidad de puertas.
- Cantidad de asientos: número de asientos disponibles que tiene el vehículo.
- Velocidad máxima: velocidad máxima sostenida por el vehículo en km/h.
- Color: valor enumerado con los posibles valores de blanco, negro, rojo, naranja, amarillo, verde, azul, violeta.
- Velocidad actual: velocidad del vehículo en un momento dado.

La clase debe incluir los siguientes métodos:

- Un constructor para la clase Automóvil donde se le pasen como parámetros los valores de sus atributos.
- Métodos *get* y *set* para la clase Automóvil.
- Métodos para acelerar una cierta velocidad, desacelerar y frenar (colocar la velocidad actual en cero). Es importante tener en cuenta que no se debe acelerar más allá de la velocidad máxima permitida para el automóvil. De igual manera, tampoco es posible desacelerar a una velocidad negativa. Si se cumplen estos casos, se debe mostrar por pantalla los mensajes correspondientes.
- Un método para calcular el tiempo estimado de llegada, utilizando como parámetro la distancia a recorrer en kilómetros. El tiempo estimado se calcula como el cociente entre la distancia a recorrer y la velocidad actual.
- Un método para mostrar los valores de los atributos de un Automóvil en pantalla.
- Un método *main* donde se deben crear un automóvil, colocar su velocidad actual en 100 km/h, aumentar su velocidad en 20 km/h, luego decrementar su velocidad en 50 km/h, y después frenar. Con cada cambio de velocidad, se debe mostrar en pantalla la velocidad actual.

Solución

Clase: Automóvil

```
/**
 * Esta clase define objetos de tipo Automóvil con una marca, modelo,
 * motor, tipo de combustible, tipo de automóvil, número de puertas,
 * cantidad de asientos, velocidad máxima, color y velocidad actual.
 * @version 1.2/2020
 */
public class Automóvil {
    // Atributo que define la marca de un automóvil
```

```
String marca;
// Atributo que define el modelo de un automóvil
int modelo;
// Atributo que define el motor de un automóvil
int motor;
// Tipo de combustible como un valor enumerado
enum tipoCom {GASOLINA, BIOETANOL, DIESEL, BIODISESEL,
    GAS_NATURAL}
// Atributo que define el tipo de combustible
tipoCom tipoCombustible;
// Tipo de automóvil como un valor enumerado
enum tipoA {CIUDAD, SUBCOMPACTO, COMPACTO, FAMILIAR,
    EJECUTIVO, SUV}
// Atributo que define el tipo de automóvil
tipoA tipoAutomóvil;
// Atributo que define el número de puertas de un automóvil
int númeroPuertas;
// Atributo que define la cantidad de asientos de un automóvil
int cantidadAsientos;
// Atributo que define la velocidad máxima de un automóvil
int velocidadMáxima;
// Color del automóvil como un valor enumerado
enum tipoColor {BLANCO, NEGRO, ROJO, NARANJA,
    AMARILLO, VERDE, AZUL, VIOLETA}
// Atributo que define el color de un automóvil
tipoColor color;
// Atributo que define la velocidad de un automóvil
int velocidadActual = 0;

/**
* Constructor de la clase Automóvil
* @param marca Parámetro que define la marca de un automóvil
* @param modelo Parámetro que define el modelo (año de
*fabricación) de un automóvil
* @param motor Parámetro que define el volumen del cilindraje del
* motor (puede ser gasolina, bioetanol, diésel, biodiesel o gas natural)
* @param tipoAutomóvil Parámetro que define el tipo de automóvil
* (puede ser Carro de ciudad, Subcompacto, Compacto, Familiar,
* Ejecutivo o SUV)
* @param númeroPuertas Parámetro que define el número de
* puertas de un automóvil
```

```
* @param cantidadAsientos Parámetro que define la cantidad de
* asientos que tiene el automóvil
* @param velocidadMáxima Parámetro que define la velocidad
* máxima permitida al automóvil
* @param color Parámetro que define el color del automóvil (puede
* ser Blanco, Negro, Rojo, Naranja, Amarillo, Verde, Azul o Violeta)
*/

Automóvil(String marca, int modelo, int motor, tipoCom
    tipoCombustible, tipoA tipoAutomóvil, int númeroPuertas, int
    cantidadAsientos, int velocidadMáxima, tipoColor color) {
    this.marca = marca;
    this.modelo = modelo;
    this.motor = motor;
    this.tipoCombustible = tipoCombustible;
    this.tipoAutomóvil = tipoAutomóvil;
    this.númeroPuertas = númeroPuertas;
    this.cantidadAsientos = cantidadAsientos;
    this.velocidadMáxima = velocidadMáxima;
    this.color = color;
}

/**
* Método que devuelve la marca de un automóvil
* @return La marca de un automóvil
*/

String getMarca() {
    return marca;
}

/**
* Método que devuelve el modelo de un automóvil
* @return El modelo de un automóvil
*/
int getModelo() {
    return modelo;
}
```

```
    /**
    * Método que devuelve el volumen en litros del cilindraje del motor
    * de un automóvil
    * @return El volumen en litros del cilindraje del motor de un
    * automóvil
    */
    int getMotor() {
        return motor;
    }

    /**
    * Método que devuelve el tipo de combustible utilizado por el motor
    * de un automóvil
    * @return El tipo de combustible utilizado por el motor de un
    * automóvil
    */
    tipoCom getTipoCombustible() {
        return tipoCombustible;
    }

    /**
    * Método que devuelve el tipo de automóvil
    * @return El tipo de automóvil
    */
    tipoA getTipoAutomóvil() {
        return tipoAutomóvil;
    }

    /**
    * Método que devuelve el número de puertas de un automóvil
    * @return El número de puertas que tiene un automóvil
    */
    int getNúmeroPuertas() {
        return númeroPuertas;
}

    /**
    * Método que devuelve la cantidad de asientos de un automóvil
    * @return La cantidad de asientos que tiene un automóvil
    */
    int getCantidadAsientos() {
        return cantidadAsientos;
    }
```

```
/**
* Método que devuelve la velocidad máxima de un automóvil
* @return La velocidad máxima de un automóvil
*/
int getVelocidadMáxima() {
    return velocidadMáxima;
}

/**
* Método que devuelve el color de un automóvil
* @return El color de un automóvil
*/
tipoColor getColor() {
    return color;
}

/**
* Método que devuelve la velocidad actual de un automóvil
* @return La velocidad actual de un automóvil
*/
int getVelocidadActual() {
    return velocidadActual;
}

/**
* Método que establece la marca de un automóvil
* @param marca Parámetro que define la marca de un automóvil
*/
void setMarca(String marca) {
    this.marca = marca;
}

/**
* Método que establece el modelo de un automóvil
* @param modelo Parámetro que define el modelo de un automóvil
*/
void setModelo(int modelo) {
    this.modelo = modelo;
}

/**
* Método que establece el volumen en litros del motor de un automóvil
* @param motor Parámetro que define el volumen en litros del
* motor de un automóvil
*/
```

```
void setMotor(int motor) {
    this.motor = motor;
}

/**
* Método que establece el tipo de combustible de un automóvil
* @param tipoCombustible Parámetro que define el tipo de
* combustible de un automóvil
*/
void setTipoCombustible(tipoCom tipoCombustible) {
    this.tipoCombustible = tipoCombustible;
}

/**
* Método que establece el tipo de automóvil
* @param tipoAutomóvil Parámetro que define el tipo de automóvil
*/
void setTipoAutomóvil(tipoA tipoAutomóvil) {
    this.tipoAutomóvil = tipoAutomóvil;
}

/**
* Método que establece el número de puertas de un automóvil
* @param númeroPuertas Parámetro que define el número de
* puertas de un automóvil
*/
void setNúmeroPuertas(int númeroPuertas) {
    this.númeroPuertas = númeroPuertas;
}

/**
* Método que establece la cantidad de asientos de un automóvil
* @param cantidadAsientos Parámetro que define la cantidad de
* asientos de un automóvil
*/
void setCantidadAsientos(int cantidadAsientos) {
    this.cantidadAsientos = cantidadAsientos;
}

/**
* Método que establece la velocidad máxima de un automóvil
* @param velocidadMáxima Parámetro que define la velocidad
* máxima de un automóvil
```

```
*/
void setVelocidadMáxima(int velocidadMáxima) {
    this.velocidadMáxima = velocidadMáxima;
}

/**
* Método que establece el color de un automóvil
* @param color Parámetro que define el color de un automóvil
*/
void setColor(tipoColor color) {
    this.color = color;
}

/**
* Método que establece la velocidad de un automóvil
* @param velocidadActual Parámetro que define la velocidad actual
* de un automóvil
*/
void setVelocidadActual(int velocidadActual) {
    this.velocidadActual = velocidadActual;
}

/**
* Método que incrementa la velocidad de un automóvil
* @param incrementoVelocidad Parámetro que define la cantidad a
* incrementar en la velocidad actual de un automóvil
*/
void acelerar(int incrementoVelocidad) {
    if (velocidadActual + incrementoVelocidad < velocidadMáxima) {
        /* Si el incremento de velocidad no supera la velocidad
            máxima */
        velocidadActual = velocidadActual + incrementoVelocidad;
    } else { /* De otra manera no se puede incrementar la velocidad y
        se genera mensaje */
    System.out.println("No se puede incrementar a una velocidad
        superior a la máxima del automóvil.");
    }
}

/**
* Método que decrementa la velocidad de un automóvil
* @param marca Parámetro que define la cantidad a decrementar en
* la velocidad actual de un automovil
*/
```

```
    void desacelerar(int decrementoVelocidad) {
        /* La velocidad actual no se puede decrementar alcanzando un
           valor negativo */
        if ((velocidadActual - decrementoVelocidad) > 0) {
            velocidadActual = velocidadActual - decrementoVelocidad;
        } else { /* De otra manera no se puede decrementar la velocidad y
           se genera mensaje */
            System.out.println("No se puede decrementar a una velocidad
               negativa.");
        }
    }

    /**
    * Método que coloca la velocidad actual de un automóvil en cero
    */
    void frenar() {
        velocidadActual = 0;
    }

    /**
    * Método que calcula el tiempo que tarda un automóvil en recorrer
    * cierta distancia
    * @param distancia Parámetro que define la distancia a recorrer por
    * el automóvil (en kilómetros)
    */
    double calcularTiempoLlegada(int distancia) {
        return distancia/velocidadActual;
    }

    /**
    * Método que imprime en pantalla los valores de los atributos de un
        automóvil
    */
    void imprimir() {
        System.out.println("Marca = " + marca);
        System.out.println("Modelo = " + modelo);
        System.out.println("Motor = " + motor);
        System.out.println("Tipo de combustible = " + tipoCombustible);
        System.out.println("Tipo de automóvil = " + tipoAutomóvil);
        System.out.println("Número de puertas = " + númeroPuertas);
```

```
        System.out.println("Cantidad de asientos = " +
            cantidadAsientos);
        System.out.println("Velocida máxima = " + velocidadMáxima);
        System.out.println("Color = " + color);
    }

    /**
    * Método main que crea un automóvil, imprime sus datos en
    * pantalla y realiza varios cambios en su velocidad
    */
    public static void main(String args[]) {
        Automóvil auto1 = new
Automóvil("Ford",2018,3,tipoCom.DIESEL,tipoA.EJECUTIVO,5,6,250,
            tipoColor.NEGRO);
        auto1.imprimir();
        auto1.setVelocidadActual(100);
        System.out.println("Velocidad actual = " + auto1.
            velocidadActual);
        auto1.acelerar(20);
        System.out.println("Velocidad actual = " + auto1.
            velocidadActual);
        auto1.desacelerar(50);
        System.out.println("Velocidad actual = " + auto1.
            velocidadActual);
        auto1.frenar();
        System.out.println("Velocidad actual = " + auto1.
            velocidadActual);
        auto1.desacelerar(20);
    }
}
```

Diagrama de clases

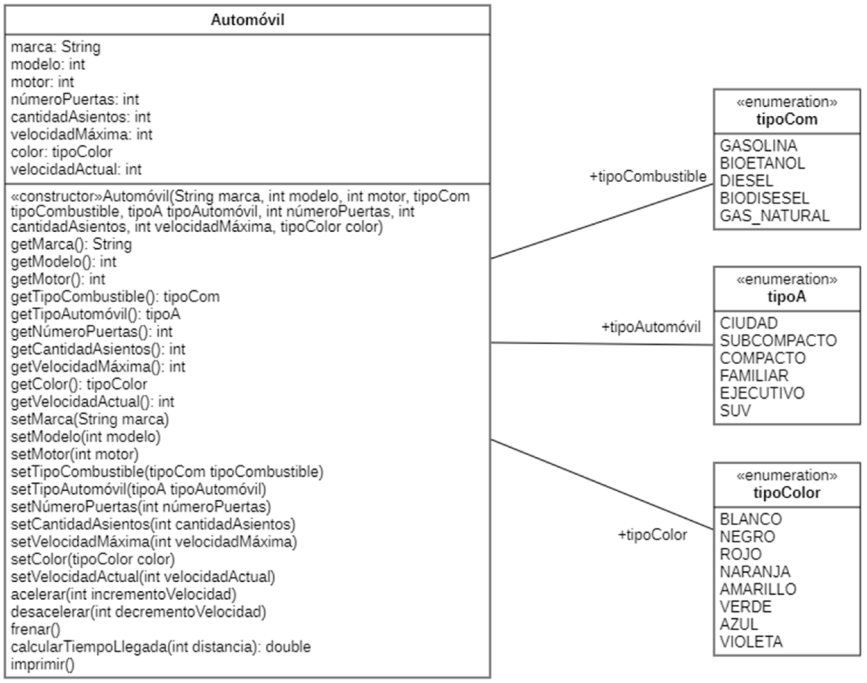


Figura 2.7. Diagrama de clases del ejercicio 2.3.

Explicación del diagrama de clases

Se ha definido una clase denominada Automóvil con sus respectivos atributos y métodos. El compartimiento de métodos incluye los métodos *get* y *set* para cada atributo de la clase. Los atributos de tipo enumerado se modelan como asociaciones entre la clase Automóvil y diferentes clases independientes con el estereotipo <<enumeration>>, cada una de ellas con su respectiva lista de constantes definidas en el compartimiento de atributos de la clase.

La clase Automóvil posee atributos como: marca (de tipo *String*), modelo (de tipo *int*), motor (de tipo *int*), tipo de combustible (valor enumerado que se representa con la asociación a la clase tipoCom de tipo <<enumaration>>), tipo de automóvil (valor enumerado que se representa con la asociación a la clase tipoA de tipo <<enumaration>>), número de puertas (de tipo *int*), cantidad de asientos (de tipo *int*), número de asientos

disponibles que tiene el vehículo (de tipo *int*), velocidad máxima (de tipo *int*), color (valor enumerado que se representa con la asociación a la clase tipoColor de tipo <<enumaration>>) y velocidad actual (de tipo *int*).

La clase Automóvil cuenta además de los métodos *get* y *set*, con un constructor para inicializar los valores de todos sus atributos y métodos para acelerar y desacelerar a cierta velocidad, frenar el automóvil, calcular tiempo de llegada para recorrer una cierta distancia y para imprimir los valores de los atributos en pantalla.

Diagrama de objetos

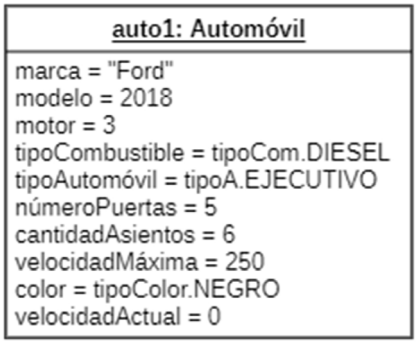


Figura 2.8. Diagrama de objetos del ejercicio 2.3.

Ejecución del programa

```
Marca = Ford
Modelo = 2018
Motor = 3
Tipo de combustible = DIESEL
Tipo de automóvil = EJECUTIVO
Número de puertas = 5
Cantidad de asientos = 6
Velocida máxima = 250
Color = NEGRO
Velocidad actual = 100
Velocidad actual = 120
Velocidad actual = 70
Velocidad actual = 0
No se puede decrementar a una velocidad negativa.
```

Figura 2.9. Ejecución del programa del ejercicio 2.3.

Ejercicios propuestos

- Agregar a la clase Automóvil, un atributo para determinar si el vehículo es automático o no. Agregar los métodos *get* y *set* para dicho atributo. Modificar el constructor para inicializar dicho atributo.
- Modificar el método acelerar para que si la velocidad máxima se sobrepase se genere una multa. Dicha multa se puede incrementar cada vez que el vehículo intenta superar la velocidad máxima permitida.
- Agregar un método para determinar si un vehículo tiene multas y otro método para determinar el valor total de multas de un vehículo.

Ejercicio 2.4. Definición de métodos con y sin valores de retorno

Los métodos de una clase con y sin valores de retorno tienen el siguiente formato, el cual se denomina signatura del método (Deitel y Deitel, 2017):

tipoRetorno nombreMétodo(tipo parámetro1, tipo parámetro2, …)

- **tipoRetorno**: representa el tipo de retorno del método, el cual puede ser un tipo de dato primitivo, un objeto, o si no devuelve nada debe colocarse la palabra reservada *void*.
- **nombreMétodo**: representa el nombre del método.
- **Parámetros**: es un listado de parámetros separados por comas, que se utilizan en el cuerpo del método para cumplir la finalidad del método. Para cada parámetro se debe colocar previamente su tipo de dato. Un método puede no tener parámetros.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Definir métodos con y sin valores de retorno en una clase.
- Definir un programa conformado por varias clases y una clase de prueba que contenga el método *main*.

Enunciado: clases sobre figuras geométricas

Se requiere un programa que modele varias figuras geométricas: el círculo, el rectángulo, el cuadrado y el triángulo rectángulo.

- El círculo tiene como atributo su radio en centímetros.
- El rectángulo, su base y altura en centímetros.
- El cuadrado, la longitud de sus lados en centímetros.
- El triángulo, su base y altura en centímetros.

Se requieren métodos para determinar el área y el perímetro de cada figura geométrica. Además, para el triángulo rectángulo se requiere:

- Un método que calcule la hipotenusa del rectángulo.
- Un método para determinar qué tipo de triángulo es:

  Equilátero: todos sus lados son iguales.

  Isósceles: tiene dos lados iguales.

  Escaleno: todos sus lados son diferentes.

Se debe desarrollar una clase de prueba con un método *main* para crear las cuatro figuras y probar los métodos respectivos.

Instrucciones Java del ejercicio

Tabla 2.4. Instrucciones Java del ejercicio 2.4.

Instrucción	Descripción	Formato
Math.PI	Un valor *double* que representa la variable matemática pi = 3.1416, la relación entre el radio de la circunferencia y su diámetro.	Math.PI
Math.pow	Método que devuelve el valor del primer argumento elevado a la potencia del segundo argumento.	*Math.pow*(argumento1, argumento2);
&&	Operador lógico AND. Devuelve solamente un valor verdadero cuando las dos expresiones son verdaderas, en caso contrario devuelve falso.	(expresión) && (expresión2)

Solución

Clase: Círculo

```
/**
 * Esta clase define objetos de tipo Círculo con su radio como atributo.
 * @version 1.2/2020
 */
public class Círculo {
    int radio; // Atributo que define el radio de un círculo

    /**
    * Constructor de la clase Círculo
    * @param radio Parámetro que define el radio de un círculo
    */
    Círculo(int radio) {
        this.radio = radio;
    }

    /**
    * Método que calcula y devuelve el área de un círculo como pi
    * multiplicado por el radio al cuadrado
    * @return Área de un círculo
    */
    double calcularArea() {
        return Math.PI*Math.pow(radio,2);
    }

    /**
    * Método que calcula y devuelve el perímetro de un círculo como la
    * multiplicación de pi por el radio por 2
    * @return Perímetro de un círculo
    */
    double calcularPerímetro() {
        return 2*Math.PI*radio;
    }
}
```

Clase: Rectángulo

```
/**
* Esta clase define objetos de tipo Rectángulo con una base y una
* altura como atributos.
* @version 1.2/2020
*/
public class Rectángulo {
    int base; // Atributo que define la base de un rectángulo
    int altura; // Atributo que define la altura de un rectángulo

    /**
    * Constructor de la clase Rectangulo
    * @param base Parámetro que define la base de un rectángulo
    * @param altura Parámetro que define la altura de un rectángulo
    */
    Rectángulo(int base, int altura) {
        this.base = base;
        this.altura = altura;
    }

    /**
    * Método que calcula y devuelve el área de un rectángulo como la
    * multiplicación de la base por la altura
    * @return Área de un rectángulo
    */
    double calcularArea() {
        return base * altura;
    }

    /**
    * Método que calcula y devuelve el perímetro de un rectángulo
    * como (2 * base) + (2 * altura)
    * @return Perímetro de un rectángulo
    */
    double calcularPerímetro() {
        return (2 * base) + (2 * altura);
    }
}
```

Clase: Cuadrado

```
/**
 * Esta clase define objetos de tipo Cuadrado con un lado como atributo.
 * @version 1.2/2020
 */
public class Cuadrado {
    int lado; // Atributo que define el lado de un cuadrado

    /**
    * Constructor de la clase Cuadrado
    * @param lado Parámetro que define la longitud de la base de un
    * cuadrado
    */
    public Cuadrado(int lado) {
        this.lado = lado;
    }

    /**
    * Método que calcula y devuelve el área de un cuadrado como el
    * lado elevado al cuadrado
    * @return Área de un Cuadrado
    */
    double calcularArea() {
        return lado*lado;
    }

    /**
    * Método que calcula y devuelve el perímetro de un cuadrado como
    * cuatro veces su lado
    * @return Perímetro de un cuadrado
    */
    double calcularPerímetro() {
        return (4*lado);
    }
}
```

Clase: TriánguloRectángulo

```
/**
* Esta clase define objetos de tipo Triángulo Rectángulo con una
* base y una altura como atributos.
* @version 1.2/2020
*/
public class TriánguloRectángulo {
    int base; // Atributo que define la base de un triángulo rectángulo
    int altura; // Atributo que define la altura de un triángulo rectángulo

    /**
    * Constructor de la clase TriánguloRectángulo
    * @param base Parámetro que define la base de un triángulo
    * rectángulo
    * @param altura Parámetro que define la altura de un triángulo
    * rectángulo
    */
    public TriánguloRectángulo(int base, int altura) {
        this.base = base;
        this.altura = altura;
    }

    /**
    * Método que calcula y devuelve el área de un triángulo rectángulo
    * como la base multiplicada por la altura sobre 2
    * @return Área de un triángulo rectángulo
    */
    double calcularArea() {
        return (base * altura / 2);
    }

    /**
    * Método que calcula y devuelve el perímetro de un triángulo
    * rectángulo como la suma de la base, la altura y la hipotenusa
    * @return Perímetro de un triángulo rectángulo
    */
    double calcularPerímetro() {
        return (base + altura + calcularHipotenusa()); /* Invoca al
            método calcular hipotenusa */
    }
```

```
    /**
    * Método que calcula y devuelve la hipotenusa de un triángulo
    * rectángulo utilizando el teorema de Pitágoras
    * @return Hipotenusa de un triángulo rectángulo
    */
    double calcularHipotenusa() {
        return Math.pow(base*base + altura*altura, 0.5);
    }

    /**
    * Método que determina si un triángulo es:
    * - Equilatero: si sus tres lados son iguales
    * - Escaleno: si sus tres lados son todos diferentes
    * - Escaleno: si dos de sus lados son iguales y el otro es diferente de
    * los demás
    */
    void determinarTipoTriángulo() {
        if ((base == altura) && (base == calcularHipotenusa()) && (altura
            == calcularHipotenusa()))
            System.out.println("Es un triángulo equilátero"); /* Todos sus
                lados son iguales */
        else if ((base != altura) && (base != calcularHipotenusa()) &&
            (altura != calcularHipotenusa()))
            System.out.println("Es un triángulo escaleno"); /* Todos sus
                lados son diferentes */
        else
            System.out.println("Es un triángulo isósceles"); /* De otra
                manera, es isósceles */
        }
}
```

Clase: PruebaFiguras

```
/**
 * Esta clase prueba diferentes acciones realizadas en diversas figuras
 * geométricas.
 * @version 1.2/2020
 */
public class PruebaFiguras {
```

```
    /**
    * Método main que crea un círculo, un rectángulo, un cuadrado y
    * un triángulo rectángulo. Para cada uno de estas figuras geométricas,
    * se calcula su área y perímetro.
    */
    public static void main(String args[]) {
        Círculo figura1 = new Círculo(2);
        Rectángulo figura2 = new Rectángulo(1,2);
        Cuadrado figura3 = new Cuadrado(3);
        TriánguloRectángulo figura4 = new TriánguloRectángulo(3,5);

        System.out.println("El área del círculo es = " + figura1.
            calcularArea());
        System.out.println("El perímetro del círculo es = " + figura1.
            calcularPerímetro());
        System.out.println();
        System.out.println("El área del rectángulo es = " + figura2.
            calcularArea());
        System.out.println("El perímetro del rectángulo es = " + figura2.
            calcularPerímetro());
        System.out.println();
        System.out.println("El área del cuadrado es = " + figura3.
            calcularArea());
        System.out.println("El perímetro del cuadrado es = " + figura3.
            calcularPerímetro());
        System.out.println();
        System.out.println("El área del triángulo es = " + figura4.
            calcularArea());
        System.out.println("El perímetro del triángulo es = " + figura4.
            calcularPerímetro());
        figura4.determinarTipoTriángulo();
    }
}
```

Diagrama de clases

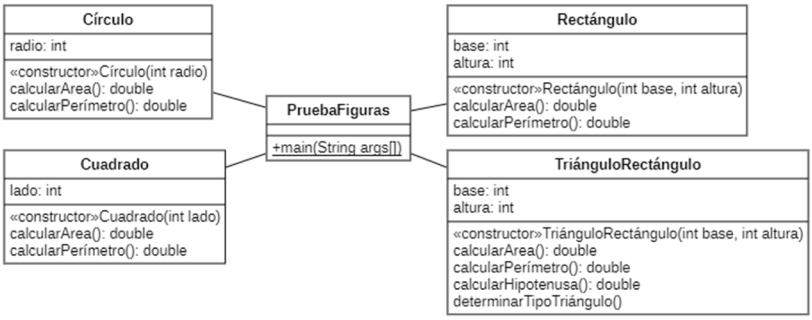


Figura 2.10. Diagrama de clases del ejercicio 2.4.

Explicación del diagrama de clases

Se ha definido una clase para cada figura geométrica (Círculo, Cuadrado, Rectángulo y TriánguloRectángulo). Cada clase contiene sus atributos respectivos. La clase Círculo tiene su radio como atributo (de tipo *int*). La clase Cuadrado tiene la longitud de sus lados como atributo (de tipo *int*). La clase Rectángulo tiene como atributos su base y altura (ambos de tipo *int*). La clase TriánguloRectángulo, su base y altura (ambos de tipo *int*). La hipotenusa no es atributo, en este caso será un valor que será calculado por medio de un método.

Todas las clases poseen los mismos métodos: un constructor y métodos para calcular el área y calcular el perímetro que devuelven valores de tipo *double*. La clase TriánguloRectángulo tiene dos métodos adicionales para calcular el valor de la hipotenusa y para determinar qué tipo de triángulo es (de acuerdo con la longitud de sus lados).

La clase PruebaFiguras es una clase que no tiene significado en el dominio conceptual del problema. Como su nombre lo indica se ha definido para definir un único método *main* que sirva como punto de entrada para la ejecución del programa. El método *main* es un método estático cuya representación en UML es subrayando la signatura del método.

Diagrama de objetos

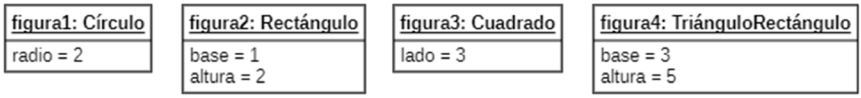


Figura 2.11. Diagrama de objetos del ejercicio 2.4.

Ejecución del programa

```
El área del círculo es = 12.566370614359172
El perímetro del círculo es = 12.566370614359172

El área del rectángulo es = 2.0
El perímetro del rectángulo es = 6.0

El área del cuadrado es = 9.0
El perímetro del cuadrado es = 12.0

El área del triángulo es = 7.0
El perímetro del triángulo es = 13.8309518948453
Es un triángulo escaleno
```

Figura 2.12. Ejecución del programa del ejercicio 2.4.

Ejercicios propuestos

- Agregar una nueva clase denominada Rombo. Definir los métodos para calcular el área y el perímetro de esta nueva figura geométrica.
- Agregar una nueva clase denominada Trapecio. Definir los métodos para calcular el área y el perímetro de esta nueva figura geométrica.

Ejercicio 2.5. Definición de métodos con parámetros

Los métodos de una clase pueden tener parámetros, los cuales son datos que se requieren para la ejecución del método. Un método puede no contener ningún parámetro, un solo parámetro o muchos parámetros.

Se recomienda que la cantidad de parámetros sea la adecuada para el correcto funcionamiento del método, es decir, que no tenga demasiados parámetros.

Objetivo de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para definir métodos con parámetros de entrada.

Enunciado: clase CuentaBancaria

Se requiere un programa que modele una cuenta bancaria que posee los siguientes atributos:

- Nombres del titular.
- Apellidos del titular.
- Número de la cuenta bancaria.
- Tipo de cuenta: puede ser una cuenta de ahorros o una cuenta corriente.
- Saldo de la cuenta.

Se debe definir un constructor que inicialice los atributos de la clase. Cuando se crea una cuenta bancaria, su saldo inicial tiene un valor de cero.

En una determinada cuenta bancaria se puede:

- Imprimir por pantalla los valores de los atributos de una cuenta bancaria.
- Consultar el saldo de una cuenta bancaria.
- Consignar un determinado valor en la cuenta bancaria, actualizando el saldo correspondiente.
- Retirar un determinado valor de la cuenta bancaria, actualizando el saldo correspondiente. Es necesario tener en cuenta que no se puede realizar el retiro si el valor solicitado supera el saldo actual de la cuenta.

Solución

Clase: CuentaBancaria

```
/**
 * Esta clase define objetos que representan una cuenta bancaria que
 * tienen un nombre y apellidos del titular, un número de cuenta, un
 * tipo de cuenta (ahorros o corriente) y un saldo.
 * @version 1.2/2020
 */
public class CuentaBancaria {
    // Atributo que define los nombres del titular de la cuenta bancaria
    String nombresTitular;
    // Atributo que define los apellidos del titular de la cuenta bancaria
    String apellidosTitular;
    // Atributo que define el número de la cuenta bancaria
    int númeroCuenta;
    // Tipo de cuenta como un valor enumerado
    enum tipo {AHORROS, CORRIENTE}
    // Atributo que define el tipo de cuenta bancaria
    tipo tipoCuenta;
    /* Atributo que define el saldo de la cuenta bancaria con valor inicial
       cero */
    float saldo = 0;

    /**
    * Constructor de la clase CuentaBancaria
    * @param nombresTitular Parámetro que define los nombres del
    * titular de una cuenta bancaria
    * @param apellidosTitular Parámetro que define los apellidos del
    * titular de una cuenta bancaria
    * @param numeroCuenta Parámetro que define el número de una
    * cuenta bancaria
    * @param tipoCuenta Parámetro que define el tipo de una cuenta
    * bancaria (puede ser ahorros o corriente)
    */
```

```
CuentaBancaria(String nombresTitular, String apellidosTitular, int
    numeroCuenta, tipo tipoCuenta) {
    /* Tener en cuenta que no se pasa como parámetro el saldo ya
        que inicialmente es cero. */
    this.nombresTitular = nombresTitular;
    this.apellidosTitular = apellidosTitular;
    this.númeroCuenta = numeroCuenta;
    this.tipoCuenta = tipoCuenta;
}

/**
* Método que imprime en pantalla los datos de una cuenta bancaria
*/
void imprimir() {
    System.out.println("Nombres del titular = " + nombresTitular);
    System.out.println("Apellidos del titular = " + apellidosTitular);
    System.out.println("Número de cuenta = " + númeroCuenta);
    System.out.println("Tipo de cuenta = " + tipoCuenta);
    System.out.println("Saldo = " + saldo);
}

/**
* Método que imprime en pantalla el saldo actual de una cuenta
* bancaria
*/
void consultarSaldo() {
    System.out.println("El saldo actual es = " + saldo);
}

/**
* Método que actualiza y devuelve el saldo de una cuenta bancaria a
* partir de un valor a consignar
* @param valor Parámetro que define el valor a consignar en la
* cuenta bancaria. El valor debe ser mayor que cero
* @return Valor booleano que indica si el valor a consignar es válido
* o no
*/
```

```
boolean consignar(int valor) {
    // El valor a consignar debe ser mayor que cero
    if (valor > 0) {
        saldo = saldo + valor; /* Se actualiza el saldo de la cuenta con
            el valor consignado */
        System.out.println("Se ha consignado $" + valor + " en la
            cuenta. El nuevo saldo es $" + saldo);
        return true;
    } else {
        System.out.println("El valor a consignar debe ser mayor que
            cero.");
        return false;
    }
}

/**
* Método que actualiza y devuelve el saldo de una cuenta bancaria a
* partir de un valor a retirar
* @param valor Parámetro que define el valor a retirar en la cuenta
* bancaria. El valor debe ser mayor que cero y el saldo de la cuenta
debe quedar con un valor positivo o igual a cero
* @return Valor booleano que indica si el valor a retirar es válido o no
*/
boolean retirar(int valor) {
    /* El valor debe ser mayor que cero y no debe superar el saldo
        actual */
    if ((valor > 0) && (valor <= saldo)) {
        saldo = saldo - valor; /* Se actualiza el saldo de la cuenta con
            el valor retirado */
        System.out.println("Se ha retirado $" + valor + " en la cuenta.
            El nuevo saldo es $" + saldo);
        return true;
    } else {
        System.out.println("El valor a retirar debe ser menor que el
            saldo actual.");
        return false;
    }
}
```

```
        /**
        * Método main que crea una cuenta bancaria sobre las cuales se
        * realizan las operaciones de consignar y retirar
        */
        public static void main(String args[]) {
            CuentaBancaria cuenta = new CuentaBancaria("Pedro","Pérez",
            123456789,tipo.AHORROS);
            cuenta.imprimir();
            cuenta.consignar(200000);
            cuenta.consignar(300000);
            cuenta.retirar(400000);
        }
    }
```

Diagrama de clases

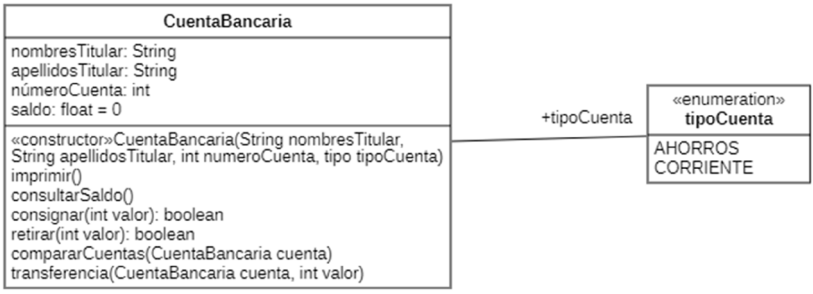


Figura 2.13. Diagrama de clases del ejercicio 2.5.

Explicación del diagrama de clases

Se ha definido una clase para modelar una cuenta bancaria. La clase cuenta con los atributos que representan los nombres del titular (de tipo *String*), los apellidos del titular (de tipo *String*), su número de cuenta (de tipo *int*) y el saldo de la cuenta con un valor inicial de cero (de tipo *float*). Se han definido los métodos de la cuenta bancaria: su constructor, un método para imprimir los datos en pantalla y consultar saldo (ambos no retornan ningún valor) y consignar y retirar que tienen un parámetro (el valor a consignar y retirar respectivamente). Los métodos consignar y retirar devuelven un valor booleano que determina si se pudo realizar la operación o no.

La clase CuentaBancaria tiene un atributo denominado tipoCuenta, que es un valor enumerado. En UML, el valor enumerado se puede representar como una clase con el estereotipo <<enumeration>> con sus valores constantes localizados como atributos en la clase.

Diagrama de objetos

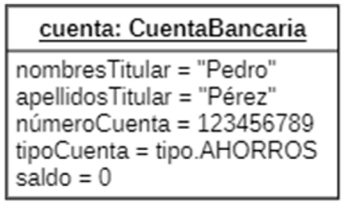


Figura 2.14. Diagrama de objetos del ejercicio 2.5.

Ejecución del programa

```
Nombres del titular = Pedro
Apellidos del titular = Pérez
Número de cuenta = 123456789
Tipo de cuenta = AHORROS
Saldo = 0.0
Se ha consignado $200000 en la cuenta. El nuevo saldo es $200000.0
Se ha consignado $300000 en la cuenta. El nuevo saldo es $500000.0
Se ha retirado $400000 en la cuenta. El nuevo saldo es $100000.0
```

Figura 2.15. Ejecución del programa del ejercicio 2.5.

Ejercicios propuestos

- Agregar a la clase CuentaBancaria, un atributo que represente el porcentaje de interés mensual aplicado a la cuenta.
- Agregar un método que calcule un nuevo saldo aplicando la tasa de interés correspondiente.

Ejercicio 2.6. Objetos como parámetros

Los métodos de una clase también pueden recibir un listado de objetos como parámetros. Es interesante que los cambios que se realicen a dichos

objetos se mantienen en los métodos que invocaron el método que utilizó el objeto como parámetro.

Por lo tanto, el paso de objetos como parámetros de un método es como el paso de parámetros por referencia; las variables mantienen una referencia al objeto. Mientras que, cuando se pasa un objeto como parámetro, se está realizando una copia de la referencia al objeto (Deitel y Deitel, 2017).

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Definir métodos que utilizan objetos como parámetros de entrada.
- Definir métodos que invocan a otros métodos definidos anteriormente.

Enunciado: clase CuentaBancaria

Se requiere modificar el programa de la cuenta bancaria (ejercicio 2.5) para que realice las siguientes actividades:

- Comparar saldos entre cuentas bancarias. La cuenta para comparar es un objeto que se envía como parámetro del método. El método devuelve un valor booleano de verdadero si la cuenta actual es mayor o igual a la cuenta que se pasó como parámetro.
- Transferir dinero de una cuenta bancaria a otra. El método debe recibir como parámetro la cuenta de destino y el valor a transferir. El saldo de la cuenta actual debe disminuir el valor a transferir y el saldo de la cuenta destino debe aumentar. El método debe reutilizar el método retirar para evaluar si la cantidad a transferir se encuentra en la cuenta de origen.

Solución

Clase: CuentaBancaria (continuación)

```
/**
* Método que compara los saldos de dos cuentas bancarias y
* muestra el resultado en pantalla
* @param cuenta Parámetro que define otra cuenta bancaria con la
* cual se va a comparar la cuenta bancaria actual
*/
void compararCuentas(CuentaBancaria cuenta) {
    /* Determina si el saldo de la cuenta actual es mayor o igual que
        el saldo de la otra cuenta */
    if (saldo >= cuenta.saldo) {
        System.out.println("El saldo de la cuenta actual es mayor o
            igual al saldo de la cuenta pasada como parámetro.");
    } else {
        System.out.println("El saldo de la cuenta actual es menor al
            saldo de la cuenta pasada como parámetro.");
    }
}

/**
* Método que transfiere un valor de una cuenta a otra
* @param cuenta Parámetro que define otra cuenta bancaria a la
* cual se le va a transferir un valor
* @param valor Parámetro que define el valor a transferir
*/
void transferencia(CuentaBancaria cuenta, int valor) {
    if (retirar(valor)) { // Si se puede retirar el valor de la cuenta actual
        cuenta.consignar(valor); /* Se realiza la consignación en la
            otra cuenta */
    }
}

/**
* Método main que crea dos cuentas bancarias sobre las cuales se
* realizan las operaciones de consignar, comparar, transferir y
* consultar saldos
*/
```

```
    public static void main(String args[]) {
        CuentaBancaria cuenta1 = new
            CuentaBancaria("Pedro","Pérez",123456789,tipo.
            AHORROS);
        CuentaBancaria cuenta2 = new
            CuentaBancaria("Pablo","Pinzón",44556677,tipo.
            AHORROS);
        cuenta1.consignar(200000);
        cuenta2.consignar(100000);
        cuenta1.compararCuentas(cuenta2);
        cuenta1.transferencia(cuenta2,50000);
        cuenta1.consultarSaldo();
        cuenta2.consultarSaldo();
    }
```

Diagrama de clases

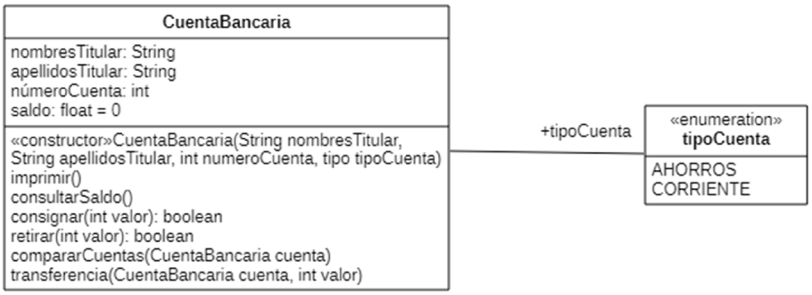


Figura 2.16. Diagrama de clases del ejercicio 2.6.

Explicación del diagrama de clases

Comparando este diagrama con el del ejercicio anterior, se han agregado dos nuevos métodos a la clase CuentaBancaria:

- Comparar cuentas, que tiene como parámetro un objeto de tipo CuentaBancaria.
- Transferencia, cuyos parámetros son un objeto de tipo CuentaBancaria y el valor a transferir a dicha cuenta.

Ambos métodos no tienen valor de retorno. La clase CuentaBancaria tiene un atributo denominado tipoCuenta, un valor enumerado. En UML, el

valor enumerado se puede representar como una clase con el estereotipo <<enumeration>> con sus valores constantes localizados como atributos en la clase.

Diagrama de objetos

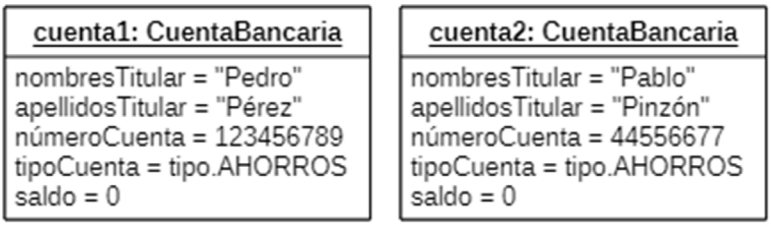


Figura 2.17. Diagrama de objetos del ejercicio 2.6.

Ejecución del programa

```
Se ha consignado $200000 en la cuenta. El nuevo saldo es $200000.0
Se ha consignado $100000 en la cuenta. El nuevo saldo es $100000.0
El saldo de la cuenta actual es mayor o igual al saldo de la cuenta pasada como parámetro.
Se ha retirado $50000 en la cuenta. El nuevo saldo es $150000.0
Se ha consignado $50000 en la cuenta. El nuevo saldo es $150000.0
El saldo actual es = 150000.0
El saldo actual es = 150000.0
```

Figura 2.18. Ejecución del programa del ejercicio 2.6.

Ejercicios propuestos

- Agregar un atributo a la clase CuentaBancaria, que determine si la cuenta está activa (de tipo *boolean*). Una cuenta está activa si tiene un saldo positivo. No se pueden realizar consignaciones a la cuenta si está inactiva. Si al retirar dinero, el saldo de la cuenta es cero, la cuenta pasa a considerarse inactiva.

Ejercicio 2.7. Métodos de acceso

Java posee modificadores de acceso para restringir el alcance de una clase, constructores, atributos, métodos y variables.

De acuerdo con Reyes y Stepp (2016), los modificadores de acceso disponibles en Java son:

- **Por defecto**: cuando no se especifica el modificador de acceso se dice que está predeterminado. Los elementos que no se declaran utilizando modificadores de acceso son accesibles solo dentro del mismo paquete.
- **Privado**: el modificador de acceso privado se especifica usando la palabra clave *private*. Los elementos declarados como privados son accesibles solo dentro de la clase en la que se declaran. Cualquier otra clase del mismo paquete no podrá acceder a estos elementos.
- **Protegido**: el modificador de acceso protegido se especifica mediante la palabra clave *protected*. Los elementos declarados como protegidos son accesibles dentro del mismo paquete o subclases en paquetes diferentes.
- **Público**: el modificador de acceso público se especifica mediante la palabra clave *public*. El modificador de acceso público tiene el alcance más amplio entre todos los demás modificadores de acceso. Los elementos que se declaran públicos son accesibles desde cualquier lugar del programa.

Objetivo de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para definir métodos de acceso para clases, atributos, constructores y métodos de un programa.

Enunciado: clase Película

Realizar un programa en Java que defina una clase Película con los siguientes atributos privados:

- Nombre: es el nombre de la película y es de tipo *String*.
- Director: representa el nombre del director de la película y es de tipo *String*.
- Género: es el género de la película, un tipo enumerado con los siguientes valores: ACCIÓN, COMEDIA, DRAMA y SUSPENSO.
- Duración: duración de la película en minutos, esta es de tipo *int*.
- Año: representa el año de realización de la película.
- Calificación: es la valoración de la película por parte de sus usuarios y es de tipo *double*.

Se debe definir un constructor público para la clase, que recibe como parámetros los valores de todos sus atributos. También se deben definir los siguientes métodos:

- Métodos *get* y *set* para cada atributo y con los derechos de acceso privados para los *set* y públicos para los *get*.
- Un método imprimir público que muestre en pantalla los valores de los atributos.
- Un método privado *boolean esPeliculaEpica()*, el cual devuelve un valor verdadero si la duración de la película es mayor o igual a tres horas, en caso contrario devuelve falso.
- Un método privado *String calcularValoración()*, el cual según la tabla 2.5 devuelve una valoración subjetiva.

Tabla 2.5. Valoración de las películas

Calificación	Valoración	Calificación	Valoración
[0, 2]	Muy mala	(7, 8]	Buena
(2, 5]	Mala	(8, 10]	Excelente
(5, 7]	Regular		

- El método privado *boolean esSimilar()* compara la película actual con otra película que se le pasa como parámetro. Si ambas películas son del mismo género y tienen la misma valoración, devuelve verdadero; en caso contrario, devuelve falso.
- Un método *main* que construya dos películas; determinar si son películas épicas; calcular su respectiva valoración y determinar si son similares. Las dos películas están en la tabla 2.6.
- Mostrar los resultados de la ejecución del método *main*.

Tabla 2.6. Objetos películas

Objeto	Nombre	Director	Género	Duración	Año	Calif.
película1	Gandhi	Richard Attenborough	DRAMA	191	1982	8.1
película2	Thor	Kenneth Branagh	ACCIÓN	115	2011	7.0

Solución

Clase: Película

```
/**
 * Esta clase define objetos que representan una Película. Una película
 * tiene un nombre, un director, un tipo, una duración, un año de
 * estreno y una calificación realizada por los usuarios.
 * @version 1.2/2020
 */
public class Película {
    // Atributo que define el nombre de la película
    private String nombre;
    // Atributo que define el director de la película
    private String director;
    // Tipo de película como un valor enumerado
    private enum tipo {ACCIÓN, COMEDIA, DRAMA, SUSPENSO};
    // Atributo que define el tipo de película
    tipo género;
    // Atributo que define la duración de la película
    private int duración;
    // Atributo que define el año de estreno de la película
    private int año;
    // Atributo que define la calificación de la película por el público
    private double calificación;

    /**
    * Constructor de la clase Película
    * @param nombre Parámetro que define el nombre o título de la
    * película
    * @param director Parámetro que define el nombre completo del
    * director de la película
    * @param género Parámetro que define el género de la película
    * @param duración Parámetro que define la duración de una
    * película (en minutos)
    * @param año Parámetro que define el año de estreno de la película
    * @param calificación Parámetro que define la calificación de la
    * pelicula realizada por el público
    */
```

```
public Película(String nombre, String director, tipo género, int dura-
    ción, int año, double calificación) {
    this.nombre = nombre;
    this.director = director;
    this.género = género;
    this.duración = duración;
    this.año = año;
    this.calificación = calificación;
}

/**
* Método que devuelve el nombre de una película
* @return El nombre de una película
*/
public String getNombre() {
    return nombre;
}

/**
* Método que establece el nombre de una película
* @param nombre Parámetro que define el nombre de una película
*/
private void setNombre(String nombre) {
    this.nombre = nombre;
}

/**
* Método que devuelve el director de una película
* @return El director de una película
*/
public String getDirector() {
    return director;
}

/**
* Método que establece el director de una película
* @param director Parámetro que define el director de una película
*/
private void setDirector(String director) {
    this.director = director;
}
```

```
/**
* Método que devuelve el género de una película
* @return El género de una película
*/
public tipo getGénero() {
    return género;
}

/**
* Método que establece el género de una película
* @param género Parámetro que define el género de una película
*/
private void setGénero(tipo género) {
    this.género = género;
}

/**
* Método que devuelve la duración de una película
* @return La duración de una película
*/
public int getDuración() {
    return duración;
}

/**
* Método que establece la duración de una película
* @param duración Parámetro que define la duración de una película
*/
private void setDuración(int duración) {
    this.duración = duración;
}

/**
* Método que devuelve el año de una película
* @return El año de estreno de una película
*/
public int getAño() {
    return año;
}

/**
* Método que establece el año de estreno de una película
* @param año Parámetro que define el año de una película
*/
```

```
    private void setAño(int año) {
        this.año = año;
    }

    /**
    * Método que devuelve la calificación de una película
    * @return La calificación de una película
    */
    public double getCalificación() {
        return año;
    }

    /**
    * Método que establece la calificación de una película
    * @param calificación Parámetro que define la calificación de una
        película
    */
    private void setCalificación(double calificación) {
        this.calificación = calificación;
    }

    /**
    * Método que imprime en pantalla los datos de una película
    */
    public void imprimir() {
        System.out.println("Nombre: " + nombre);
        System.out.println("Director: " + director);
        System.out.println("Género: " + género);
        System.out.println("Duración: " + duración);
        System.out.println("Año: " + año);
        System.out.println("Calificación: " + calificación);
    }

    /**
    * Método que determina si una película se puede considerar como épica
    * @return Valor booleano que determina si una película es épica o no
    */
    private boolean esPelículaEpica() {
        /* Una película se considera épica si tiene una duración igual o
            superior a 180 minutos */
        if (duración >= 180) {
            return true;
        } else {
```

```
            return false;
        }
    }

    /**
    * Método que determina la valoración cualitativa de una película
    * @return Valoración de una película
    */
    private String calcularValoración() {
        if (calificación >= 0 && calificación <= 2) { /* Entre [0, 2] se con-
            sidera "Muy mala" */
            return "Muy mala";
        } else if (calificación > 2 && calificación <= 5) { /* Entre (2, 5] se
            considera "Mala" */
            return "Mala";
        } else if (calificación > 5 && calificación <= 7) { /* Entre (5,7] se
            considera "Regular" */
            return "Regular";
        } else if (calificación > 7 && calificación <= 8) { /* Entre (7,8] se
            considera "Buena" */
            return "Buena";
        } else if (calificación > 8 && calificación <= 10){ /* Entre (8,10] se
            considera "Excelente" */
            return "Excelente";
        } else {
            return "No tiene asignada una valoración válida";
        }
    }

    /**
    * Método que determina si una película es similar a otra
    * @return Valor booleano que determina si una película es similar a
    * otra
    */
    private boolean esSimilar(Película película) {
        /* Dos películas son similares si ambas son del mismo género y si
            tienen la misma valoración */
        if (película.género == género && película.calcularValoración() ==
            calcularValoración()) {
            return true;
        } else {
            return false;
        }
```

```
    }

    /**
    * Método main que crea dos películas, imprime sus datos en
    * pantalla, determina si son épicas y si son similares
    */
    public static void main(String ags[]) {
        Película película1 = new Película("Gandhi", "Richard Attenborough",
            tipo.DRAMA,191,1982,8.3);
        Película película2 = new Película("Thor", "Kenneth Branagh",
            tipo.ACCIÓN, 115,2011,7.0);
        película1.imprimir();
        System.out.println();
        película2.imprimir();
        System.out.println();
        System.out.println("La película " + película1.getNombre() + " es
            épica: " + película1.esPelículaEpica());
        System.out.println("La película " + película2.getNombre() + " es
            épica: " + película2.esPelículaEpica());
        System.out.println("La película " + película1.getNombre() + " y
            la película " + película2.getNombre() + " son similares = " +
                película1.esSimilar(película2));
    }
}
```

Diagrama de clases

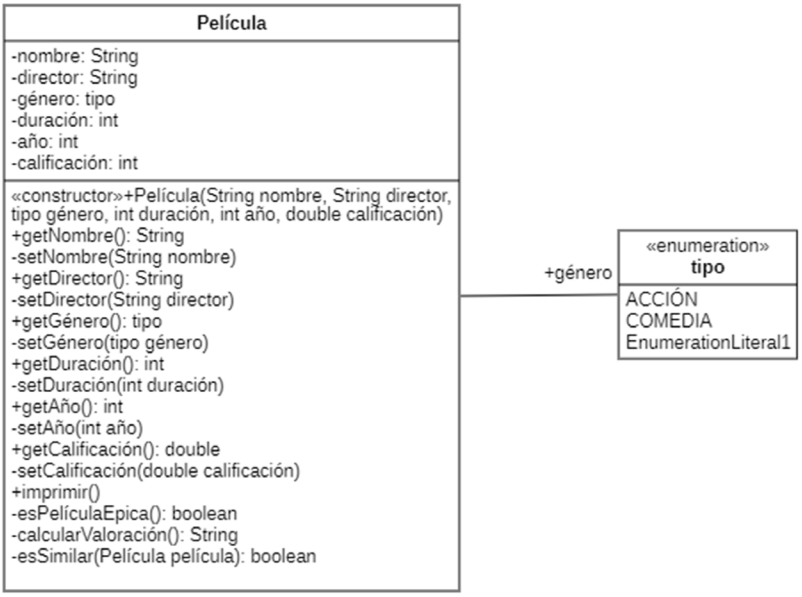


Figura 2.19. Diagrama de clases del ejercicio 2.7.

Explicación del diagrama de clases

Se ha definido una clase Película con los atributos nombre, director, género, duración, año y calificación, cada uno de ellos con su respectivo tipo de dato. Todos los atributos de la clase son privados, lo cual en UML se identifica con el símbolo (-), este símbolo antecede el nombre del atributo.

El constructor de la clase y los métodos *get* de cada atributo tienen acceso público representados con el símbolo (+), que antecede el nombre del método, en UML. Así mismo, métodos privados, con el símbolo (-) antes del nombre del método, los métodos *set* de cada atributo y los métodos para determinar si es una película épica, para calcular la valoración de la película y para determinar si una película es similar a otra.

Los métodos públicos podrán ser accedidos por objetos diferentes a Película, mientras que los métodos privados solo serán accedidos por objetos de la misma clase. Como los atributos son privados, la única forma de cambiar valores a sus atributos es por objetos de la misma clase y por medio del método *set*.

La consulta de los atributos por medio de los métodos *get* es pública y puede hacerse por objetos diferentes a Película.

Para representar el género de la película, el cual es un atributo enumerado, se ha definido una asociación denominada género con una clase tipo *enumeration*. Este tipo de clases se representan en UML con el estereotipo <<*enumeration*>> y con su lista de constantes como atributos en el segundo compartimiento de la clase.

Diagrama de objetos

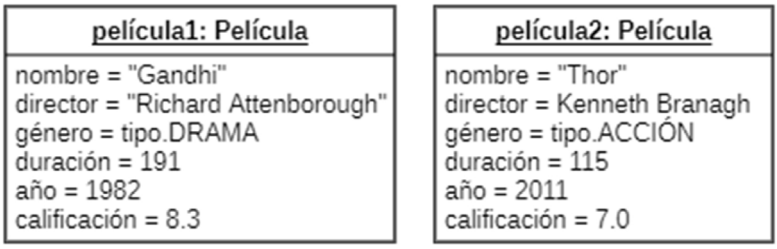


Figura 2.20. Diagrama de objetos del ejercicio 2.7.

Ejecución del programa

```
Nombre: Gandhi
Director: Richard Attenborough
Género: DRAMA
Duración: 191
Año: 1982
Calificación: 8.3

Nombre: Thor
Director: Kenneth Branagh
Género: ACCIÓN
Duración: 115
Año: 2011
Calificación: 7.0

La película Gandhi es épica: true
La película Thor es épica: false
La película Gandhi y la película Thor son similares = false
```

Figura 2.21. Ejecución del programa del ejercicio 2.7.

Ejercicios propuestos

Definir un método privado denominado *imprimirCartel* que muestra en pantalla los datos de la película en el siguiente formato:

-------- Título ----------

****

Año

Género1, Género2, Género3

Director

La valoración se debe convertir a una cantidad determinada de asteriscos (*).

Una película puede tener hasta tres géneros. Los géneros se muestran separados por comas.

Ejercicio 2.8. Asignación de objetos

En Java, cuando se asigna un objeto a otro de la misma clase, ambos están referenciando al mismo objeto (Schildt, 2018). En una sentencia de asignación como:

```
Object o1, o2;
o1 = new Object();
o2 = o1;
```

Ambos objetos o1 y o2 están referenciando el mismo objeto, no se crea un objeto duplicado, ni se asigna memoria extra. Los cambios que se realicen a un objeto se actualizan de inmediato en el otro.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Determinar el contenido de un objeto que ha pasado por varias instrucciones de asignación y se ha cambiado su contenido o referencia a otros objetos.
- Cambiar el estado de un objeto utilizando métodos *set*.

Enunciado: clase Avión

Se tiene un código que modela una clase Avión que posee dos atributos: nombre del fabricante del avión (tipo *String*) y número de motores del avión (tipo *int*). La clase tiene un constructor y métodos *get* y *set* para cada atributo. Además, tiene los siguientes métodos adicionales:

- Un método denominado imprimirFabricante(), que muestra en pantalla el nombre del fabricante de un avión.
- Un método denominado cambiarFabricante(Avión a), que recibe como parámetro un objeto de tipo Avión y le cambia el valor de su atributo fabricante a un valor predefinido "Loockhead".

En el método *main* se crean dos aviones: a1 (fabricante "Airbus" con cuatro motores) y a2 (fabricante "Lookheed" con cuatro motores). Luego, los datos de cada avión se imprimen por pantalla. Después, se realizan llamadas a los métodos setFabricante y cambiarFabricante, los cuales cambiarán los valores de sus atributos. ¿Cuál es el resultado final de la ejecución del método *main*? Determinar lo que se imprime en pantalla.

Solución

Clase: Avión

```
/**
 * Esta clase define objetos de tipo Avión con un fabricante y número de
 * motores como atributos.
 * @version 1.2/2020
 */
public class Avión {
    private String fabricante; /* Atributo que define el nombre del fabri-
        cante del avión */
    private int númeroMotores; /* Atributo que define el número de mo-
        tores del avión */

    /**
    * Constructor de la clase Avión
    * @param fabricante Parámetro que define el nombre del fabricante
    * de un avión
    * @param númeroMotores Parámetro que define el número de
    * motores que tiene un avión
```

```
*/
private Avión(String fabricante, int númeroMotores) {
    this.fabricante = fabricante;
    this.númeroMotores = númeroMotores;
}

/**
* Método que devuelve el nombre del fabricante de un avión
* @return El nombre del fabricante de un avión
*/
public String getFabricante() {
    return fabricante;
}

/**
* Método que establece el nombre de un fabricande de un avión
* @param fabricante Parámetro que define el nombre del fabricante
* de un avión
*/
private void setFabricante(String fabricante) {
    this.fabricante = fabricante;
}

/**
* Método que cambia el fabricante de un avión pasado como
* parámetro por el valor "Lockheed"
* @param a Parámetro que define un avión
*/
private void cambiarFabricante(Avión a) {
    a.setFabricante("Lockheed");
}

/**
* Método que devuelve el número de motores de un avión
* @return El número de motores de un avión
*/
public int getNúmeroMotores() {
    return númeroMotores;
}
```

```
    /**
    * Método que establece el número de motores de un avión
    * @param númeroMotores Parámetro que define el número de
    * motores de un avión
    */
    private void setNúmeroMotores(int númeroMotores) {
        this.númeroMotores = númeroMotores;
    }

    /**
    * Método que imprime en pantalla el fabricante de un avión
    */
    public void imprimirFabricante() {
        System.out.println("El fabricante del avión es: " + fabricante);
    }

    /**
    * Método main que crea dos aviones y modifica sus fabricantes
    */
    public static void main(String args[]) {
        Avión a1 = new Avión("Airbus",4);
        Avión a2;
        Avión a3 = new Avión("Boeing",4);
        a2 = a1;
        a1.imprimirFabricante();
        a2.imprimirFabricante();
        a1.setFabricante("Douglas");
        a1.imprimirFabricante();
        a2.imprimirFabricante();
        a1.cambiarFabricante(a2);
        a2.imprimirFabricante();
    }
}
```

Diagrama de clases

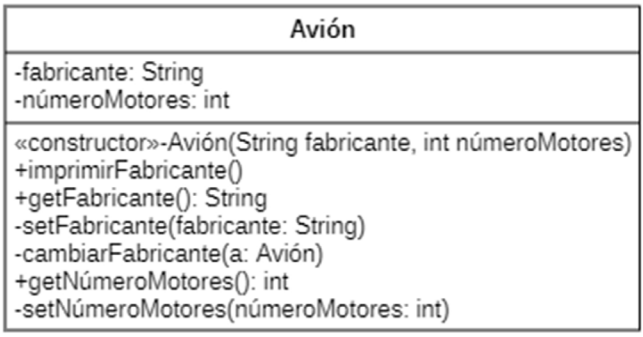


Figura 2.22. Diagrama de clases del ejercicio 2.8.

Explicación del diagrama de clases

Se ha definido una clase Avión con dos atributos que representan el fabricante del avión (de tipo *String*) y el número de motores que tiene (de tipo *int*). Ambos atributos son privados, tienen el símbolo (-) precediendo el nombre del atributo.

La clase Avión tiene un constructor que inicializa los atributos del avión. Además, tiene los métodos *get* y *set* de cada atributo. Los métodos *get* son públicos y los *set*, privados. El método cambiarFabricante es privado y recibe como parámetro un objeto de tipo Avión.

Diagrama de objetos

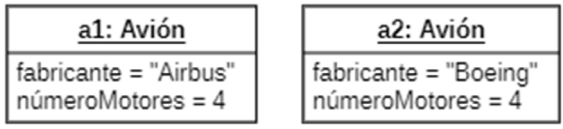


Figura 2.23. Diagrama de objetos del ejercicio 2.8.

Ejecución del programa

```
El fabricante del avión es: Airbus
El fabricante del avión es: Airbus
El fabricante del avión es: Douglas
El fabricante del avión es: Douglas
El fabricante del avión es: Lockheed
```

Figura 2.24. Ejecución del programa del ejercicio 2.8.

El primer avión a1 tiene como fabricante "Airbus", por ello imprime en pantalla dicho valor en la primera línea. Al segundo avión a2 se le asignó el contenido de a1, por ello imprime también "Airbus" en la segunda línea. Luego, se cambia el fabricante de a1, pasa de "Airbus" a "Douglas", por ello imprime "Douglas" en la tercera línea. Al cambiar el contenido de a1, también cambia en forma automática el contenido de a2, por ello imprime "Douglas" en la cuarta línea. Por último, al invocar el método cambiarFabricante se cambia el valor del objeto pasado como parámetro a2 y se imprime "Lockheed" en la quinta línea.

Ejercicios propuestos

- En un método *main* se deben crear dos objetos Avión y asignar el primer objeto al segundo. Luego, mostrar en pantalla los valores de los atributos de los dos objetos. Después, asignar el valor "*Stealth*" al atributo fabricante del segundo objeto. Por último, es necesario imprimir en pantalla el valor del atributo fabricante del primer objeto ¿Qué se mostrará en pantalla?

Ejercicio 2.9. Variables locales dentro de un método

En Java, las variables locales se declaran dentro de un método. Su visibilidad está solo dentro del método donde se define. Ningún código fuera de un método puede ver las variables locales dentro de otro método. No es necesario declarar una variable local con un modificador de visibilidad (Deitel y Deitel, 2017).

La vida útil de una variable local cubre desde su declaración hasta el retorno del método. Las variables locales se crean en la pila de llamadas cuando se ingresa al método y se destruyen cuando se sale del método.

Las constantes en Java son variables cuyos valores no pueden cambiar, una vez hayan sido definidas. Las constantes se declaran por medio de instrucciones con el siguiente formato:

static final tipoDato nombreConstante = valor;

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Definir variables locales en un método.
- Definir constantes en una clase.

Enunciado: clase ConversorMetros

Realizar un programa en Java que permita realizar las siguientes conversiones de unidades de longitud:

- Metros a centímetros.
- Metros a milímetros.
- Metros a pulgadas.
- Metros a pies.
- Metros a yardas.

Instrucciones Java del ejercicio

Tabla 2.7. Instrucciones Java del ejercicio 2.9.

Instrucción	Descripción	Formato
final	Palabra clave para definir que una vez que a una variable se le ha asignado un valor, siempre contendrá el mismo valor (permanece constante). Se recomienda que los nombres de las constantes se escriban en mayúscula.	*final tipo* VARIABLE = valor;

Solución

Clase: ConversorMetros

```
/**
 * Esta clase define objetos de tipo ConversorMetros el cual permite
 * realizar conversiones entre diferentes unidades de medición de longitud.
 * @version 1.2/2020
 */
public class ConversorMetros {
    /* Atributo que define la cantidad de metros a convertir a diferentes
       unidades de longitud */
    double metros;
    final int FACTOR_METROS_CM = 100; /* Factor de conversión de
       metros a centímetros */
    final int FACTOR_METROS_MILIM = 1000; /* Factor de conversión
       de metros a milímetros */
    final double FACTOR_METROS_PULGADAS = 39.37; /* Factor de
       conversión de metros a pulgadas */
    final double FACTOR_METROS_PIES = 3.28; /* Factor de
       conversión de metros a pies */
    final double FACTOR_METROS_YARDAS = 1.09361; /* Factor de
       conversión de metros a yardas */

    /**
    * Constructor de la clase ConversorMetros
    * @param metros Parámetro que define la cantidad de metros a
    * convertir a otras unidades de longitud
    */
    public ConversorMetros(double metros) {
        this.metros = metros;
    }

    /**
    * Método que convierte metros a centímetros
    * @return Resultado de la conversión de metros a centímetros
    */
    public double convertirMetrosToCentímetros() {
        double centímetros;
        centímetros = FACTOR_METROS_CM * metros;
        return centímetros;
    }
```

```
/**
* Método que convierte metros a milímetros
* @return Resultado de la conversión de metros a milimetros
*/
public double convertirMetrosToMilímetros() {
    double milímetros;
    milímetros = FACTOR_METROS_MILIM * metros;
    return milímetros;
}

/**
* Método que convierte metros a pulgadas
* @return Resultado de la conversión de metros a pulgadas
*/
public double convertirMetrosToPulgadas() {
    double pulgadas;
    pulgadas = FACTOR_METROS_PULGADAS * metros;
    return pulgadas;
}

/**
* Método que convierte metros a pies
* @return Resultado de la conversión de metros a pies
*/
public double convertirMetrosToPies() {
    double pies;
    pies = FACTOR_METROS_PIES * metros;
    return pies;
}

/**
* Método que convierte metros a yardas
* @return Resultado de la conversión de metros a yardas
*/
public double convertirMetrosToYardas() {
    double yardas;
    yardas = FACTOR_METROS_YARDAS * metros;
    return yardas;
}
```

```
    /**
    * Método main que define una cierta cantidad de metros y los
    * convierte a diferentes unidades de longitud
    */
    public static void main (String args[]) {
        ConversorMetros conversor = new ConversorMetros(3.5);
        System.out.println("Metros = " + conversor.metros);
        System.out.println("Metros a centímetros = "  + conversor.convertirMetrosToCentímetros());
        System.out.println("Metros a milímetros = "  + conversor.convertirMetrosToMilímetros());
        System.out.println("Metros a pulgadas = "  + conversor.convertirMetrosToPulgadas());
        System.out.println("Metros a pies = "  + conversor.convertirMetrosToPies());
        System.out.println("Metros a yardas = "  + conversor.convertirMetrosToYardas());
    }
}
```

Diagrama de clases

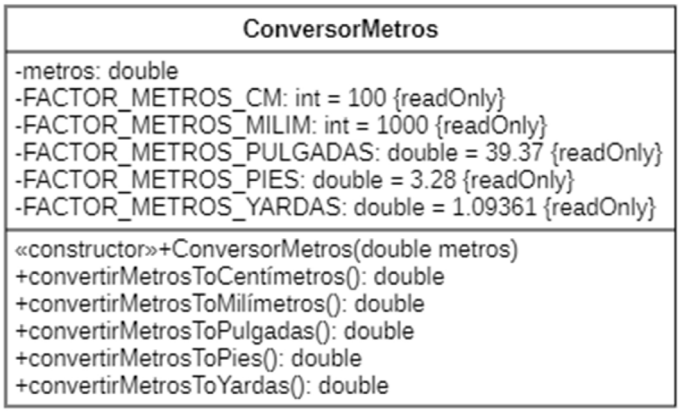


Figura 2.25. Diagrama de clases del ejercicio 2.9.

Explicación del diagrama de clases

Se ha definido una clase denominada ConversorMetros. Tiene un atributo denominado metros (de tipo *double*) para definir la cantidad de metros a

convertir a diferentes unidades de longitud. También se han definido como atributos un conjunto de constantes que se identifican con la etiqueta UML *{readOnly}*. Se recomienda que las constantes se escriban en mayúsculas y si el identificador es una palabra compuesta, cada palabra se separa con el símbolo: _.

La clase tiene un constructor para inicializar la cantidad de metros a convertir y un conjunto de cinco métodos públicos (identificados con el símbolo +) para realizar diferentes conversiones, cada uno con su valor de retorno correspondiente (de tipo *double*). Los métodos permiten convertir un valor en metros a centímetros, milímetros, pulgadas, pies y yardas.

Diagrama de objetos

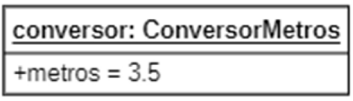


Figura 2.26. Diagrama de objetos del ejercicio 2.9.

Ejecución del programa

```
Metros = 3.5
Metros a centímetros = 350.0
Metros a milímetros = 3500.0
Metros a pulgadas = 137.795
Metros a pies = 11.479999999999999
Metros a yardas = 3.827635
```

Figura 2.27. Ejecución del programa del ejercicio 2.9.

Ejercicios propuestos

- Hacer clases similares para realizar conversiones de unidades de medición como:
  - Medidas de superficie o área: convertir áreas (1 área= 100 m^2) a: hectáreas (1 hectárea= 10 000 m^2); kilómetros cuadrados (1 kilómetro cuadrado= 1 000 000 m^2); fanegas (1 fanega = 6460 m^2) y acres (1 acre= 4046.85 m^2).
  - Medidas de volumen: convertir litros a: galones (1 galón=4,41 litros); pintas (1 pinta= 0.46 litros); barriles (1 barril= 158.99

litros), metros cúbicos (1 m^3= 1000 litros) y hectolitros (1 hectolitro= 100 litros).

Ejercicio 2.10. Sobrecarga de métodos

La sobrecarga de métodos se refiere a que una clase puede poseer múltiples métodos con el mismo nombre (Samoylov, 2019). Es requisito que cada método sobrecargado tenga diferente signatura. Por lo tanto, los métodos sobrecargados realizan la "misma acción", pero teniendo en cuenta:

- Diferentes cantidades de parámetros.

void método(int paramétro1, int paramétro2, int paramétro3)
void método(int paramétro1, int paramétro2)

- Diferentes tipos de parámetros.

void método(int paramétro1, int paramétro2, int paramétro3)
void método(double paramétro1, double paramétro2, double paramétro3)

- Diferente orden de parámetros.

void método(int paramétro1, double paramétro2, float paramétro3)
void método(float paramétro1, double paramétro2, int paramétro3)

- Diferente valor de retorno.

void método(int paramétro1, double paramétro2, float paramétro3)
String método(int paramétro1, double paramétro2, float paramétro3)

Objetivo de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para definir métodos sobrecargados en una clase.

Enunciado: clase Pedido

Realizar un programa en Java que permita calcular el pedido que realiza un cliente en un restaurante.

Los pedidos de un restaurante están conformados por las siguientes partes:

- Un primer plato.
- Un segundo plato.
- Una bebida.
- Un postre.

Cada uno de dichas partes tiene un nombre y un valor. Se requiere definir métodos sobrecargados para calcular el valor del pedido dependiendo si el cliente solicita:

- Un primer plato y una bebida.
- Un primer plato, un segundo plato y una bebida.
- Un primer plato, un segundo plato, una bebida y un postre.

Implementar un método *main* que utilice los tres métodos sobrecargados en tres diferentes pedidos.

Solución

Clase: Pedido

```
/**
 * Esta clase define objetos de tipo Pedido de un restaurante que consta
 * de diferentes platos y que tiene un determinado valor.
 * @version 1.2/2020
 */
public class Pedido {

    /**
     * Método que tiene como parámetros los elementos que conforman
     * un pedido con un primer plato, su bebida y sus costos
     * correspondientes. El método calcula el costo total del pedido y
     * muestra en pantalla los datos del pedido y su costo total.
     * @param primerPlato Parámetro que define el nombre del primer
     * plato que conforma el pedido
     * @param costoPrimerPlato Parámetro que define el costo del
     * primer plato que conforma el pedido
     * @param bebida Parámetro que define el nombre de la bebida que
     * conforma el pedido
```

```
* @param costoBebida Parámetro que define el costo de la bebida
* que conforma el pedido
*/
public void calcularPedido(String primerPlato, double costoPrimer
    Plato, String bebida, double costoBebida) {
    double total = costoPrimerPlato + costoBebida;
    System.out.println("El costo de " + primerPlato + " y " + bebida +
        " es = $" + total);
}

/**
* Método sobrecargado que tiene como parámetros los elementos
* que conforman un pedido con un primer plato, un segundo plato,
* su bebida y sus costos correspondientes. El método calcula el costo
* total del pedido y muestra en pantalla los datos del pedido y su
* costo total.
* @param primerPlato Parámetro que define el nombre del primer
* plato que conforma el pedido
* @param costoPrimerPlato Parámetro que define el costo del
* primer plato que conforma el pedido
* @param segundoPlato Parámetro que define el nombre del
* segundo plato que conforma el pedido
* @param costoSegundoPlato Parámetro que define el costo del
* segundo plato que conforma el pedido
* @param bebida Parámetro que define el nombre de la bebida que
* conforma el pedido
* @param costoBebida Parámetro que define el costo de la bebida
* que conforma el pedido
*/
public void calcularPedido(String primerPlato, double
    costoPrimerPlato, String segundoPlato, double
    costoSegundoPlato, String bebida, double costoBebida) {
    double total = costoPrimerPlato + costoSegundoPlato +
        costoBebida;
    System.out.println("El costo de " + primerPlato + " + " +
        segundoPlato + " + " + bebida + " es = $" + total);
}
```

```
public void calcularPedido(String primerPlato, double
    costoPrimerPlato, String segundoPlato, double
    costoSegundoPlato, String bebida, double costoBebida) {
    double total = costoPrimerPlato + costoSegundoPlato +
        costoBebida;
    System.out.println("El costo de " + primerPlato + " + " +
        segundoPlato + " + " + bebida + " es = $" + total);
}

/**
* Método sobrecargado que tiene como parámetros los elementos
* que conforman un pedido con un primer plato, segundo plato, su
* bebida, postre y sus costos correspondientes. El método calcula el
* costo total del pedido y muestra en pantalla los datos del pedido y
* su costo total.
* @param primerPlato Parámetro que define el nombre del primer
* plato que conforma el pedido
* @param costoPrimerPlato Parámetro que define el costo del
* primer plato que conforma el pedido
* @param segundoPlato Parámetro que define el nombre del
* segundo plato que conforma el pedido
* @param costoSegundoPlato Parámetro que define el costo del
* segundo plato que conforma el pedido
* @param postre Parámetro que define el nombre del postre que
* conforma el pedido
* @param costoPostre Parámetro que define el costo del postre que
* conforma el pedido
* @param bebida Parámetro que define el nombre de la bebida que
* conforma el pedido
* @param costoBebida Parámetro que define el costo de la bebida
* que conforma el pedido
*/
public void calcularPedido(String primerPlato, double
    costoPrimerPlato, String segundoPlato, double
    costoSegundoPlato, String postre, double costoPostre, String
    bebida, double costoBebida) {
    double total = costoPrimerPlato + costoSegundoPlato +
        costoBebida + costoPostre;
```

```
            System.out.println("El costo de " + primerPlato + " + " +
                segundoPlato + " + " + bebida + " + " +
                    postre + " es = $" + total);
        }
        /**
        * Método main que crea tres diferentes tipos de pedidos en el
        * restaurante.
        */
        public static void main (String args[]) {
            Pedido pedido1 = new Pedido();
            pedido1.calcularPedido("Sancocho", 5000, "Gaseosa", 2000);
            Pedido pedido2 = new Pedido();
            pedido2.calcularPedido("Crema de verduras", 5000,
                "Churrasco", 6000, "Gaseosa", 2000);
            Pedido pedido3 = new Pedido();
            pedido3.calcularPedido("Crema de espinacas",
                5000,"Salmón",10000,"Tiramisú",5000,"Gaseosa",2000);
        }
    }
```

Diagrama de clases

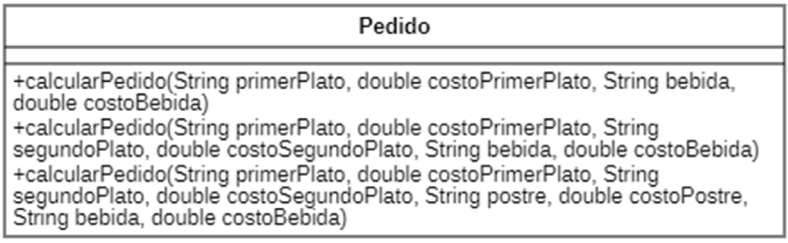


Figura 2.28. Diagrama de clases del ejercicio 2.10.

Explicación del diagrama de clases

Se ha definido una clase denominada Pedido, la cual tiene tres métodos públicos (identificados con el símbolo +) sobrecargados. Al estar sobrecargados, los métodos tienen el mismo nombre, pero diferente signatura. En este caso, los métodos tienen diferente cantidad de parámetros. La clase no tiene atributos.

Diagrama de objetos

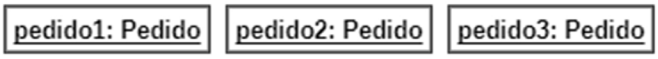


Figura 2.29. Diagrama de objetos del ejercicio 2.10.

Ejecución del programa

```
El costo de Sancocho y Gaseosa es = $7000.0
El costo de Crema de verduras + Churrasco + Gaseosa es = $13000.0
El costo de Crema de espinacas + Salmón + Gaseosa + Tiramisú es = $22000.0
```

Figura 2.30. Ejecución del programa del ejercicio 2.10.

Ejercicios propuestos

- Definir una clase denominada Suma, la cual tiene varios métodos sumar sobrecargados:
  - Un método sumar que obtiene la suma de dos valores enteros pasados como parámetros.
  - Un método sumar que obtiene la suma de tres valores enteros pasados como parámetros.
  - Un método sumar que obtiene la suma de dos valores *double* pasados como parámetros.
  - Un método sumar que obtiene la suma de tres valores *double* pasados como parámetros.

Ejercicio 2.11. Sobrecarga de constructores

A veces es necesario inicializar un objeto de diferentes formas. Para ello, se realiza la sobrecarga de constructores. Los constructores sobrecargados deben tener diferente número o tipo de parámetros. Cada constructor realiza una tarea diferente (Liang, 2017).

Se puede utilizar la palabra reservada *this* para llamar a un constructor desde otro. La llamada *this* debe ser la primera línea de dicho constructor.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Definir constructores sobrecargados.
- Definir constructores que llamen a otros constructores.

Enunciado: clase ArtículoCientífico

Realizar un programa en Java que permita modelar un artículo científico. Los artículos científicos contienen los siguientes metadatos: nombre del artículo, autor, palabras claves, nombre de la publicación, año y resumen.

Se deben definir tres constructores sobrecargados:

- El primero inicializa un artículo científico con solo su título y autor.
- El segundo constructor, un artículo científico con su nombre, autor, palabras claves, nombre de la publicación y año. Debe invocar al primer constructor.
- El tercer constructor, un artículo científico con su nombre, autor, palabras claves, nombre de la publicación, año y resumen. Debe invocar al segundo constructor.

Se requiere un método que imprima los atributos de un artículo en pantalla. Realizar un método *main* que utilice el tercer constructor para instanciar un artículo científico e imprima los valores de sus atributos en pantalla.

Instrucciones Java del ejercicio

Tabla 2.8. Instrucciones Java del ejercicio 2.11.

Instrucción	Descripción	Formato
length	Variable final que permite obtener el tamaño del *array*. No confundir con *length*() con paréntesis, que se aplica a objetos *String*.	nombreArray.length
++	Operador de incremento utilizado para incrementar el valor en 1. Hay dos tipos: preincremento (el valor se incrementa primero y luego se calcula el resultado) y posincremento (el valor se usa por primera vez para calcular el resultado y luego se incrementa).	Preincremento: ++variable Posincremento: variable++

Solución

Clase: ArtículoCientífico

```
/**
 * Esta clase define objetos de tipo ArtículoCientífico con un título,
 * autor, tres palabras clave, año de publicación y un resumen.
 * @version 1.2/2020
 */
public class ArtículoCientífico {
    String título; // Atributo que define el título de un artículo científico
    String autor; // Atributo que define el autor de un artículo científico
    // Atributo que define las palabras clave de un artículo científico
    String[] palabrasClaves = new String[3];
    String publicación; /* Atributo que define la publicación que incluye
        el artículo científico */
    int año; /* Atributo que define el año de publicación de un artículo
        científico */
    String resumen; /* Atributo que define el resumen de un artículo
        científico */

    /**
    * Constructor de la clase ArtículoCientífico
    * @param título Parámetro que define el título del artículo científico
    * @param autor Parámetro que define el autor del artículo científico
    */
    public ArtículoCientífico(String título, String autor) {
        this.título = título;
        this.autor = autor;
    }

    /**
    * Constructor sobrecargado de la clase ArtículoCientífico
    * @param título Parámetro que define el título del artículo científico
    * @param autor Parámetro que define el autor del artículo científico
    * @param palabrasClaves Parámetro que define un listado de
    * palabras clave para el artículo científico
    * @param publicación Parámetro que define el nombre de la
    * publicación a la que pertenece el artículo científico
    * @param año Parámetro que define el año de publicación del
    * artículo científico
    */
```

```
public ArtículoCientífico(String título, String autor, String[] palabras-
    Claves, String publicación, int año) {
    this(título, autor); // Invoca al constructor sobrecargado
    this.palabrasClaves = palabrasClaves;
    this.publicación = publicación;
    this.año = año;
}

/**
* Constructor sobrecargado de la clase ArtículoCientífico
* @param título Parámetro que define el título del artículo científico
* @param autor Parámetro que define el autor del artículo científico
* @param palabrasClaves Parámetro que define un listado de
* palabras clave para el artículo científico
* @param publicación Parámetro que define el nombre de la
* publicación a la que pertenece el artículo científico
* @param año Parámetro que define el año de publicación del
* artículo científico
* @param resumen Parámetro que define el resumen del artículo
* científico
*/
public ArtículoCientífico(String título, String autor, String[] palabras-
    Claves, String publicación, int año, String resumen) {
    this(título, autor, palabrasClaves, publicación, año); /* Invoca al
        constructor sobrecargado */
    this.resumen = resumen;
}

/**
* Método que imprime en pantalla los datos de un artículo científico
*/
public void imprimir() {
    System.out.println("Título del artículo = " + título);
    System.out.println("Autor del artículo = " + autor);
    System.out.println("Palabras clave = ");
    // Recorre el array para imprimir cada palabra clave
    for (int i = 0; i < palabrasClaves.length; i++) {
        System.out.println(palabrasClaves[i]);
    }
    System.out.println("Publicación = " + publicación);
    System.out.println("Año = " + año);
    System.out.println("Resumen = " + resumen);
}
```

```
    /**
     * Método main que instancia un artículo científico y muestra sus
     * datos en pantalla
     */
    public static void main (String args[]) {
        String[] palabras = {"Física","Espacio","Tiempo"};
        ArtículoCientífico artículo = new ArtículoCientífico("La teoría es-
            pecial de la relatividad", "Albert Einstein",palabras, "Anales de
            Física", 1913, "Las leyes de la física son las mismas en todos
            los sistemas de referencia inerciales.");
        artículo.imprimir();
    }
}
```

Diagrama de clases

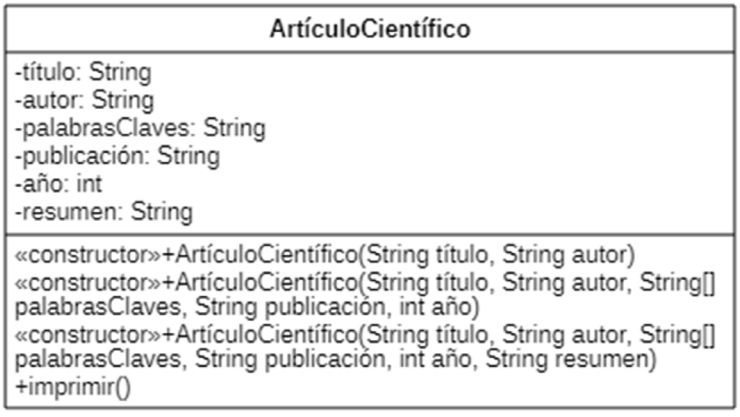


Figura 2.31. Diagrama de clases del ejercicio 2.11.

Explicación del diagrama de clases

Se ha definido una clase denominada ArtículoCientífico, la cual tiene los atributos privados (identificados con el símbolo -): título, autor, palabras clave, publicación, año y resumen. La clase tiene tres constructores públicos sobrecargados (con el mismo nombre y diferente signatura) y un método imprimir. Todos los constructores y el método imprimir son públicos (identificados con el símbolo +).

Diagrama de objetos

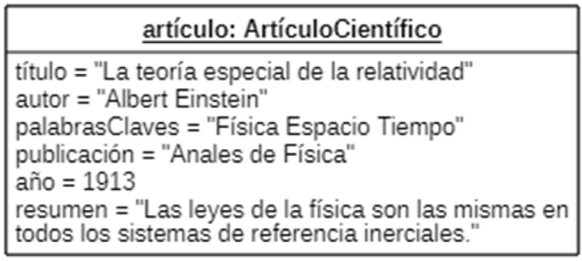


Figura 2.32. Diagrama de objetos del ejercicio 2.11.

Ejecución del programa

```
Título del artículo = La teoría especial de la relatividad
Autor del artículo = Albert Einstein
Palabras clave =
Física
Espacio
Tiempo
Publicación = Anales de Física
Año = 1913
Resumen = Las leyes de la física son las mismas en todos los sistemas de referencia inerciales.
```

Figura 2.33. Ejecución del programa del ejercicio 2.11.

Ejercicios propuestos

- Definir una clase Empleado que tiene como atributos: identificador, nombre, apellidos y edad del empleado. La clase contiene dos constructores:
  - El primer constructor no tiene parámetros e inicializa los atributos del objeto con los siguientes valores: identificador del empleado con el valor 100, el nombre con "Nuevo empleado", apellidos con "Nuevo empleado" y edad del empleado con 18.
  - El segundo constructor asigna valores a los atributos de acuerdo con los valores pasados como parámetros.
- Definir una clase Caja que tiene como atributos la longitud de su base, anchura y altura. La clase contiene tres constructores:
  - El primer constructor asigna valores a los atributos de acuerdo con los valores pasados como parámetros.

- El segundo constructor inicializa todos los atributos de una caja con valores de cero.
- El tercer constructor recibe un parámetro de longitud y les asigna dicho valor a todos sus atributos.
- Definir un nuevo atributo que represente el tipo de caja y un nuevo constructor que reciba como parámetros los valores de los cuatro atributos. Este constructor debe invocar al primero.

Capítulo 3
String, *wrappers* y estructuras de almacenamiento

El propósito general de este capítulo es comprender y poner en práctica algunos recursos que posee Java para realizar operaciones con los tipos primitivos de datos y para gestionar colecciones de elementos de un mismo tipo de dato. En este tercer capítulo se presentan siete ejercicios sobre diversos conceptos como elementos estáticos, envoltorios (*wrappers*), entrada de datos por teclado y estructuras de almacenamiento (*arrays* y vectores).

- **Ejercicio 3.1. Atributos y métodos estáticos**
  En la programación orientada a objetos existen atributos y métodos que no son parte de un objeto específico, sino que son compartidos por todos los objetos de una misma clase, son los atributos y métodos estáticos. El primer ejercicio plantea un problema para entender este concepto.

- **Ejercicio 3.2. Clase *String***
  Los atributos y variables de tipo *String* son ampliamente utilizados en el desarrollo de programas. Los datos tipo *String* no son un tipo primitivo de dato, sino una clase con una colección de métodos bastante interesante que permiten su gestión y manipulación. El segundo ejercicio está orientado a conocer y aplicar los métodos de la clase *String*.

- **Ejercicio 3.3. *Wrappers***
  Los tipos primitivos de datos no cuentan con atributos y métodos que permitan realizar operaciones sobre ellos. Los envoltorios o *wrappers* permiten llevar a cabo estas acciones sobre los tipos primitivos de datos. El tercer ejercicio plantea un problema para conocer los diferentes wrappers que permiten convertir datos *String* a otros tipos de datos.

- **Ejercicio 3.4. Entrada de datos desde teclado**
  El mecanismo más utilizado para introducir datos en un programa es el teclado. En Java, la introducción de datos desde el teclado es gestionada también por una clase. El cuarto ejercicio hace énfasis en la utilización de la clase *Scanner* para capturar datos provenientes de la introducción de información desde el teclado.

- **Ejercicio 3.5. Clases internas**
  Una clase puede contener en su definición otra clase, la cual incluye su código completo. A esta clase se le denomina clase interna. El quinto ejercicio propone el uso de una clase interna para conocer sus características, particularidades y su tratamiento en relación con la clase donde está contenida.

- **Ejercicio 3.6. *Arrays* de objetos**
  El concepto de *array*, el cual fue tratado en el apartado 1.5, también puede ser aplicado para definir *arrays* de objetos. El sexto ejercicio presenta un problema que aplica la definición de *arrays* con elementos de un tipo particular de objeto.

- **Ejercicio 3.7. Vectores de objetos**
  En Java existen diferentes estructuras, además de los *arrays*, que permiten gestionar colecciones de objetos. Una de ellas es el vector, el cual también es un objeto que contiene atributos y métodos que facilitan al programador la consulta y manipulación de sus elementos. El último ejercicio de este capítulo pretende resolver el problema del ejercicio anterior, pero en lugar de utilizar *arrays* se deben utilizar vectores.

Los diagramas UML utilizados en este capítulo son los diagramas de clase y diagramas de objetos. Los diagramas de clase modelan la estructura estática de los programas, mientras que los diagramas de clase de los ejercicios presentados, generalmente, estarán conformados por una o varias clases e incorporan notaciones para identificar tipos de atributos y métodos. Los diagramas de objeto identifican los objetos creados en el método *main* del programa.

Ejercicio 3.1. Atributos y métodos estáticos

Los atributos y métodos estáticos están asociados a la clase en que están definidos. Los atributos estáticos son compartidos por todas las instancias de la misma clase (Flanagan y Evans, 2019).

Un atributo estático se define utilizando la palabra clave *static* en la definición del atributo:

static tipoAtributo atributo;

Un método estático se define también utilizando la palabra clave *static* en la definición del método:

static tipoRetorno método(parámetros);

Los métodos estáticos se pueden llamar sin crear una instancia de la clase. La instrucción en Java tiene el siguiente formato:

NombreClase.nombreMétodo (argumentos)

Tanto los atributos como los métodos estáticos están diseñados para ser compartidos entre todos los objetos creados a partir de la misma clase. Los métodos estáticos no pueden ser redefinidos, pero pueden sobrecargarse.

Objetivo de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para definir y utilizar atributos y métodos estáticos en una clase.

Enunciado: clase Atleta

Realizar un programa en Java que permita modelar un atleta, el cual tiene los siguientes atributos de instancia:

- Identificador del atleta de tipo *int*.
- Nombre del atleta de tipo *String*.
- Tiempo en realizar una prueba de 400 metros por relevos, de tipo *double*.

Además, se tienen los siguientes atributos estáticos:

- Un contador con la cantidad de atletas creados, el cual servirá como identificador de un atleta.
- Un atributo estático denominado *selección* de tipo *String* con el valor "Colombia".
- Un atributo estático denominado *tiempoEquipo* que sume los valores de tiempo obtenidos por un equipo de atletas.

Es necesario crear un constructor para la clase con los parámetros: nombre y tiempo de cada atleta. En el constructor se debe inicializar el identificador del atleta con base en el contador estático.

También se deben implementar los siguientes métodos:

- Correr 400 metros: este método actualiza el tiempo total de carrera del equipo sumando el tiempo de carrera de cada atleta.
- Un método estático que imprima el nombre de la selección de atletismo, el cual es un atributo estático.
- Un método estático que imprima el tiempo total obtenido por el equipo de atletismo.
- Un método *main* que debe crear cuatro atletas que conforman el equipo de relevos de 400 metros, con los valores de la tabla 3.1.

Tabla 3.1. Objetos atletas

Objeto	Nombre	Tiempo (segundos)
atleta1	Alejandro Perlaza	9.55
atleta2	Anthony Zambrano	9.28
atleta3	Diego Palomeque	9.53
atleta4	Gilmar Herrera	9.29

El método *main* debe imprimir la información de cada atleta, el nombre de la selección y el tiempo total obtenido por el equipo.

Solución

Clase: Atleta

```
/**
 * Esta clase define objetos que representan un Atleta con un identificador,
 * nombre, selección a la que pertenecen, tiempo obtenido en la
 *competición y tiempo del equipo.
 * @version 1.2/2020
 */
public class Atleta {
    /* Atributo estático que se incrementa con la creación de un atleta y
       se asigna como identificador */
    static int contador = 0; /* Atributo estático que cuenta los atletas
       creados */
    int identificador; // Atributo que define el identificador de un atleta
    String nombre; // Atributo que define el nombre de un atleta
    /* Atributo que define el tiempo obtenido por un atleta en una
       competicion atlética */
    double tiempo;
    // Atributo estático que define el nombre de la selección de un atleta
    static String selección = "Colombia";
    /* Atributo estático que totaliza los tiempos de cada atleta para
       obtener el tiempo del equipo */
    static double tiempoEquipo;
```

```
/**
* Constructor de la clase Atleta
* @param nombre Parámetro que define el nombre completo del
* atleta
* @param tiempo Parámetro que define el tiempo obtenido por el
* atleta en una competición deportiva
*/
public Atleta(String nombre, double tiempo) {
    /* Cuando se crea un atleta se incrementa este contador para
       calcular el total de atletas */
    contador++;
    identificador = contador; // Se asigna el contador al identificador
    this.nombre = nombre;
    this.tiempo = tiempo;
}

/**
* Método para acumular el tiempo total del equipo a partir de la
* suma de los tiempos de cada atleta en una carrera de 400 metros
*/
public void correr400metros() {
    tiempoEquipo = tiempoEquipo + tiempo;
}

/**
* Método para mostrar en pantalla los datos de un atleta
*/
public void imprimir() {
    System.out.println("Identificador del atleta = " + identificador);
    System.out.println("Nombre del atleta = " + nombre);
    System.out.println("Tiempo del atleta = " + tiempo + "
        segundos");
    System.out.println();
}

/**
* Método estático para mostrar en pantalla el nombre de la selección
* del equipo
*/
public static void imprimirSelección() {
    System.out.println("Selección = " + selección);
```

```
    }

    /**
    * Método estático para mostrar en pantalla el tiempo obtenido por
    * el equipo
    */
    public static void imprimirTiempoEquipo() {
        System.out.println("Tiempo del equipop = " + tiempoEquipo + "
            segundos");
    }

    /**
    * Método main para instanciar cuatro atletas, mostrar sus datos en
    * pantalla y obtener el tiempo total del equipo
    */
    public static void main (String args[]) {
        Atleta atleta1 = new Atleta("Alejandro Perlaza",9.55);
        atleta1.correr400metros();
        Atleta atleta2 = new Atleta("Anthony Zambrano",9.28);
        atleta1.correr400metros();
        Atleta atleta3 = new Atleta("Diego Palomeque",9.53);
        atleta1.correr400metros();
        Atleta atleta4 = new Atleta("Gilmar Herrera",9.29);
        atleta1.correr400metros();
        Atleta.imprimirSelección();
        atleta1.imprimir();
        atleta2.imprimir();
        atleta3.imprimir();
        atleta4.imprimir();
        Atleta.imprimirTiempoEquipo();
    }
}
```

Diagrama de clases

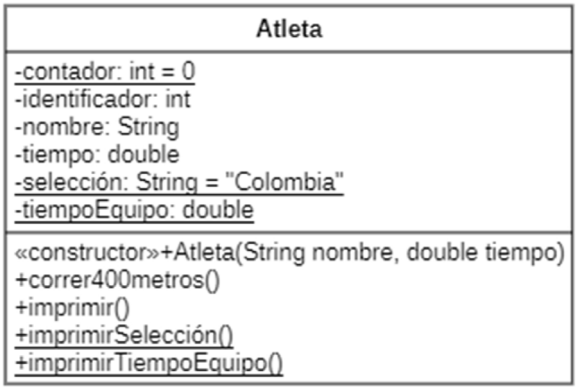


Figura 3.1. Diagrama de clases del ejercicio 3.1.

Explicación del diagrama de clases

Se ha definido una clase denominada Atleta, que tiene tres atributos de instancia privados: identificador, nombre y tiempo. Además, posee tres atributos estáticos: contador, selección y tiempoEquipo. Los atributos estáticos se representan en UML con el texto en subrayado. Los atributos contador y selección tienen asignados sus valores iniciales.

La clase posee un constructor y dos métodos de instancia públicos: correr400metros e imprimir. También posee dos métodos adicionales públicos, pero que son estáticos: imprimirSelección e imprimirTiempoEquipo (estos se identifican por el texto en subrayado).

Tanto los atributos como los métodos estáticos son compartidos por todas las instancias de la clase.

Diagrama de objetos

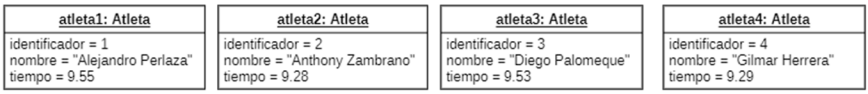


Figura 3.2. Diagrama de objetos del ejercicio 3.1.

Ejecución del programa

```
Selección = Colombia
Identificador del atleta = 1
Nombre del atleta = Alejandro Perlaza
Tiempo del atleta = 9.55 segundos

Identificador del atleta = 2
Nombre del atleta = Anthony Zambrano
Tiempo del atleta = 9.28 segundos

Identificador del atleta = 3
Nombre del atleta = Diego Palomeque
Tiempo del atleta = 9.53 segundos

Identificador del atleta = 4
Nombre del atleta = Gilmar Herrera
Tiempo del atleta = 9.29 segundos

Tiempo del equipop = 38.2 segundos
```

Figura 3.3. Ejecución del programa del ejercicio 3.1.

Ejercicios propuestos

- Definir una clase Temperatura que contenga los siguientes métodos estáticos:
  - Un método para convertir grados Fahrenheit a Celsius, el cual debe recibir como parámetro un valor de temperatura en grados Fahrenheit.
  - Un método para convertir grados Celsius a Fahrenheit, el cual debe recibir como parámetro un valor de temperatura en grados Celsius.
- Definir una clase Empleado que tenga como atributos: identificador del empleado, nombre, apellidos y salario mensual. El identificador es un número consecutivo que se incrementa cada vez que un empleado se crea. Además, se requiere un método estático que calcule el total de la nómina de empleados de la empresa.

Ejercicio 3.2. Clase *String*

La clase *String* representa cadenas que se utilizan para almacenar texto. Una variable de tipo *String* contiene una colección de caracteres (tipo *char*) rodeados por comillas dobles.

A continuación, se presenta un listado de métodos pertenecientes a la clase *String* (API Java, 2020).

Instrucciones Java del ejercicio

Tabla 3.2. Instrucciones Java del ejercicio 3.2.

Tipo de método	Método	Propósito
Obtención de caracteres	*char* charAt(*int* índice)	Retorna el carácter indicado por el índice.
	String subString(*int* inicio, *int* fin)	Retorna un nuevo *String* con los caracteres del primero que van desde la posición de inicio hasta la posición de fin.
Longitud y concatenación	*int length*()	Retorna la longitud del *String*.
	String concat(*String*)	Retorna un nuevo *String* concatenando el valor del *String* pasado como parámetro al valor del *String* actual.
Comparación	*int* compareTo(*String* cadena)	Compara si un *String* es igual, mayor o menor que otro. Regresa 0 si las cadenas son idénticas; un valor menor que 0 si la primera es menor; o un valor mayor que 0, en caso contrario.
	boolean equals(*String*)	Retorna verdadero si el valor del *String* pasado es el mismo que el actual.
	boolean equalsIgnoreCase(*String*s)	Compara dos *Strings*, pero sin distinguir entre mayúsculas y minúsculas.
Modificación del *String*	*String* replace(*char* antiguo, *char* nuevo)	Crea un nuevo *String* con el valor del *String* actual, pero cambiando todas las apariciones del carácter "antiguo" por las del "nuevo".
	char[] toCharArray()	Crea un *array* de caracteres y convierte el *String* a un *array* de *chars*.
	String toLowerCase()	Retorna un nuevo *String* con todos los caracteres en minúsculas.
	String toUpperCase()	Retorna un nuevo *String* con todos los caracteres en mayúsculas.
	String trim()	Retorna una copia del *String* con espacios en blanco iniciales y finales omitidos.
Conversión de otros tipos a *String*	tipo *String*.valueOf()	Permite convertir diferentes tipos de datos a *String*.
Búsqueda de caracteres	*int* indexOf(int c)	Permite buscar caracteres y patrones de caracteres o subcadenas.

Objetivo de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad de aplicar diferentes métodos para manipular *Strings* en Java.

Enunciado: clase Cadena

Se desea construir un programa que dado un *String* con valor inicial de: "Programación Orientada a Objetos ", realice las siguientes operaciones:

1. Obtener la longitud de dicho *String*.
2. Eliminar los espacios en blanco del *String* obtenido en el paso anterior.
3. Pasar todos los caracteres del *String* (obtenido en el paso anterior) a mayúsculas.
4. Concatenar al *String* (obtenido en el paso anterior) el *String* "12345".
5. Extraer del *String* (obtenido en el paso anterior), un *Substring* desde la posición 10 al 15.
6. Reemplazar en el *String* (obtenido en el paso anterior) el carácter "o" por "O".
7. Comparar el *String* (obtenido en el paso anterior) con el *String* "Programación".

Después de cada paso, se debe mostrar el resultado en pantalla.

Solución

Clase: Cadena

```
/**
 * Esta clase define objetos que representan una cadena de texto con un
 * atributo que representa la cadena con su valor inicial
 * @version 1.2/2020
 */
public class Cadena {
    // La cadena inicial tiene espacios en blanco al comienzo y al final
    String cadenaInicial = "   Programación Orientada a Objetos   ";

    /**
```

```
        * Método main que realiza varias operaciones que modifican
        * sucesivamente la cadena inicial
        */
        public static void main(String args[]) {
            Cadena cadena = new Cadena();
            // Determina la longitud de la cadena
            int longitud = cadena.cadenaInicial.length();
            System.out.println("La longitud del String es = " + longitud);
            // Elimina espacios en blanco al comienzo y al final
            String cadenaSinEspacios = cadena.cadenaInicial.trim();
            System.out.println("El String sin espacios en blanco = " + cade-
                naSinEspacios);
            // Convierte los caracteres de la cadena a mayúscula
            String cadenaMayúscula = cadenaSinEspacios.toUpperCase();
            System.out.println("El String en mayúscula = " + cadenaMayúscula);
            String cadenaConcatenada = cadenaMayúscula.concat("12345");
                // Concatena cadenas
            System.out.println("El String concatenado = " + cadenaConcate-
                nada);
            // Extrae una porción de la cadena
            String cadenaExtraida = cadenaConcatenada.subString(24,31);
            System.out.println("El String extraído = " + cadenaExtraida);
            // Reemplaza el carácter especificado
            String cadenaReemplazada = cadenaExtraida.replace("O","X");
            System.out.println("El String reemplazado = " + cadenaReempla-
                zada);
            boolean comparación = cadenaReemplazada.equals("Programa-
                ción"); // Compara cadenas
            System.out.println("Los String son iguales = " + comparación);
        }
    }
```

Diagrama de clases

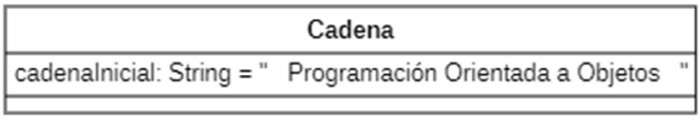


Figura 3.4. Diagrama de clases del ejercicio 3.2.

Explicación del diagrama de clases

Se ha definido una clase denominada Cadena con un atributo de tipo *String* junto con su valor inicial. La clase no tiene métodos especificados.

Diagrama de objetos

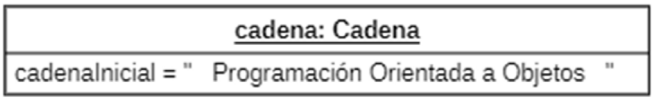


Figura 3.5. Diagrama de objetos del ejercicio 3.2.

Ejecución del programa

```
La longitud del String es = 38
El String sin espacios en blanco = Programación Orientada a Objetos
El String en mayúscula = PROGRAMACIÓN ORIENTADA A OBJETOS
El String concatenado = PROGRAMACIÓN ORIENTADA A OBJETOS12345
El String extraído =  OBJETO
El String reemplazado =  XBJETX
Los String son iguales = false
```

Figura 3.6. Ejecución del programa del ejercicio 3.2.

Ejercicios propuestos

- Hacer una clase que contenga métodos que realicen las siguientes acciones:
  - Un método que reciba como parámetro un *String* y calcule cuántas mayúsculas tiene.
  - Un método que reciba como parámetro un *String* y una letra, y determine cuántas veces está la letra en el *String* (la letra puede estar en mayúscula o minúscula).
  - Un método que reciba como parámetro un *String*, y que elimine todos sus espacios en blanco y escriba en pantalla el *String* resultante.
  - Un método que reciba como parámetro un *String*, y obtenga el *String* escrito al revés.

Ejercicio 3.3. *Wrappers*

Un *wrapper* (contenedor o envoltorio) es una clase cuyos objetos contienen tipos primitivos de datos. Los *wrappers* convierten tipos primitivos de datos en objetos. Los *wrappers* son objetos que se pueden clasificar en *Number*, *Character* o *Boolean* como se observa en la figura 3.7. A su vez, los objetos *Number* pueden ser *Byte*, *Short*, *Integer*, *Long*, *Float* y *Double*.

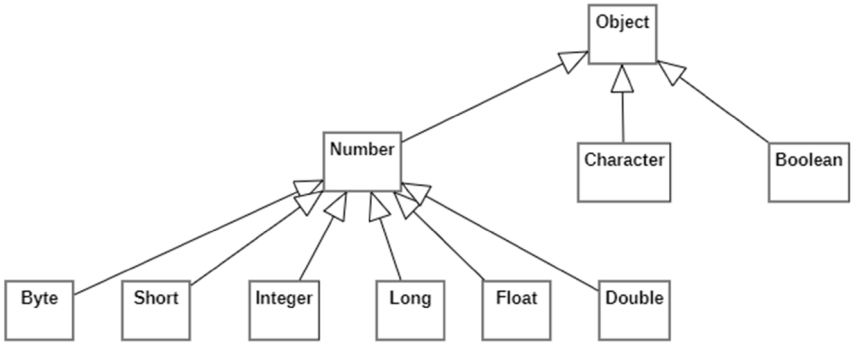


Figura 3.7. Jerarquía de clases de *wrappers*

Los *wrappers* definidos en Java están en la tabla 3.3 (API Java, 2020).

Tabla 3.3. *Wrappers* definidos en Java

Tipo de dato primitivo	*Wrapper*	Principales métodos
char	Character	**Constructor Character(*char* ch):** construye un objeto *Character* a partir del *char ch* pasado como parámetro. **public static boolean isLowerCase(*char* ch)**: retorna *true* si *ch* es una letra minúscula. **public static boolean isUpperCase(*char* ch)**: retorna *true* si *ch* es una letra mayúscula. **public static boolean isDigit(*char* ch)**: retorna *true* si *ch* es un dígito. **public static boolean isLetter(*char* ch)**: retorna *true* si *ch* es una letra. **public static boolean isLetterOrDigit(*char* ch)**: retorna *true* si *ch* es una letra o un dígito. **public static char toLowerCase(*char* ch)**: retorna *ch* convertido en minúscula. **public static char toUpperCase(*char* ch)**: retorna *ch* convertido en mayúscula.

Tipo de dato primitivo	*Wrapper*	Principales métodos
int	Integer	**Constructor Integer(*int* i):** construye un objeto *Integer* a partir del *int i* pasado como parámetro. **public static *int* parseInt(*String* str):** retorna el valor del entero del *String str* en la base 10. **public static Integer valueOf(*String* str, *int* radix):** equivalente a *parseInt()* pero retorna un objeto *Integer.*
long	*Long*	**Constructor Long(*long* i):** construye un objeto *Long* a partir del *long i* pasado como parámetro. **public static long parseLong(*String* str):** equivalente al anterior, pero con base 10. **public static Long valueOf(*String* str, *int* radix):** equivalente a *parseLong()*, pero retorna un objeto *Long.*
float	*Float*	**Constructor Float(*float* f):** construye un objeto *Float* a partir del *float f* pasado como parámetro. **public float floatValue():**retorna el valor convertido a tipo *float*.
double	*Double*	**Constructor Double(*double* d):** construye un objeto *Double* a partir del *double d* pasado como parámetro. **public double doubleValue()**:retorna el valor convertido a tipo *double*.
boolean	*Boolean*	**Constructor Boolean(*String* str):** crea un objeto Boolean que representa el valor *true* o *false* de acuerdo al *String*, ignorando mayúsculas y minúsculas.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Crear *wrappers* a partir de diferentes tipos primitivos de datos.
- Realizar diferentes operaciones sobre los *wrappers* creados.

Enunciado: clase Envoltorio

Se desea construir un programa para manipular los tipos primitivos presentados en la tabla 3.4.

Tabla 3.4. Tipos primitivos de datos

Tipo primitivo	Variable	Valor
byte	a	1
int	b	100
float	c	28.9f
double	d	271.8
char	e	'M'

- Definir un método que convierta los tipos primitivos de datos presentados en la tabla en sus respectivos *wrappers* y, luego, imprimir sus valores en pantalla.
- Definir un método que realice las conversiones inversas, es decir, que convierta los *wrappers* anteriores en sus tipos primitivos de datos e imprima sus valores en pantalla.
- Definir un método que evalúe si el valor de *e*:
  - Está en mayúscula.
  - Está en minúscula.
  - Es una letra.
  - En un dígito.
  - Convierta el carácter en minúscula.

Solución

Clase: Envoltorio

```
/**
 * Esta clase denominada Envoltorio que utiliza diferentes wrappers
 * asociados a tipos de datos.
 * @version 1.2/2020
 */
public class Envoltorio {
    /* Atributos que son tipos primitivos de datos, cada uno con su
       valor inicial */
    byte a = 7;
    int b = 100;
    float c = 28.9f;
    double d = 271.8;
    char e = 'M';
    /* Atributos que son envoltorios (wrappers). Cada envoltorio está
       asociado a su correspondiente tipo primitivo de dato */
    Byte objetoByte;
    Integer objetoInteger;
    Float objetoFloat;
    Double objetoDouble;
    Character objetoChar;
```

```
/**
* Método que crea un envoltorio a partir de su tipo primitivo de
* dato y muestra su valor en pantalla
*/
public void convertirToWrapper() {
    objetoByte = new Byte(a);
    objetoInteger = new Integer(b);
    objetoFloat = new Float(c);
    objetoDouble = new Double(d);
    objetoChar = e; /* Character no tiene un constructor como los
        otros envoltorios */

    System.out.println("Objeto Byte =  " + objetoByte);
    System.out.println("Objeto Integer = " + objetoInteger);
    System.out.println("Objeto Float =   " + objetoFloat);
    System.out.println("Objeto Double = " + objetoDouble);
    System.out.println("Objeto Character = " + objetoChar);
}

/**
* Método que convierte los envoltorios en tipos primitivos de datos
* y muestra el resultado en pantalla. Realiza la acción inversa del
* método convertirToWrapper
*/
public void convertirToTipoPrimitivo() {
    byte tipoByte = objetoByte;
    int tipoInt = objetoInteger;
    float tipoFloat = objetoFloat;
    double tipoDouble = objetoDouble;
    char tipoChar = objetoChar;

    System.out.println("Tipo byte = " + tipoByte);
    System.out.println("Tipo int = " + tipoInt);
    System.out.println("Tipo float = " + tipoFloat);
    System.out.println("Tipo double  = " + tipoDouble);
    System.out.println("Tipo char = " + tipoChar);
}

/**
* Método que realiza varias acciones en un objeto de tipo Character
*/
public void consultarChar() {
    boolean esMiníscula = Character.isLowerCase(e); /* Determina si
        el carácter está en minúscula */
```

```
            System.out.println("¿El carácter e = " + e + " está en minúscula?
                = " + esMiníscula);
            // Determina si el carácter está en mayúscula
            boolean esMayúscula = Character.isUpperCase(e);
            System.out.println("¿El carácter e = " + e + " está en mayúscula?
                = " + esMayúscula);
            boolean esLetra = Character.isLetter(e); /* Determina si el
                carácter es una letra */
            System.out.println("¿El carácter e = " + e + " está una letra? = " +
                esLetra);
            boolean esDigito = Character.isDigit(e); /* Determina si el
                carácter es un dígito */
            System.out.println("¿El carácter e = " + e + " es un dígito? = " +
                esDigito);
            char charMinúscula = Character.toLowerCase(e); /* Convierte el
                carácter a minúscula */
            System.out.println("Carácter e = " + e + " convertido a minúscula
                = " + charMinúscula);
        }

        /**
        * Método main que instancia un objeto Envoltorio y prueba los
        * métodos para convertir de tipo primitivo de dato a envoltorio, de
        * envoltorio a tipo primitivo de dato y diferentes métodos del
        * envoltorio Character
        */
        public static void main(String args[]) {
            Envoltorio wrapper = new Envoltorio();
            wrapper.convertirToWrapper();
            wrapper.convertirToTipoPrimitivo();
            wrapper.consultarChar();
        }
    }
```

Diagrama de clases

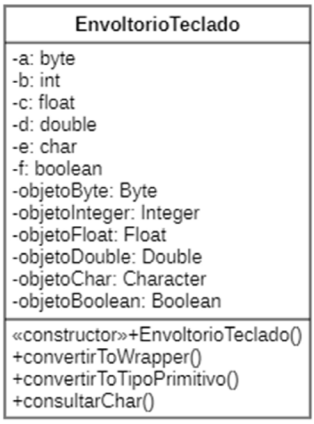


Figura 3.8. Diagrama de clases del ejercicio 3.3.

Explicación del diagrama de clases

Se ha definido una clase denominada Envoltorio con atributos privados (-), que representan tipos primitivos de datos y sus envoltorios (*wrappers*) correspondientes. La clase tienen tres métodos públicos: para convertir los tipos primitivos de datos a envoltorios (convertirToWrapper), convertir el envoltorio a tipo primitivo de dato (convertirToPrimitivo) y para realizar diferentes transformaciones de un envoltorio *Character* (consultarChar).

Diagrama de objetos

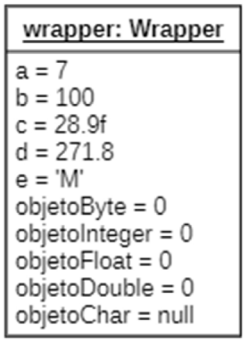


Figura 3.9. Diagrama de objetos del ejercicio 3.3.

Ejecución del programa

```
Objeto Byte =  7
Objeto Integer = 100
Objeto Float =   28.9
Objeto Double = 271.8
Objeto Character = M
Tipo byte = 7
Tipo int = 100
Tipo float = 28.9
Tipo double  = 271.8
Tipo char = M
¿El caracter e = M está en minúscula? = false
¿El caracter e = M está en mayúscula? = true
¿El caracter e = M está una letra? = true
¿El caracter e = M es un dígito? = false
Caracter e = M convertido a minúscula = m
```

Figura 3.10. Ejecución del programa del ejercicio 3.3.

Ejercicios propuestos

- Crear un *array* de caracteres y luego ordenarlo alfabéticamente. Para conocer cómo trabajar con *arrays* consultar el apartado 1.5 de este libro.
- Crear un programa que imprima en pantalla los valores mínimos y máximo de un entero (*int*), *long*, *float* y *double*.
- Crear un método que reciba como parámetro un tipo de dato *int*, luego, convertirlo a *long*, a *float* y, finalmente, imprimir en pantalla los tres valores.

Ejercicio 3.4. Entrada de datos desde teclado y *wrappers*

La entrada de datos desde teclado en Java utiliza el objeto *Scanner* (Arroyo-Díaz, 2019a). El formato es el siguiente:

Scanner sc = new Scanner(System.in)

Para utilizar la clase *Scanner* se debe importar el paquete *java.util* al inicio del programa.

En la tabla 3.5 se presenta un listado de métodos de la clase *Scanner* para leer diferentes tipos de datos.

Tabla 3.5. Métodos de la clase *Scanner*

Método	Descripción
nextLine()	Leer un valor *String* ingresado por teclado.
nextBoolean()	Lee un valor de tipo *boolean* ingresado por teclado.
nextByte()	Lee un valor *byte* ingresado por teclado.
nextDouble()	Lee un valor *double* ingresado por teclado.
nextFloat()	Lee un valor *float* ingresado por teclado.
nextInt()	Lee un valor *int* ingresado por teclado.
nextLine()	Lee un valor *String* ingresado por teclado.
nextLong()	Lee un valor *long* ingresado por teclado.
nextShort()	Lee un valor *short* ingresado por teclado.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Capturar datos ingresados por teclado utilizando la clase *Scanner*.
- Capturar tipos diferentes de datos.

Enunciado: clase EnvoltorioTeclado

Se desea construir un programa que capture por teclado los valores de la clase Envoltorio (del ejercicio anterior 3.3).

Agregar a la clase un atributo *f* de tipo *boolean*, cuyo valor por teclado pueda a su vez ser leído.

Solución

Clase: EnvoltorioTeclado

```
import java.util.*;
/**
* Esta clase denominada EnvoltorioTeclado modifica el código de la clase
* Envoltorio para solicitar entrada de datos por teclado.
* @version 1.2/2020
*/
```

```
public class EnvoltorioTeclado {
    // Atributos que son tipos primitivos de datos
    byte a;
    int b;
    float c;
    double d;
    char e;
    boolean f;

    /* Atributos que son envoltorios (wrappers). Cada envoltorio está
       asociado a su correspondiente tipo primitivo de dato */
    Byte objetoByte;
    Integer objetoInteger;
    Float objetoFloat;
    Double objetoDouble;
    Character objetoChar;
    Boolean objetoBoolean;

    /**
    * Constructor de la clase EnvoltorioTeclado. Solicita ingreso de
    * datos por teclado y cada dato ingresado se almacena en un
    * atributo de la clase
    */
    EnvoltorioTeclado() {
        /* Crea un objeto Scanner que gestiona la captura de datos por
           teclado */
        Scanner sc = new Scanner(System.in);
        System.out.print("Ingresar dato tipo bye = ");
        a = sc.nextByte(); // Lee un dato de tipo byte
        System.out.print("Ingresar dato tipo int = ");
        b = sc.nextInt(); // Lee un dato de tipo int
        System.out.print("Ingresar dato tipo float = ");
        c = sc.nextFloat(); // Lee un dato de tipo float
        System.out.print("Ingresar dato tipo double = ");
        d = sc.nextDouble(); // Lee un dato de tipo double
        System.out.print("Ingresar dato tipo char = ");
        e = sc.next().charAt(0); // Lee un dato de tipo char
        System.out.print("Ingresar dato tipo boolean = ");
        f = sc.nextBoolean(); // Lee un dato de boolean
    }
```

```
/**
* Método que crea un envoltorio a partir de su tipo primitivo de
* dato y muestra su valor en pantalla
*/
public void convertirToWrapper() {
    objetoByte = new Byte(a);
    objetoInteger = new Integer(b);
    objetoFloat = new Float(c);
    objetoDouble = new Double(d);
    objetoChar = new Character(e);
    objetoBoolean = new Boolean(f);

    System.out.println("Objeto Byte = " + objetoByte);
    System.out.println("Objeto Integer = " + objetoInteger);
    System.out.println("Objeto Float =   " + objetoFloat);
    System.out.println("Objeto Double = " + objetoDouble);
    System.out.println("Objeto Character = " + objetoChar);
    System.out.println("Objeto Booelan = " + objetoBoolean);
}

/**
* Método que convierte los envoltorios en tipos primitivos de datos
* y muestra el resultado en pantalla. Realiza la acción inversa del
* método convertirToWrapper
*/
public void convertirToTipoPrimitivo() {
    byte tipoByte = objetoByte;
    int tipoInt = objetoInteger;
    float tipoFloat = objetoFloat;
    double tipoDouble = objetoDouble;
    char tipoChar = objetoChar;
    boolean tipoBoolean = objetoBoolean;

    System.out.println("Tipo byte = " + tipoByte);
    System.out.println("Tipo int = " + tipoInt);
    System.out.println("Tipo float = " + tipoFloat);
    System.out.println("Tipo double  = " + tipoDouble);
    System.out.println("Tipo char = " + tipoChar);
}
```

```
    /**
    * Método que realiza varias acciones en un objeto de tipo Character
    */
    public void consultarChar() {
        // Determina si el carácter está en minúscula
        boolean esMiníscula = Character.isLowerCase(e);
        System.out.println("¿El carácter e = " + e + " está en minúscula? =
            " + esMiníscula);
        // Determina si el carácter está en mayúscula
        boolean esMayúscula = Character.isUpperCase(e);
        System.out.println("¿El carácter e = " + e + " está en mayúscula? =
            " + esMayúscula);
        boolean esLetra = Character.isLetter(e); /* Determina si el carácter
            es una letra */
        System.out.println("¿El carácter e = " + e + " está una letra? = " +
            esLetra);
        boolean esDigito = Character.isDigit(e); /* Determina si el carácter
            es un dígito */
        System.out.println("¿El carácter e = " + e + " es un dígito? = " +
            esDigito);
        char charMinúscula = Character.toLowerCase (e); /* Convierte el
            carácter a minúscula */
        System.out.println("Carácter e = " + e + " convertido a minúscula
            = " + charMinúscula);
    }
    /**
    * Método main que instancia un objeto EnvoltorioTeclado y prueba
    * los métodos para convertir de tipo primitivo de dato a envoltorio,
    * de envoltorio a tipo primitivo de dato y diferentes métodos sobre
    * el envoltorio Character
    */
    public static void main(String args[]) {
        EnvoltorioTeclado wrapper = new EnvoltorioTeclado();
        wrapper.convertirToWrapper();
        wrapper.convertirToTipoPrimitivo();
        wrapper.consultarChar();
    }
}
```

Diagrama de clases

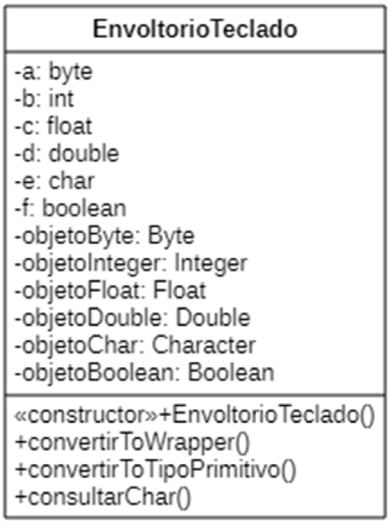


Figura 3.11. Diagrama de clases del ejercicio 3.4.

Explicación del diagrama de clases

Se ha definido una clase denominada EnvoltorioTeclado con atributos privados que representan tipos primitivos de datos y sus envoltorios (*wrappers*) correspondientes. La clase tiene un constructor y tres métodos públicos para convertir los tipos primitivos de datos a envoltorios (convertirToWrapper); el envoltorio a tipo primitivo de dato (convertirToPrimitivo) y para realizar diferentes transformaciones de un envoltorio *Character* (consultarChar). Se diferencia del diagrama del ejercicio anterior en que tiene un constructor que inicializa los atributos (tipos primitivos de datos), solicitando que se ingresen sus valores por teclado.

Diagrama de objetos

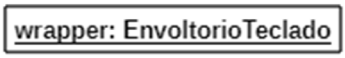


Figura 3.12. Diagrama de objetos del ejercicio 3.4.

Ejecución del programa

```
Ingresar dato tipo byte = 20
Ingresar dato tipo int = 500
Ingresar dato tipo float = 29,5
Ingresar dato tipo double = 521,3456
Ingresar dato tipo char = T
Ingresar dato tipo boolean = trUE
Objeto Byte = 20
Objeto Integer = 500
Objeto Float =   29.5
Objeto Double = 521.3456
Objeto Character = T
Objeto Booelan = true
Tipo byte = 20
Tipo int = 500
Tipo float = 29.5
Tipo double  = 521.3456
Tipo char = T
¿El caracter e = T está en minúscula? = false
¿El caracter e = T está en mayúscula? = true
¿El caracter e = T está una letra? = true
¿El caracter e = T es un dígito? = false
Caracter e = T convertido a minúscula = t
```

Figura 3.13. Ejecución del programa del ejercicio 3.4.

Ejercicios propuestos

- Realizar el ejercicio propuesto en el apartado 3.2, pero ingresando los datos de entrada por teclado.

Ejercicio 3.5. Clases internas

Java permite declarar clases dentro de una clase. Para acceder a los atributos y métodos de una clase interna se requiere instanciar un objeto de la clase interna (Liang, 2017). Una clase interna tiene acceso a todas las variables y métodos de su clase externa (incluido los privados).

La sintaxis para escribir una clase interna es:

```
class ClaseExterna {
        class ClaseInterna{
        }
}
```

Para crear una instancia de la clase interna, primero, se debe crear una instancia de la clase donde está contenida y, luego, instanciar la clase interna siguiendo el siguiente formato:

```
ClasePrincipal objetoExterno = new ClasePrincipal();
ClaseInterna objetoInterno = ClasePrincipal.new ClaseInterna();
```

Para invocar un método de la clase interna se debe indicar el nombre del objeto de la clase contenedora:

```
objetoExterno.objetoInterno.método();
```

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Definir una clase interna dentro de otra clase.
- Crear instancias de una clase interna.
- Invocar métodos definidos dentro de una clase interna.

Enunciado: clase Medicamento

Se requiere desarrollar un programa que modele la siguiente información de un medicamento:

- Nombre del medicamento.
- Fabricante del medicamento.
- Vía de administración del medicamento.

La clase medicamento debe tener una clase interna, que represente la posología del medicamento con la siguiente información:

- Usuarios del medicamento.
- Dosis del medicamento en miligramos
- Periodo de tiempo para tomar el medicamento.
- Recomendaciones.

En un método *main* se debe crear el medicamento. También se debe crear una posología y asignársela al medicamento creado. Se deben mostrar en pantalla los datos del medicamento y de su posología.

Solución

Clase: Medicamento

```
/**
 * Esta clase denominada Medicamento modela un medicamento con los
 * atributos nombre del medicamento, su fabricante, vía en que se
 * administra el medicamento y una posología para el medicamento
 * @version 1.2/2020
 */
public class Medicamento {
    String nombre; // Atributo que define el nombre de un medicamento
    String fabricante; /* Atributo que define el nombre del fabricante de
        un medicamento */
    String víaAdministración; /* Atributo que define la vía de
        administración de un medicamento */
    Posología posología; /* Atributo que define la posología de un
        medicamento */

    /**
    * Constructor de la clase Medicamento
    * @param nombre Parámetro que define el nombre del medicamento
    * @param fabricante Parámetro que define el nombre del fabricante
    * del medicamento
    * @param víaAdministración Parámetro que define la vía de
    * administración del medicamento
    */
    Medicamento(String nombre, String fabricante, String
        víaAdministración) {
        this.nombre = nombre;
        this.fabricante = fabricante;
        this.víaAdministración = víaAdministración;
    }

    /**
    * Método que establece la posología de un medicamento
    * @param posología Parámetro que define la posología que se
    * recomienda para un medicamento
    */
    void setPosología(Posología posología) {
```

```
        this.posología = posología;
    }

    /**
    * Método que muestra en pantalla los datos de un medicamento
    */
    void imprimir() {
        System.out.println("Nombre del medicamento = " + nombre);
        System.out.println("Fabricante del medicamento = " + fabricante);
        System.out.println("Vía de administración = " +
            víaAdministración);
}

    /**
    * Esta clase "interna" se denomina Posología y modela la forma en
    * que se debe tomar un medicamento. Para ello, se indican los
    * tipos de usuarios, la dosis recomendada, el periodo de tiempo de
    * aplicación del medicamento y recomendaciones adicionales
    */
    class Posología {
        String usuarios;  /* Atributo que define los tipos de usuarios de
            un medicamento */
        int dosis; /* Atributo que define la dosis a aplicar de un
            medicamento */
        String periodo; /* Atributo que define el periodo de aplicación
            de un medicamento */
        String recomendaciones; /* Atributo que define las recomendaciones
            para un medicamento */

    /**
    * Constructor de la clase Posología
    * @param usuarios Parámetro que define los usuarios del medicamento
    * @param dosis Parámetro que define la dosis del medicamento
    * @param periodo Parámetro que define el tiempo y regularidad de
    * administración del medicamento
    * @param recomendaciones Parámetro que define posibles
    * recomendaciones adicionales sobre el uso del medicamento
    */
```

```
        Posología(String usuarios, int dosis, String periodo, String reco-
            mendaciones) {
            this.usuarios = usuarios;
            this.dosis = dosis;
            this.periodo = periodo;
            this.recomendaciones = recomendaciones;
        }
        /**
        * Método que muestra en pantalla los datos de una posología
        */
        void imprimir() {
            System.out.println("Usuarios = " + usuarios);
            System.out.println("Dosis = " + dosis);
            System.out.println("Periodo = " + periodo);
            System.out.println("Recomendaciones = " + recomendacio-
                nes);
        }
    }

    /**
    * Método que crea un medicamento y una posología para dicho
    * medicamento.
    * Luego, se muestran en pantalla tanto los datos del medicamento
    * como de su posología
    */
    public static void main(String args[]) {
        Medicamento medicamento = new Medicamento("Aspirina","Ba-
            yer","Oral");
        /* Para crear una instancia de una clase interna se debe referir al
            objeto donde está contenida la clase interna */
        Posología posología = medicamento.new Posología("Adultos y
            mayores de 16 años", 500, "6 horas", "No debe tomar este
            medicamento con el estómago vacío.");
        medicamento.setPosología(posología);
        medicamento.imprimir();
        /* Para invocar un método de la clase interna se debe indicar el
            nombre de la clase contenedora */
        medicamento.posología.imprimir();
    }
}
```

Diagrama de clases

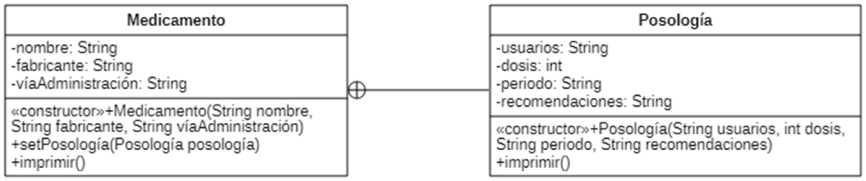


Figura 3.14. Diagrama de clases del ejercicio 3.5.

Explicación del diagrama de clases

Se han definido dos clases: Medicamento y Posología. Ambas están asociadas mediante una relación de composición especializada. En UML, este tipo de relación se expresa con una línea que tiene en un extremo un círculo con una cruz interna. La clase que tiene enlazada este círculo es la clase externa y la clase interna es la que no tiene el círculo asociado en el otro extremo de la asociación.

La clase externa Medicamento tiene tres atributos privados: nombre, fabricante y vía de administración del medicamento. Tiene un constructor y dos métodos: un método *set* para establecer la posología y un método *imprimir* para mostrar los datos del medicamento en pantalla.

La clase interna Posología tiene cuatro atributos privados: usuarios, dosis, periodo y recomendaciones. Tiene un constructor y un método imprimir para mostrar los datos de la posología en pantalla.

El objetivo de modelar conceptos utilizando clases internas es proporcionar un conjunto de servicios que solo utiliza su clase interna. La clase interna no puede existir independientemente de la clase externa. La clase interna Posología se modela como un objeto completo con los datos de una posología determinada y siempre estará asociada a un medicamento del cual forma parte. Si el medicamento se elimina, la posología también será eliminada.

Diagrama de objetos

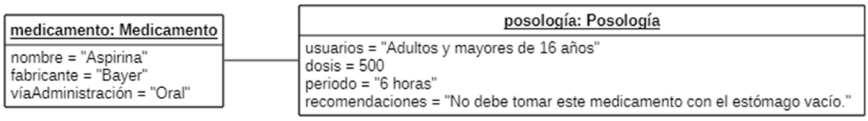


Figura 3.15. Diagrama de objetos del ejercicio 3.5.

Ejecución del programa

```
Nombre del medicamento = Aspirina
Fabricante del medicamento = Bayer
Vía de administración = Oral
Usuarios = Adultos y mayores de 16 años
Dosis = 500
Periodo = 6 horas
Recomendaciones = No debe tomar este medicamento con el estómago vacío.
```

Figura 3.16. Ejecución del programa del ejercicio 3.5.

Ejercicios propuestos

- Se desea modelar una clase Paquete de envío, la cual tiene como atributos: el remitente, el destinatario, tipo de envío, el contenido y peso del paquete. El remitente y el destinatario son de tipo persona, la cual es una clase interna con los atributos: nombres, apellidos, número de documento de identidad, dirección y teléfono. El tipo de envío es un valor enumerado con los valores: nacional o internacional. El contenido es un dato enumerado con los valores: documento o mercancía. Se requiere un método para calcular el valor del envío que depende del peso, del tipo de envío y contenido del paquete (ver tabla 3.6).

Tabla 3.6. Cálculo del valor de envío

Tipo de envío	Contenido	Peso	Valor por Kg
Nacional	Documento	<= 2 kg	$ 2000
		> 2 kg	$ 3000
	Mercancía	<= 5 kg	$ 5000
		> 5 kg	$ 7000
Internacional	Documento	<= 2 kg	$ 10 000
		> 2 kg	$ 15 000
	Mercancía	<= 5 kg	$ 12 000
		> 5 kg	$ 20 000

Ejercicio 3.6. *Arrays* de objetos

Los *arrays* son estructuras de almacenamiento muy utilizadas en los lenguajes de programación. Un *array* es un grupo de variables de tipo similar a las que se hace referencia con un nombre común (Schildt, 2018). En el apartado 1.5 se indicó la forma en que se declaran y definen los *arrays*. Estos pueden almacenar objetos y tipos primitivos de datos.

Los *arrays* de objetos se definen colocando el nombre de la clase seguido de corchetes cuadrados. El formato de definición es el siguiente:

Clase[] array;

El *array* se debe crear con un tamaño fijo:

Array = new Clase[tamaño]

Luego, el *array* se puede recorrer mediante instrucciones de iteración como *for*, *while* o *do-while*.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Definir un *array* de objetos.
- Recorrer el *array* de objetos para realizar operaciones sobre el mismo.

Enunciado: clase tienda *array* de computadores

Se requiere desarrollar un programa que modele una tienda de computadores. La tienda posee los siguientes atributos:

- Nombre de la tienda.
- Propietario de la tienda.
- Identificador tributario de la tienda.

Los computadores de la tienda tienen los siguientes atributos:

- Marca del computador.
- Cantidad de memoria.
- Características del procesador.
- Sistema operativo.
- Precio del computador.

El programa debe poseer métodos que permitan:

- Agregar un computador a la tienda.
- Eliminar un computador de la tienda dada su marca.
- Buscar un computador en la tienda dada su marca.
- Listar la información de todos los computadores que tiene la tienda.

Solución

Clase: Tienda

```
/**
 * Esta clase denominada Tienda modela una tienda de computadores
 * con los atributos nombre de la tienda, nombre de su propietario, un
 * identificador tributario, su fabricante y un array de computadores
 * @version 1.2/2020
 */
import java.util.*;

public class Tienda {
    String nombre; // Atributo que define el nombre de la tienda
    String propietario; // Atributo que define el propietario de la tienda
    int identificadorTributario; /* Atributo que define el identificador
        tributario de la tienda */
    Computador[] computadores; /* Atributo que define un array de
        computadores de la tienda */
    /* Atributo que define la cantidad de computadores que tiene la
        tienda */
    static int númeroComputadores;

    /**
    * Constructor de la clase Tienda
    * @param nombre Parámetro que define el nombre de la tienda
    * @param propietario Parámetro que define el nombre del
    * propietario de la tienda
    * @param identificadorTributario Parámetro que define un
    * identificador tributario para la tienda
    * @param tamaño Parámetro que define la cantidad de computadores
    * que tiene la tienda
    */
```

```
public Tienda(String nombre, String propietario, int identificadorTri-
    butario, int tamaño) {
    if (tamaño < 1) { // Si el tamaño es menor que 1, es insuficiente
        System.out.println("Cantidad de computadores insuficien-
            tes.");
    } else {
        this.nombre = nombre;
        this.propietario = this.propietario;
        this.identificadorTributario = identificadorTributario;
        computadores = new Computador[tamaño]; /* Se crea el array
            de computadores */
        númeroComputadores = 0;
    }
}

/**
* Método que determina si la tienda está llena, es decir, si su array
* de computadores completó su tamaño
* @return Valor booleano que determina si el array de computadores
* está lleno
*/
public boolean tiendaLlena() {
    return númeroComputadores == computadores.length;
}

/**
* Método que determina si la tienda está vacía, es decir, si su array
* de computadores no tiene elementos
* @return Valor booleano que determina si el array de
* computadores está vacío
*/
public boolean tiendaVacía() {
    return númeroComputadores == 0;
}

/**
* Método que añade un computador a la tienda
* @param computador Parámetro que define el computador que se
* agregará al array de computadores de la tienda
*/
```

```
public void añadir(Computador computador) {
    // Si la tienda está llena, no se puede agregar el computador
    if (tiendaLlena()) {
        System.out.println("La tienda está llena. No se puede añadir
            elemento.");
    } else {
        computadores[númeroComputadores] = computador;
        númeroComputadores++; /* Se incrementa el contador de
            computadores */
    }
}

/**
* Método que elimina un computador del array de computadores de
* la tienda
* @param marcaComputador Parámetro que define la marca de un
* computador que se eliminará del array de computadores de la
* tienda
* @return Valor booleano que determina si un computador se pudo
* eliminar o no del array de computadores
*/
public boolean eliminar(String marcaComputador) {
    // Se busca el computador en el array
    int pos = buscar(marcaComputador);
    if (pos < 0 ) {
        return false; /* Si no encuentra al computador, devuelve un
            valor boolean con false */
    }
    /* Si el computador se encuentra, se elimina el computador
        de su posición y se mueven las posiciones de los demás
        elementos en el array */
    for (int i = pos; i < númeroComputadores; i ++) {
        computadores[i] = computadores[i+1];
    }
    númeroComputadores--; /* Decrementa la cantidad de
        computadores */
    return true;
}
```

```
    /**
    * Método que busca un determinado computador en el array de
    * computadores
    * @param marcaComputador Parámetro que define la marca de un
    * computador que se buscará en el array de computadores de la
    * tienda
    * @return Valor entero que determina si un computador se encontró
    * o no en el array de computadores
    */
    public int buscar(String marcaComputador) {
        for (int i = 0; i < númeroComputadores; i ++) { /* Recorre el
            array de computadores */
            if (computadores[i].marca.equals(marcaComputador))
                // Si se encuentra el computador buscado
                return i; /* Retorna la posición del computador buscado
                    en el array */
        }
        return -1; // Si no encontró el computador en el array
    }

    /**
    * Método que imprime en pantalla los datos del array de
    * computadores de la tienda
    */
    public void imprimir() {
        for (int i = 0; i < númeroComputadores; i ++) {
            System.out.println("Computador" + i);
            System.out.println("Marca = " + computadores[i].marca);
            System.out.println("Cantidad de memoria = " +
                computadores[i].cantidadMemoria);
            System.out.println("Características del procesador = " +
                computadores[i].característicasProcesador);
            System.out.println("Sistema operativo = " + computadores[i].
                sistemaOperativo);
            System.out.println("Precio = " + computadores[i].precio);
        }
    }
}
```

Clase: Computador

```
/**
 * Esta clase denominada Computador modela un computador con los
 * atributos marca, cantidad de memoria, características del procesador,
 * sistema operativo y precio.
 * @version 1.2/2020
 */
public class Computador {
    String marca; // Atributo que define la marca del computador
    int cantidadMemoria; /* Atributo que define la cantidad de memoria
        del computador */
    /* Atributo que define las características del procesador del
        computador */
    String característicasProcesador;
    String sistemaOperativo; /* Atributo que define el sistema operativo
        del computador */
    double precio; // Atributo que define el precio del computador

    /**
    * Constructor de la clase Computador
    * @param marca Parámetro que define la marca de un computador
    * @param cantidadMemoria Parámetro que define la cantidad de
    * memoria de un computador
    * @param característicasProcesador Parámetro que define las
    * características del procesador
    * @param sistemaOperativo Parámetro que define el sistema
    * operativo de un computador
    * @param precio Parámetro que define el precio de un computador
    */
    public Computador(String marca, int cantidadMemoria, String
        característicasProcesador, String sistemaOperativo, double precio) {
        this.marca = marca;
        this.cantidadMemoria = cantidadMemoria;
        this.característicasProcesador = característicasProcesador;
        this.sistemaOperativo = sistemaOperativo;
        this.precio = precio;
    }
}
```

Clase: Prueba

```
import java.util.*;

/**
 * Esta clase prueba diferentes acciones realizadas por la tienda de
 * computadores.
 * @version 1.2/2020
 */
public class Prueba {

    /**
     * Método main que crea una tienda de computadores que contiene
     * tres computadores. Luego se solicita ingresar por teclado la marca
     * de un computador a buscar en la tienda
     */
    public static void main() {
        Tienda tienda = new Tienda("Tienda Cuántica","Pepito
            Pérez",123456, 5);
        Computador computador1 = new Computador("Acer", 50, "Intel
            iCore 7", "Windows", 18500000);
        Computador computador2 = new Computador("Toshiba", 80,
            "Intel iCore 5", "Windows", 15500000);
        Computador computador3 = new Computador("Mac", 100,
            "Intel iCore 7", "Mac", 2500000);
        tienda.añadir(computador1);
        tienda.añadir(computador2);
        tienda.añadir(computador3);
        Scanner sc = new Scanner(System.in);
        String marca = sc.next();
        System.out.println("El computador a buscar: " + marca + " se
            encuentra en la posición " + tienda.buscar(marca));
        tienda.imprimir();
    }
}
```

Diagrama de clases

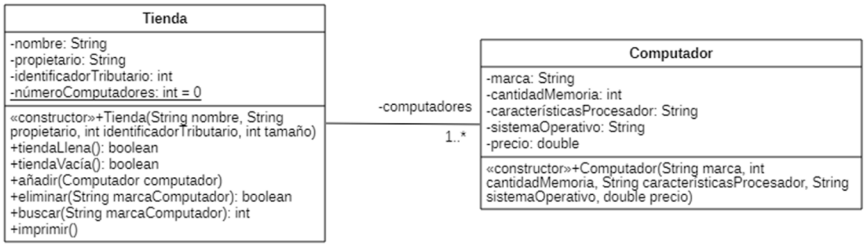


Figura 3.17. Diagrama de clases del ejercicio 3.6.

Explicación del diagrama de clases

Se han definido dos clases: Tienda y Computador. Una tienda posee muchos computadores; esta relación se modela en UML mediante una asociación que conecta las clases Tienda y Computador. La asociación tiene un rol (texto en el extremo de la asociación) que corresponde al atributo denominado "computadores", el cual es un *array* de objetos de tipo Computador.

La clase Tienda tiene los atributos privados: nombre, propietario, identificador tributario y número de computadores (el cual es estático y se representa con el texto subrayado). La clase contiene un constructor y métodos públicos para determinar si la tienda está llena (tiendaLLena)o vacía (tiendaVacía), para añadir; buscar y eliminar computadores en la tienda, y, por último, para imprimir los datos de la tienda y todos sus computadores en pantalla.

La clase Computador tiene los atributos privados: marca, cantidad de memoria, características del procesador, sistema operativo y precio. La clase Computador contiene un constructor y no tiene métodos adicionales.

Diagrama de objetos

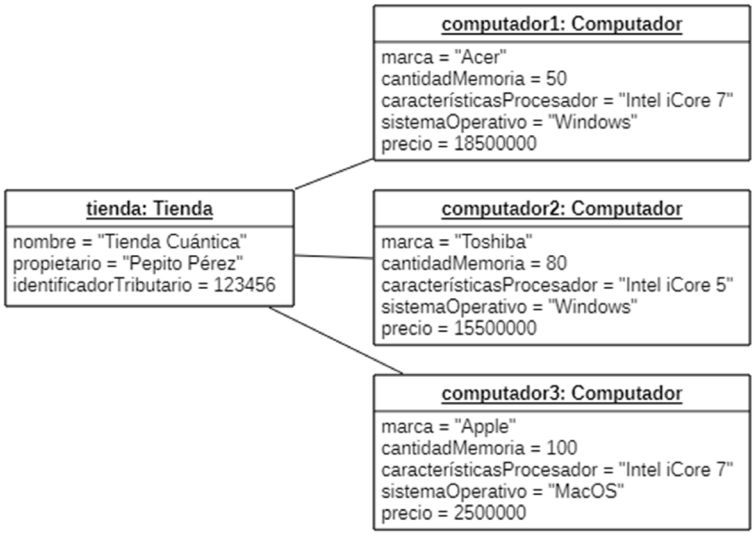


Figura 3.18. Diagrama de objetos del ejercicio 3.6.

Ejecución del programa

```
Acer
El computador a buscar: Acer se encuentra en la posición 0
Computador0
Marca = Acer
Cantidad de memoria = 50
Características del procesador = Intel iCore 7
Sistema operativo = Windows
Precio = 1.85E7
Computador1
Marca = Toshiba
Cantidad de memoria = 80
Características del procesador = Intel iCore 5
Sistema operativo = Windows
Precio = 1.55E7
Computador2
Marca = Mac
Cantidad de memoria = 100
Características del procesador = Intel iCore 7
Sistema operativo = Mac
Precio = 2500000.0
```

Figura 3.19. Ejecución del programa del ejercicio 3.6.

Ejercicios propuestos

- Se tiene un curso universitario el cual contiene un *array* de estudiantes. Para cada estudiante se tienen los datos: nombre y apellidos del estudiante, código, número de semestre y nota final del estudiante. Se requiere implementar los siguientes métodos:
  - Añadir un estudiante al curso: se ingresan por teclado los datos del estudiante. El código del estudiante debe ser único, si el código ya existe se debe generar el mensaje correspondiente.
  - Buscar un estudiante de acuerdo con su código ingresado por teclado: si se encuentra muestra los datos del estudiante. De lo contrario, debe mostrar el mensaje correspondiente.
  - Eliminar un estudiante de acuerdo con su código ingresado por teclado: si se encuentra muestra los datos del estudiante y se solicita una confirmación de la eliminación. Si no, debe mostrar el mensaje correspondiente.
  - Calcular promedio del curso: sumar las notas de los estudiantes y dividirlas por la cantidad de estudiantes que tiene el curso.
  - Obtener la cantidad de estudiantes que aprobó el curso: calcular el número de estudiantes que obtuvo un promedio mayor o igual a 3.0 y mostrarlo en pantalla. También se debe calcular el porcentaje de estudiantes que aprobó el curso.

Ejercicio 3.7. Vectores de objetos

Los vectores son estructuras de almacenamiento que implementan un *array* dinámico, lo que significa que este puede crecer o reducirse según sea necesario (Bloch, 2017). Al igual que un *array*, contiene componentes a los que se puede acceder mediante un índice entero.

Para crear un vector se utiliza el siguiente constructor:

```
Vector v = new Vector();
```

En la tabla 3.7 se presenta un listado de algunos métodos para manipular un vector.

Tabla 3.7. Métodos de la clase vector

Método	Descripción
boolean add(Object obj)	Agrega el elemento especificado al final del vector.
void add(int índice, Object obj)	Inserta el elemento especificado en una posición especificada en el vector.
void clear()	Elimina todos los elementos del vector.
Object get(int índice)	Retorna el elemento en la posición especificada en el vector.
int indexOf(Object o)	Retorna el índice de la primera aparición del elemento especificado en el vector, o -1 si el vector no contiene el elemento.
boolean isEmpty()	Prueba si el vector no tiene elementos.
boolean remove(Object o)	Elimina la primera aparición del elemento especificado en el vector. Si el vector no contiene el elemento, este no se modifica.
void removeElementAt(int índice)	Elimina la primera aparición (el índice más bajo) del parámetro en el vector.
int size()	Retorna el número de componentes del vector.
void removeAllElements()	Elimina todos los componentes del vector y establece su tamaño en cero.
void insertElementAt(Object obj, int index):	Inserta el objeto especificado como un componente en el vector en el índice especificado.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Definir un vector de objetos.
- Recorrer el vector de objetos para realizar operaciones sobre el mismo.

Enunciado: clase vector de computadores

Desarrollar el programa del ejercicio anterior (3.6), pero utilizando un vector de computadores en lugar de un *array* de computadores.

Solución

Clase: Tienda2

```
import java.util.*;
/**
 * Esta clase denominada Tienda2 modela una tienda de computadores
 * con los atributos nombre de la tienda, nombre de su propietario, un
 * identificador tributario y un vector de computadores.
 * @version 1.2/2020
 */
public class Tienda2 {
    String nombre; /* Atributo que define el nombre de la tienda de
        computadores */
    String propietario; /* Atributo que define el propietario de la tienda
        de computadores */
    /* Atributo que define el identificador tributario de la tienda de
        computadores */
    int identificadorTributario;
    Vector computadores; /* Atributo que define un vector de
        computadores */

    /**
    * Constructor de la clase Tienda2
    * @param nombre Parámetro que define el nombre de la tienda
    * @param propietario Parámetro que define el nombre del
    * propietario de la tienda
    * @param identificadorTributario Parámetro que define un
    * identificador tributario para la tienda
    * @param tamaño Parámetro que define la cantidad máxima de
    * computadores que tiene la tienda
    */
    public Tienda2(String nombre, String propietario, int
        identificadorTributario) {
        this.nombre = nombre;
        this.propietario = this.propietario;
        this.identificadorTributario = identificadorTributario;
        computadores = new Vector(); // Se crea el vector de computadores
    }
```

```
/**
* Método que determina si la tienda está llena, es decir, si su vector
* de computadores completó su tamaño
* @return Valor booleano que determina si el vector de computadores
* está lleno
*/
public boolean tiendaLlena() {
    /* Un vector no tiene un tamaño predefinido, nunca está lleno,
       devuelve siempre false */
    return false;
}

/**
* Método que determina si la tienda está vacía, es decir, si su vector
* de computadores no tiene elementos
* @return Valor booleano que determina si el vector de computadores
* está vacío
*/
public boolean tiendaVacío() {
    return computadores.size() == 0; /* El método size determinar el
       tamaño del vector */
}

/**
* Método que añade un computador a la tienda
* @param computador Parámetro que define el computador que se
* agregará al vector de computadores de la tienda
*/
public void añadir(Computador computador) {
    computadores.add(computador); /* El método add agrega un
       elemento al vector */
}

/**
* Método que elimina un computador del vector de computadores
* de la tienda
* @param marcaComputador Parámetro que define la marca de un
* computador que se eliminará del vector de computadores de la
* tienda
* @return Valor booleano que determina si un computador se pudo
* eliminar o no del vector de computadores
*/
```

```
public boolean eliminar(String marcaComputador) {
    int pos = buscar(marcaComputador); /* Busca el computador y
        devuelve su posición */
    if (pos < 0 ) { /* Si la posición es menor que cero, no encontró el
        computador */
        return false;
    }
    // Elimina el elemento que se encuentra en la posición pos
    computadores.removeElementAt(pos);
    return true;
}

/**
* Método que busca un determinado computador en el vector de
* computadores
* @param marcaComputador Parámetro que define la marca de un
* computador que se buscará en el vector de computadores de la
* tienda
* @return Valor entero que determina si un computador se encontró
* o no en el vector de computadores
*/
public int buscar(String marcaComputador) {
    Computador computador;
    // Se busca el computador en el vector
    for (int i = 0; i < computadores.size(); i ++) { /* Se recorre el
        vector de computadores */
        computador = (Computador) computadores.elementAt(i);
        if (computador.marca.equals(marcaComputador))
            return i; /* Devuelve la posición donde se encontró el
                computador */
    }
    return -1; // Si no encontró el computador, retorna -1
}
```

```
    /**
     * Método que muestra en pantalla los datos de los computadores de
     * la tienda
     */
    public void imprimir() {
        for (int i = 0; i < computadores.size(); i ++) {
            System.out.println("Computador" + i);
            Computador computador = (Computador) computadores.
                elementAt(i);
            System.out.println("Marca = " + computador.marca);
            System.out.println("Cantidad de memoria = " + computador.
                cantidadMemoria);
            System.out.println("Características del procesador = " +
                computador.característicasProcesador);
            System.out.println("Sistema operativo = " + computador.
                sistemaOperativo);
            System.out.println("Precio = " + computador.precio);
        }
    }
}
```

Clase: Computador

```
/**
 * Esta clase denominada Computador modela un computador con los
 * atributos marca, cantidad de memoria, características del procesador,
 * sistema operativo y precio.
 * @version 1.2/2020
 */
public class Computador {
    String marca; // Atributo que define la marca del computador
    int cantidadMemoria; /* Atributo que define la cantidad de memoria
        del computador */
    // Atributo que define las características del procesador del computador
    String característicasProcesador;
    String sistemaOperativo; /* Atributo que define el sistema operativo
        del computador */
    double precio; // Atributo que define el precio del computador
```

```
    /**
     * Constructor de la clase Computador
     * @param marca Parámetro que define la marca de un computador
     * @param cantidadMemoria Parámetro que define la cantidad de
     * memoria de un computador
     * @param caracterísicasProcesador Parámetro que define las
     * características del procesador de un computador
     * @param sistemaOperativo Parámetro que define el sistema
     * operativo de un computador
     * @param precio Parámetro que define el precio de un computador
     */
    public Computador(String marca, int cantidadMemoria, String
        característicasProcesador, String sistemaOperativo, double precio) {
        this.marca = marca;
        this.cantidadMemoria = cantidadMemoria;
        this.característicasProcesador = característicasProcesador;
        this.sistemaOperativo = sistemaOperativo;
        this.precio = precio;
    }
}
```

Clase: Prueba2

```
import java.util.*;
/**
 * Esta clase define una prueba con diferentes acciones realizadas por la
 * tienda de computadores que implementa un vector de computadores
 * @version 1.2/2020
 */
public class Prueba2 {

    /**
     * Método main que crea una tienda de computadores que contiene
     * tres computadores.
     * Luego se solicita ingresar por teclado la marca de un computador a
     * buscar en la tienda
     */
    public static void main() {
        Tienda2 tienda = new Tienda2("Tienda Cuántica","Pepito
            Pérez",123456);
```

```
            Computador computador1 = new Computador("Acer", 50, "Intel
                iCore 7", "Windows", 18500000);
            Computador computador2 = new Computador("Toshiba", 80,
                "Intel iCore 5", "Windows", 15500000);
            Computador computador3 = new Computador("Mac", 100,
                "Intel iCore 7", "Mac", 2500000);
            tienda.añadir(computador1);
            tienda.añadir(computador2);
            tienda.añadir(computador3);
            Scanner sc = new Scanner(System.in);
            String marca = sc.next();
            System.out.println("El computador a buscar: " + marca + " se
                encuentra en la posición " + tienda.buscar(marca));
            tienda.imprimir();
        }
    }
```

Diagrama de clases

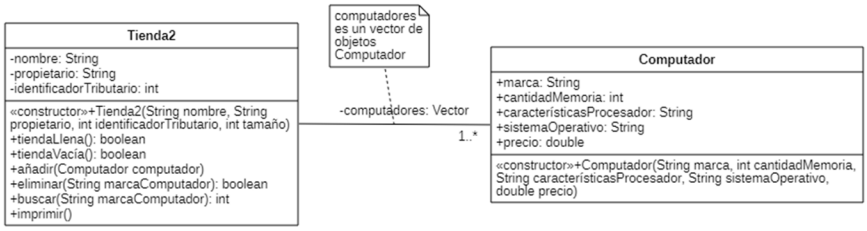


Figura 3.20. Diagrama de clases del ejercicio 3.7.

Explicación del diagrama de clases

Al comparar el diagrama UML con la solución del ejercicio anterior (utilizando un *array* en lugar de vector) se observa que, prácticamente, es igual. Solamente se ha suprimido el atributo estático "númeroComputadores" en la clase Tienda, que determinaba la cantidad de elementos que tiene un *array* y como los vectores no tienen un tamaño predefinido, este atributo no es necesario.

Consultando el diagrama UML, no hay forma de determinar aspectos de implementación del conjunto de computadores, es decir, no se puede concluir si el atributo "computadores" se implementa con un *array* o un

vector. Para hacer un diagrama que identifique en forma concreta esta solución con vectores es necesario agregar una nota en la asociación (cuadrado con el borde superior doblado) o aclarar el tipo de datos en el atributo "computadores", ambas soluciones se incluyen en el diagrama 3.7.

Diagrama de objetos

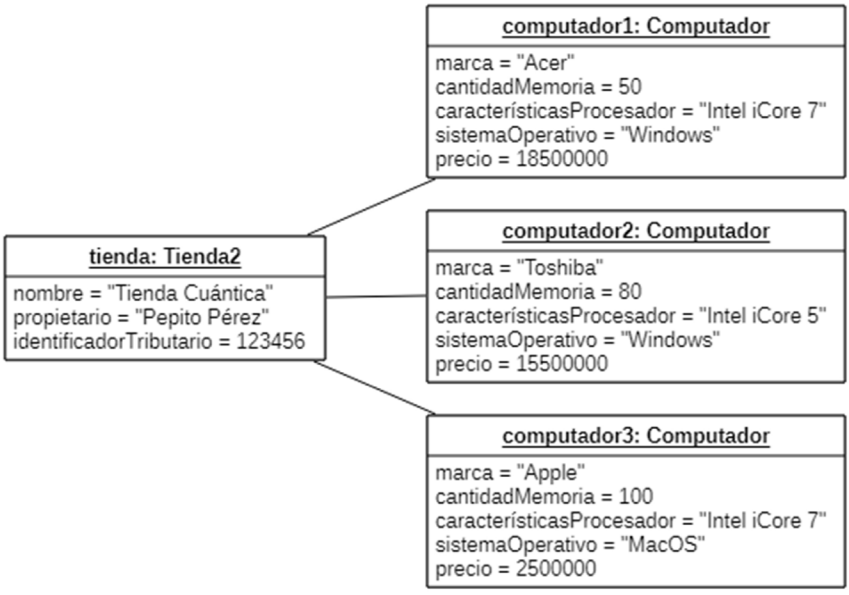


Figura 3.21. Diagrama de objetos del ejercicio 3.7.

Ejecución del programa

```
Acer
El computador a buscar: Acer se encuentra en la posición 0
Computador0
Marca = Acer
Cantidad de memoria = 50
Características del procesador = Intel iCore 7
Sistema operativo = Windows
Precio = 1.85E7
Computador1
Marca = Toshiba
Cantidad de memoria = 80
Características del procesador = Intel iCore 5
Sistema operativo = Windows
Precio = 1.55E7
Computador2
Marca = Mac
Cantidad de memoria = 100
Características del procesador = Intel iCore 7
Sistema operativo = Mac
Precio = 2500000.0
```

Figura 3.22. Ejecución del programa del ejercicio 3.7.

Ejercicios propuestos

- Realizar el ejercicio propuesto en 3.6, pero utilizando un vector como estrategia de implementación.

Capítulo 4
Herencia y polimorfismo

El propósito general del cuarto capítulo es comprender uno de los conceptos más importantes del paradigma orientado a objetos: la herencia. Conocer y aplicar en forma correcta los principios de la herencia permitirá realizar programas más compactos, que faciliten la reutilización, la modificación y el mantenimiento de los programas. En este capítulo se presentan doce ejercicios sobre herencia, clases abstractas, interfaces y polimorfismo utilizando Java.

- **Ejercicio 4.1. Herencia**
  La herencia de clase es el mecanismo por el cual una clase adquiere o hereda los atributos y métodos de su clase padre y clases antecesoras. El primer ejercicio está orientado a entender este mecanismo y las diferentes instrucciones que permiten su implementación.

- **Ejercicio 4.2. Paquetes y métodos de acceso**
  Un programa en Java constará de numerosas clases que deben estar agrupadas de acuerdo con algún criterio lógico. El mecanismo utilizado en Java para organizar las clases se denomina paquete. A su vez, tanto los paquetes como las jerarquías de clase relacionadas por medio de la herencia tienen sus métodos de acceso. El segundo ejercicio propone el uso de paquetes y la definición de métodos de acceso.

- **Ejercicio 4.3. Invocación implícita de constructor heredado**
  El mecanismo de la herencia permite que un constructor de una clase invoque en forma explícita el constructor de su clase padre. Sin embargo, es necesario conocer algunos detalles adicionales que ocurren cuando el constructor de la clase padre no se invoca en forma explícita. El tercer ejercicio revisa este concepto.

- **Ejercicio 4.4. Polimorfismo**
  Un concepto asociado a la herencia es el polimorfismo, así, la invocación de un método puede tener diferentes comportamientos dependiendo del tipo de objeto por el cual es llamado. El cuarto ejercicio aborda el concepto de polimorfismo.

- **Ejercicio 4.5. Conversión descendente**
  Las conversiones de tipo entre objetos que están relacionados por medio de la herencia poseen algunas restricciones. El quinto ejercicio está orientado a revisar dichas restricciones, particularmente, la conversión descendente en la asignación de objetos relacionados por herencia.

- **Ejercicio 4.6. Métodos polimórficos**
  Un método polimórfico se comporta en forma diferente cuando es invocado desde diferentes objetos. El sexto ejercicio plantea un problema para entender y aplicar métodos polimórficos.

- **Ejercicio 4.7. Clases abstractas**
  En las jerarquías de clases, muchas clases antecesoras no van a ser explícitamente instanciadas porque serían objetos demasiado generales. Es mejor declarar estas clases como abstractas para que no se permita su instanciación. El séptimo ejercicio afronta la definición y aplicación de clases abstractas.

- **Ejercicio 4.8. Métodos abstractos**
  Un concepto adicional a las clases abstractas son los métodos abstractos, los cuales no tienen ningún cuerpo y deben tener obligatoriamente código en la clase hija, a menos que se declare como abstracta nuevamente. En el octavo ejercicio se debe aplicar dicho concepto.

- **Ejercicio 4.9. Operador *instanceof***
  El uso de métodos polimórficos puede llevar a cometer errores porque no se conoce a qué clase pertenece realmente una instancia en un momento dado. Para resolver esta situación, Java ofrece el operador *instanceof*, que determina si un objeto es una instancia de un clase dada. La aplicación de este operador se presenta en el noveno ejercicio.

- **Ejercicio 4.10. Interfaces**
  Un mecanismo adicional de la programación orientada a objetos relacionado con la herencia, muy similar a las clases abstractas, pero con connotaciones semánticas diferentes es el concepto de interfaces. El décimo ejercicio plantea un problema a ser resuelto aplicando dicho concepto.

- **Ejercicio 4.11. Interfaces múltiples**
  El concepto de interfaz es similar a la herencia. Sin embargo, una diferencia fundamental es que una clase puede implementar más de una interfaz; mientras que la herencia debe ser simple y no se permite la herencia múltiple en Java. En el undécimo ejercicio se propone un problema que aplica la implementación de interfaces múltiples.

- **Ejercicio 4.12. Herencia de interfaces**
  Las interfaces, al igual que la herencia, se pueden organizar en una jerarquía donde las interfaces hijas heredan y amplían los métodos de la interfaz padre. El duodécimo ejercicio propone la definición de una jerarquía de interfaces.

Los diagramas UML utilizados en este capítulo son los diagramas de clase y de objetos. Los paquetes incluidos en los diagramas de clase permiten agrupar elementos UML y en los ejercicios se organizarán las clases en un único paquete debido a la pequeña cantidad de clases que contienen las soluciones presentadas. Los diagramas de clase modelan la estructura estática de los programas. Así, los diagramas de clase de los ejercicios presentados, generalmente, estarán conformados por una o varias clases relacionadas por medio de la herencia e incorporan notaciones para identificar tipos de atributos y métodos. Los diagramas de objeto identifican los objetos creados en el método *main* del programa.

Ejercicio 4.1. Herencia

Como ya se ha mencionado, en la programación orientada a objetos, un concepto muy importante y ampliamente utilizado es la herencia. En Java es posible heredar atributos y métodos de una clase a otra.

Cuando se diseña un programa con herencia, se coloca código común en una clase y, luego, se determina a otras clases más específicas que la clase común es su superclase (Schildt, 2018). Cuando una clase hereda de otra, la subclase hereda de la superclase y, consecuentemente, sus atributos y métodos.

Hay dos conceptos importantes cuando una clase hereda de otra clase

- **Subclase (clase hija)**: la clase que hereda de otra clase.
- **Superclase (clase padre)**: la clase de la que se hereda.

Para heredar de una clase se utiliza la palabra clave *extends*, con el siguiente formato:

class ClaseHija extends ClasePadre

Los atributos y métodos heredados se pueden utilizar tal como están, reemplazarlos o complementarlos con nuevos atributos y métodos (Reyes y Stepp, 2016).

Es posible declarar nuevos atributos y métodos en la subclase que no están en la superclase. Se puede reescribir el constructor en la subclase, el cual invoca al constructor de la superclase, ya sea implícitamente o usando la palabra clave *super*.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Definir una jerarquía de herencia con clases y subclases.
- Utilizar la herencia para compartir atributos y métodos entre clases.
- Invocar métodos heredados.
- Redefinir métodos heredados.

Enunciado: clase Cuenta con herencia

Desarrollar un programa que modele una cuenta bancaria que tiene los siguientes atributos, que deben ser de acceso protegido:

- Saldo, de tipo *float*.
- Número de consignaciones con valor inicial cero, de tipo *int*.
- Número de retiros con valor inicial cero, de tipo *int*.
- Tasa anual (porcentaje), de tipo *float*.
- Comisión mensual con valor inicial cero, de tipo *float*.

La clase Cuenta tiene un constructor que inicializa los atributos saldo y tasa anual con valores pasados como parámetros. La clase Cuenta tiene los siguientes métodos:

- Consignar una cantidad de dinero en la cuenta actualizando su saldo.
- Retirar una cantidad de dinero en la cuenta actualizando su saldo. El valor a retirar no debe superar el saldo.
- Calcular el interés mensual de la cuenta y actualiza el saldo correspondiente.
- Extracto mensual: actualiza el saldo restándole la comisión mensual y calculando el interés mensual correspondiente (invoca el método anterior).
- Imprimir: muestra en pantalla los valores de los atributos.

La clase Cuenta tiene dos clases hijas:

- Cuenta de ahorros: posee un atributo para determinar si la cuenta de ahorros está activa (tipo *boolean*). Si el saldo es menor a $10 000, la cuenta está inactiva, en caso contrario se considera activa. Los siguientes métodos se redefinen:
  - Consignar: se puede consignar dinero si la cuenta está activa. Debe invocar al método heredado.
  - Retirar: es posible retirar dinero si la cuenta está activa. Debe invocar al método heredado.
  - Extracto mensual: si el número de retiros es mayor que 4, por cada retiro adicional, se cobra $1000 como comisión mensual. Al generar el extracto, se determina si la cuenta está activa o no con el saldo.

- Un nuevo método imprimir que muestra en pantalla el saldo de la cuenta, la comisión mensual y el número de transacciones realizadas (suma de cantidad de consignaciones y retiros).

- Cuenta corriente: posee un atributo de sobregiro, el cual se inicializa en cero. Se redefinen los siguientes métodos:
  - Retirar: se retira dinero de la cuenta actualizando su saldo. Se puede retirar dinero superior al saldo. El dinero que se debe queda como sobregiro.
  - Consignar: invoca al método heredado. Si hay sobregiro, la cantidad consignada reduce el sobregiro.
  - Extracto mensual: invoca al método heredado.
  - Un nuevo método imprimir que muestra en pantalla el saldo de la cuenta, la comisión mensual, el número de transacciones realizadas (suma de cantidad de consignaciones y retiros) y el valor de sobregiro.

Realizar un método *main* que implemente un objeto Cuenta de ahorros y llame a los métodos correspondientes.

Instrucciones Java del ejercicio

Tabla 4.1. Instrucciones Java del ejercicio 4.1.

Instrucción	Descripción	Formato
super	Si un método sobrescribe un método de su superclase, se puede invocar el método de la clase padre mediante el uso de la palabra clave *super*.	*super(parámetros);*

Solución

Clase: Cuenta

```
/**
 * Esta clase denominada Cuenta modela una cuenta bancaria con los
 * atributos saldo, número de consignaciones, número de retiros, tasa
 * anual de interés y comisión mensual.
 * @version 1.2/2020
 */
public class Cuenta {
```

```
/* Atributo que define el sueldo de una cuenta bancaria
protected float saldo; */
/* Atributo que define el número de consignaciones realizadas en
    una cuenta bancaria */
protected int númeroConsignaciones = 0;
// Atributo que define el número de retiros de una cuenta bancaria
protected int númeroRetiros = 0;
// Atributo que define la tasa anual de intereses de una cuenta bancaria
protected float tasaAnual;
/* Atributo que define la comisión mensual que se cobra a una
    cuenta bancaria */
protected float comisiónMensual = 0;

/**
* Constructor de la clase Cuenta
* @param saldo Parámetro que define el saldo de la cuenta
* @param tasaAnual Parámetro que define la tasa anual de interés de
* la cuenta
*/
public Cuenta(float saldo, float tasaAnual) {
    this.saldo = saldo;
    this.tasaAnual = tasaAnual;
}

/**
* Método que recibe una cantidad de dinero a consignar y actualiza
* el saldo de la cuenta
* @param saldo Parámetro que define la cantidad de dinero a
* consignar en la cuenta
*/
public void consignar(float cantidad) {
    saldo = saldo + cantidad; /* Se actualiza el saldo con la cantidad
        consignada */
    // Se actualiza el número de consignaciones realizadas en la cuenta
    númeroConsignaciones = númeroConsignaciones + 1;
}

/**
* Método que recibe una cantidad de dinero a retirar y actualiza el
* saldo de la cuenta
* @param saldo Parámetro que define la cantidad de dinero a retirar
* de la cuenta
```

```
    */
    public void retirar(float cantidad) {
        float nuevoSaldo = saldo - cantidad;
        /* Si la cantidad a retirar no supera el saldo, el retiro no se puede
            realizar */
        if (nuevoSaldo >= 0) {
            saldo -= cantidad;
            númeroRetiros = númeroRetiros + 1;
        } else {
            System.out.println("La cantida a retirar excede el saldo
                actual.");
        }
    }

    /**
    * Método que calcula interés mensual de la cuenta a partir de la tasa
    * anual aplicada
    */
    public void calcularInterés() {
        float tasaMensual = tasaAnual / 12; /* Convierte la tasa anual en
            mensual */
        float interesMensual = saldo * tasaMensual;
        saldo += interesMensual; /* Actualiza el saldo aplicando el interés
            mensual */
    }

    /**
    * Método que genera un extracto aplicando al saldo actual una
    * comisión y calculando sus intereses
    */
    public void extractoMensual() {
        saldo -= comisiónMensual;
        calcularInterés();
    }
}
```

Clase: CuentaAhorros

```
/**
 * Esta clase denominada CuentaAhorros modela una cuenta de ahorros
 * que es una subclase de Cuenta. Tiene un nuevo atributo: activa.
 * @version 1.2/2020
 */
```

```
public class CuentaAhorros extends Cuenta {
    /* Atributo que identifica si una cuenta está activa; lo está si su saldo
       es superior a 10000 */
    private boolean activa;

    /**
    * Constructor de la clase CuentaAhorros
    * @param saldo Parámetro que define el saldo de la cuenta de ahorros
    * @param tasa Parámetro que define la tasa anual de interés de la
    * cuenta de ahorros
    */
    public CuentaAhorros(float saldo, float tasa) {
        super(saldo, tasa);
        if (saldo < 10000) /* Si el saldo es menor a 10000, la cuenta no
            se activa */
            activa = false;
        else
            activa = true;
    }

    /**
    * Método que recibe una cantidad de dinero a retirar y actualiza el
    * saldo de la cuenta
    * @param saldo Parámetro que define la cantidad a retirar de una
    * cuenta de ahorros
    */
    public void retirar(float cantidad) {
        if (activa) // Si la cuenta está activa, se puede retirar dinero
            super.retirar(cantidad); /* Invoca al método retirar de la clase
                padre */
    }

    /**
    * Método que recibe una cantidad de dinero a consignar y actualiza
    * el saldo de la cuenta
    * @param saldo Parámetro que define la cantidad a consignar en
    * una cuenta de ahorros
    */
    public void consignar(float cantidad) {
        if (activa) // Si la cuenta está activa, se puede consignar dinero
```

```
            super.consignar(cantidad); /* Invoca al método consignar de
                la clase padre */
        }

        /**
        * Método que genera el extracto mensual de una cuenta de ahorros
        */
        public void extractoMensual() {
            /* Si la cantidad de retiros es superior a cuatro, se genera una
                comisión mensual */
            if (númeroRetiros > 4) {
                comisiónMensual += (númeroRetiros - 4) * 1000;
            }
            super.extractoMensual(); // Invoca al método de la clase padre
            /* Si el saldo actualizado de la cuenta es menor a 10000, la
                cuenta no se activa */
            if ( saldo < 10000 )
                activa = false;
        }

        /**
        * Método que muestra en pantalla los datos de una cuenta de
            ahorros
        */
        public void imprimir() {
            System.out.println("Saldo = $ " + saldo);
            System.out.println("Comisión mensual =  $ " +
                comisiónMensual);
            System.out.println("Número de transacciones = " +
                (númeroConsignaciones + númeroRetiros));
            System.out.println();
        }
    }
```

Clase: CuentaCorriente

```
/**
 * Esta clase denominada CuentaCorriente modela una cuenta bancaria
 * que es una subclase de Cuenta. Tiene un nuevo atributo: sobregiro.
 * @version 1.2/2020
 */
```

```
public class CuentaCorriente extends Cuenta {
    /* Atributo que define un sobregiro de la cuenta que surge cuando el
       saldo de la cuenta es negativo */
    float sobregiro;

    /**
    * Constructor de la clase CuentaCorrienta
    * @param saldo Parámetro que define el saldo de la cuenta corriente
    * @param tasa Parámetro que define la tasa anual de interés de la
    * cuenta corriente
    */
    public CuentaCorriente(float saldo, float tasa) {
        super(saldo, tasa); // Invoca al constructor de la clase padre
        sobregiro = 0; // Inicialmente no hay sobregiro
    }

    /**
    * Método que recibe una cantidad de dinero a retirar y actualiza el
    * saldo de la cuenta
    * @param cantidad Parámetro que define la cantidad de dinero a
    * retirar de la cuenta corriente
    */
    public void retirar(float cantidad) {
        float resultado = saldo - cantidad; // Se calcula un saldo temporal
        /* Si el valor a retirar supera el saldo de la cuenta, el valor
           excedente se convierte en sobregiro y el saldo es cero */
        if (resultado < 0) {
            sobregiro = sobregiro - resultado;
            saldo = 0;
        } else {
            super.retirar(cantidad); /* Si no hay sobregiro, se realiza un
                retiro normal */
        }
    }

    /**
    * Método que recibe una cantidad de dinero a consignar y actualiza
    * el saldo de la cuenta
    * @param cantidad Parámetro que define la cantidad de dinero a
    * consignar en la cuenta corriente
    */
    public void consignar(float cantidad) {
        float residuo = sobregiro - cantidad;
```

```
            // Si existe sobregiro la cantidad consignada se resta al sobregiro
            if (sobregiro > 0) {
                if ( residuo > 0) { /* Si el residuo es mayor que cero, se libera
                    el sobregiro */
                    sobregiro = 0;
                    saldo = residuo;
                } else { /* Si el residuo es menor que cero, el saldo es cero y
                    surge un sobregiro */
                    sobregiro = -residuo;
                    saldo = 0;
                }
            } else {
                super.consignar(cantidad); /* Si no hay sobregiro, se realiza
                    una consignación normal */
            }
        }

        /**
        * Método que genera el extracto mensual de la cuenta
        */
        public void extractoMensual() {
            super.extractoMensual(); // Invoca al método de la clase padre
        }

        /**
        * Método que muestra en pantalla los datos de una cuenta corriente
        */
        public void imprimir() {
            System.out.println("Saldo = $ " + saldo);
            System.out.println("Cargo mensual =  $ " + comisiónMensual);
            System.out.println("Número de transacciones = " +
                (númeroConsignaciones + númeroRetiros));
            System.out.println("Valor de sogregiro = $" +
                (númeroConsignaciones + númeroRetiros));
            System.out.println();
        }
    }
```

Clase: PruebaCuenta

```
import java.util.*;
/**
 * Esta clase prueba diferentes acciones realizadas por cuentas bancarias
 * de tipo Cuenta de ahorros y Cuenta corriente
 * @version 1.2/2020
 */
public class PruebaCuenta {

    /**
    * Método main que crea una cuenta de ahorros con un saldo inicial
    * y una tasa de interés solicitados por teclado, a la cual se realiza una
    * consignación y un retiro, y luego se le genera el extracto mensual
    */
    public static void main(String args[]) {
        Scanner input = new Scanner(System.in);
        System.out.println("Cuenta de ahorros");
        System.out.println("Ingrese saldo inicial= $");
        float saldoInicialAhorros = input.nextFloat();
        System.out.print("Ingrese tasa de interés= ");
        float tasaAhorros = input.nextFloat();
        CuentaAhorros cuenta1 = new
            CuentaAhorros(saldoInicialAhorros, tasaAhorros);
        System.out.print("Ingresar cantidad a consignar: $");
        float cantidadDepositar = input.nextFloat();
        cuenta1.consignar(cantidadDepositar);
        System.out.print("Ingresar cantidad a retirar: $");
        float cantidadRetirar = input.nextFloat();
        cuenta1.retirar(cantidadRetirar);
        cuenta1.extractoMensual();
        cuenta1.imprimir();
    }
}
```

Diagrama de clases

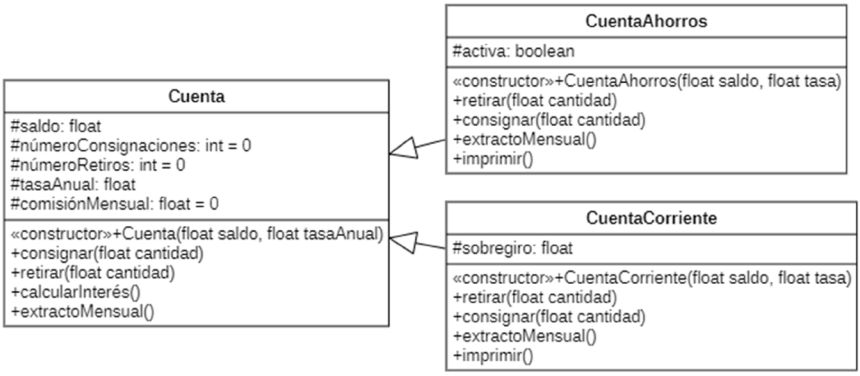


Figura 4.1. Diagrama de clases del ejercicio 4.1.

Explicación del diagrama de clases

El diagrama de clases UML presenta una jerarquía de clases muy básica con una clase padre (o superclase) denominada Cuenta y dos clases hijas (o subclases) denominadas CuentaAhorros y CuentaCorriente, las cuales son dos tipos de cuenta y heredan los atributos y métodos de su clase padre.

La relación de herencia se expresa en UML utilizando una línea continua en cuyo extremo se encuentra un triángulo. La clase adyacente al triángulo es la clase padre y la clase en el otro extremo es la hija. Para que una relación de herencia esté muy bien definida debe existir una relación semántica entre las clases, de tal manera que la relación entre ambas se lea como "es un" o "es un tipo de". En este ejemplo, CuentaAhorros es una cuenta o es un tipo de Cuenta. Por lo tanto, la relación de herencia está bien definida.

Cada clase tiene definida un conjunto de atributos y métodos. Las clases CuentaAhorros y CuentaCorriente heredan dichos atributos y métodos y no es necesario que se dupliquen los atributos y métodos de la clase padre en sus compartimientos.

El uso de atributos protegidos se destaca en el diagrama. En UML, los atributos y métodos protegidos incluyen en su signatura el símbolo (#), el cual significa que pueden ser accedidos por objetos de la misma clase o sus descendientes.

En primer lugar, la clase Cuenta tiene los atributos: saldo, número de consignaciones, número de retiros, tasa anual y comisión mensual con sus

valores iniciales. La clase tiene un constructor y métodos para consignar, retirar, calcular interés y generar extracto mensual.

En segundo lugar, la clase CuentaAhorros tiene un nuevo atributo: activa. Los métodos consignar, retirar y extracto mensual, aunque heredados se incluyen en el tercer compartimiento debido a que se redefinen. Además, cuenta con un nuevo método denominado *imprimir* que muestra sus datos en pantalla.

Finalmente, la clase CuentaCorriente tiene un nuevo atributo: sobregiro. Como en la clase CuentaAhorros, los métodos heredados consignar, retirar y extracto se incluyen en el tercer compartimiento porque se redefinen. Además, cuenta con un nuevo método denominado *imprimir* que muestra sus datos en pantalla.

Diagrama de objetos

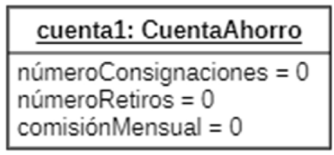


Figura 4.2. Diagrama de objetos del ejercicio 4.1.

Ejecución del programa

```
Cuenta de ahorros
Ingrese saldo inicial= $
100000
Ingrese tasa de interés= 0,10
Ingresar cantidad a consignar: $50000
Ingresar cantidad a retirar: $70000
Saldo = $ 80666.664
Comisión mensual =  $ 0.0
Número de transacciones = 2
```

Figura 4.3. Ejecución del programa del ejercicio 4.1.

Ejercicios propuestos

- Definir una clase Libro para manejar la información asociada a un libro. La información de interés para un libro es: el título, el autor y el precio. Los métodos de interés son:

- Un constructor para crear un objeto libro, con título y autor como parámetros.
- Imprimir en pantalla el título, los autores y el precio del libro.
- Métodos *get* y *set* para cada atributo de un libro.

Se debe extender la clase Libro definiendo las siguientes clases:

- Libros de texto con un nuevo atributo que especifica el curso al cual está asociado el libro.
- Libros de texto de la Universidad Nacional de Colombia: subclase de la clase anterior. Esta subclase tiene un atributo que especifica cuál facultad lo publicó.
- Novelas: pueden ser de diferente tipo, histórica, romántica, policíaca, realista, ciencia ficción o aventuras.
- Para cada una de las clases anteriores se debe definir su constructor y redefinir adecuadamente el método para visualizar del objeto.

Ejercicio 4.2. Paquetes y métodos de acceso

Un paquete en Java se utiliza para agrupar clases relacionadas de acuerdo con algún criterio lógico. Cuando el sistema *software* a desarrollar contiene demasiadas clases es recomendable dividir el sistema en paquetes, de esta forma, la complejidad del sistema se reduce (Booch, Rumbaugh y Jacobson, 2017). Los paquetes en Java se dividen en:

- Paquetes propios de Java (incorporados en la API de Java).
- Paquetes definidos por el usuario.

Para crear un paquete definido por el usuario, se debe utilizar la palabra reservada *package* al inicio de la definición de la clase:

```
package nombrePaquete;
class nombreClase {
 …
}
```

Una subclase hereda todos los atributos públicos y protegidos de su clase padre, sin importar en qué paquete esté la subclase. Si la subclase

está en el mismo paquete que su clase padre, también hereda los atributos privados del padre.

Además de los accesos públicos y privados, Java proporciona dos modificadores de acceso adicionales (Arroyo-Díaz, 2019b, 2007):

- **Paquete**: es el método de acceso por defecto. No se coloca ninguna palabra clave antecediendo al nombre de un atributo o método.
- **Protegida**: los atributos y métodos son visibles para el paquete y todas las subclases. Se utiliza la palabra clave *protected*.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Definir un paquete que contenga una colección de clases relacionadas entre sí por medio de la herencia.
- Definir atributos y métodos con acceso protegido.

Enunciado: clase Inmueble con herencia

Se requiere realizar un programa que modele diferentes tipos de inmuebles. Cada inmueble tiene los siguientes atributos: identificador inmobiliario (tipo entero); área en metros cuadrados (tipo entero) y dirección (tipo *String*).

Los inmuebles para vivienda pueden ser casas o apartamentos. Los inmuebles para vivienda tienen los siguientes atributos: número de habitaciones y número de baños. Las casas pueden ser casas rurales o casas urbanas, su atributo es la cantidad de pisos que poseen. Los atributos de casas rurales son la distancia a la cabecera municipal y la altitud sobre el nivel del mar. Las casas urbanas pueden estar en un conjunto cerrado o ser independientes. A su vez, las casas en conjunto cerrado tienen como atributo el valor de la administración y si incluyen o no áreas comunes como piscinas y campos deportivos. De otro lado, los apartamentos pueden ser apartaestudios o apartamentos familiares. Los apartamentos pagan un valor de administración, mientras que los apartaestudios tienen una sola habitación.

Los locales se clasifican en locales comerciales y oficinas. Los locales tienen como atributo su localización (si es interno o da a la calle). Los locales comerciales tienen un atributo para conocer el centro comercial donde están establecidos. Las oficinas tienen como atributo un valor *boolean* para determinar si son del Gobierno. Cada inmueble tiene un valor de compra. Este depende del área de cada inmueble según la tabla 4.2.

Tabla 4.2. Valor por metro cuadrado según tipo de inmueble

Inmueble	Valor por metro cuadrado
Casa rural	$ 1 500 000
Casa en conjunto cerrado	$ 2 500 000
Casa independiente	$ 3 000 000
Apartaestudio	$ 1 500 000
Apartamento familiar	$ 2 000 000
Local comercial	$ 3 000 000
Oficina	$ 3 500 000

Solución

Clase: Inmueble

```
package Inmuebles;

/**
 * Esta clase denominada Inmueble modela un inmueble que posee
 * como atributos un identificador, un área, una dirección y un precio
 * de venta. Es la clase raíz de una jerarquía de herencia.
 * @version 1.2/2020
 */
public class Inmueble {
    // Atributo para el identificador inmobiliario de un inmueble
    protected int identificadorInmobiliario;
    protected int área; // Atributo que identifica el área de un inmueble
    protected String dirección; /* Atributo que identifica la dirección de
        un inmueble */
    protected double precioVenta; /* Atributo que identifica el precio de
        venta de un inmueble */

    /**
    * Constructor de la clase Inmueble
    * @param identificadorInmobiliario Parámetro que define el
    * identificador de un inmueble
    * @param área Parámetro que define el área de un inmueble
    * @param dirección Parámetro que define la dirección donde se
    * encuentra localizado un inmueble
    */
```

```
    Inmueble(int identificadorInmobiliario, int área, String dirección) {
        this.identificadorInmobiliario = identificadorInmobiliario;
        this.área = área;
        this.dirección = dirección;
    }

    /**
    * Método que a partir del valor del área de un inmueble, calcula su
    * precio de venta
    * @param valorArea El valor unitario por área de un determinado
    * inmueble
    * @return Precio de venta del inmueble
    */
    double calcularPrecioVenta(double valorArea) {
        precioVenta = área * valorArea;
        return precioVenta;
    }

    /**
    * Método que muestra en pantalla los datos de un inmueble
    */
    void imprimir() {
        System.out.println("Identificador inmobiliario = " +
            identificadorInmobiliario);
        System.out.println("Area = " + área);
        System.out.println("Dirección = " + dirección);
        System.out.println("Precio de venta = $" + precioVenta);
    }
}
```

Clase: InmuebleVivienda

```
package Inmuebles;

/**
 * Esta clase denominada InmuebleVivienda modela un inmueble
 * destinado para la vivienda con atributos como el número de
 * habitaciones y el número de baños que posee
 * @version 1.2/2020
 */
public class InmuebleVivienda extends Inmueble {
```

```
    /* Atributo que identifica el número de habitación de un inmueble
       para vivienda */
    protected int númeroHabitaciones;
    /* Atributo que identifica el número de baños de un inmueble para
       vivienda */
    protected int númeroBaños;

    /**
    * Constructor de la clase InmuebleVivienda
    * @param identificadorInmobiliario Parámetro que define el
    * identificador inmobiliario de un inmueble para la vivienda
    * @param área Parámetro que define el área de un inmueble para la
    * vivienda
    * @param dirección Parámetro que define la dirección donde se
    * encuentra localizado un inmueble para la vivienda
    * @param númeroHabitaciones Parámetro que define el número de
    * habitaciones que tiene un inmueble para la vivienda
    * @param númeroBaños Parámetro que define el número de baños
    * que tiene un inmueble para la vivienda
    */
public InmuebleVivienda(int identificadorInmobiliario, int área, String
dirección, int númeroHabitaciones, int númeroBaños) {
        super(identificadorInmobiliario, área, dirección); /* Invoca al
            constructor de la clase padre */
        this.númeroHabitaciones = númeroHabitaciones;
        this.númeroBaños = númeroBaños;
    }

    /**
    * Método que muestra en pantalla los datos de un inmueble para la
    * vivienda
    */
    void imprimir() {
        super.imprimir(); // Invoca al método imprimir de la clase padre
        System.out.println("Número de habitaciones = " +
            númeroHabitaciones);
        System.out.println("Número de baños = " + númeroBaños);
    }
}
```

Clase: Casa

```
package Inmuebles;

/**
 * Esta clase denominada Casa modela un tipo específico de inmueble
 * destinado para la vivienda con atributos como el número de pisos
 * que tiene una casa.
 * @version 1.2/2020
 */
public class Casa extends InmuebleVivienda {
    protected int númeroPisos; /* Atributo que identica el número de
        pisos que tiene una casa */

    /**
    * Constructor de la clase Casa
    * @param identificadorInmobiliario Parámetro que define el
    * identificador inmobiliario de una casa
    * @param área Parámetro que define el área de una casa
    * @param dirección Parámetro que define la dirección donde se
    * encuentra localizada una casa
    * @param númeroHabitaciones Parámetro que define el número de
    * habitaciones que tiene una casa
    * @param númeroBaños Parámetro que define el número de baños
    * que tiene una casa
    * @param númeroPisos Parámetro que define el número de pisos
    * que tiene una casa
    */
    public Casa(int identificadorInmobiliario, int área, String dirección,
        int númeroHabitaciones, int númeroBaños, int númeroPisos) {
        // Invoca al constructor de la clase padre
        super(identificadorInmobiliario, área, dirección,
            númeroHabitaciones, númeroBaños);
        this.númeroPisos = númeroPisos;
    }

    /**
    * Método que muestra en pantalla los datos de una casa
    */
    void imprimir() {
        super.imprimir(); // Invoca al método imprimir de la clase padre
        System.out.println("Número de pisos = " + númeroPisos);
    }
}
```

Clase: Apartamento

```
package Inmuebles;

/**
 * Esta clase denominada Apartamento modela un tipo de inmueble
 * específico destinado para la vivienda.
 * @version 1.2/2020
 */
public class Apartamento extends InmuebleVivienda {

    /**
     * Constructor de la clase Apartamento
     * @param identificadorInmobiliario Parámetro que define el
     * identificador inmobiliario de un apartamento
     * @param área Parámetro que define el área de un apartamento
     * @param dirección Parámetro que define la dirección donde se
     * encuentra localizado un apartamento
     * @param númeroHabitaciones Parámetro que define el número de
     * habitaciones que tiene un apartamento
     * @param númeroBaños Parámetro que define el número de baños
     * que tiene un apartamento
     */
    public Apartamento(int identificadorInmobiliario, int área, String
        dirección, int númeroHabitaciones, int númeroBaños) {
        // Invoca al constructor de la clase padre
        super(identificadorInmobiliario, área, dirección,
            númeroHabitaciones, númeroBaños);
    }

    /**
     * Método que muestra en pantalla los datos de un apartamento
     */
    void imprimir() {
        super.imprimir(); // Invoca al método imprimir de la clase padre
    }
}
```

Clase: CasaRural

```
package Inmuebles;

/**
 * Esta clase denominada CasaRural modela un tipo específico de casa
 * ubicada en el sector rural
 * @version 1.2/2020
 */
public class CasaRural extends Casa {
    // Atributo que identifica el valor por área para una casa rural
    protected static double valorArea = 1500000;
    /* Atributo que identifica la distancia a la que se encuentra la casa
        rural de la cabecera municipal */
    protected int distanciaCabera;
    // Atributo que identifica la altitud a la que se encuentra una casa rural
protected int altitud;

    /**
    * Constructor de la clase CasaRural
    * @param identificadorInmobiliario Parámetro que define el
    * identificador inmobiliario de una casa rural
    * @param área Parámetro que define el área de una casa rural
    * @param dirección Parámetro que define la dirección donde se
    * encuentra localizada una casa rural
    * @param númeroHabitaciones Parámetro que define el número de
    * habitaciones que tiene una casa rural
    * @param númeroBaños Parámetro que define el número de baños
    * que tiene una casa rural
    * @param númeroPisos Parámetro que define el número de pisos
    * que tiene una casa rural
    * @param distanciaCabera Parámetro que define la distancia de la
    * casa rural a la cabecera municipal
    * @param altitud Parámetro que define la altitud sobre el nivel del
    * mar en que se encuentra una casa rural
    */
```

```
    public CasaRural(int identificadorInmobiliario, int área, String
        dirección, int númeroHabitaciones, int númeroBaños, int
        númeroPisos, int distanciaCabera, int altitud) {
        // Invoca al constructor de la clase padre
        super(identificadorInmobiliario, área, dirección,
            númeroHabitaciones, númeroBaños, númeroPisos);
        this.distanciaCabera = distanciaCabera;
        this.altitud = altitud;
    }

    /**
    * Método que muestra en pantalla los datos de una casa rural
    */
    void imprimir() {
        super.imprimir(); // Invoca al método imprimir de la clase padre
        System.out.println("Distancia la cabecera municipal = " +
            númeroHabitaciones + " km.");
        System.out.println("Altitud sobre el nivel del mar = " + altitud +
            " metros.");
        System.out.println();
    }
}
```

Clase: CasaUrbana

```
package Inmuebles;

/**
* Esta clase denominada CasaUrbana modela un tipo específico de casa
* destinada para la vivienda localizada en el sector urbano.
* @version 1.2/2020
*/
public class CasaUrbana extends Casa {
```

```
    /**
     * Constructor de la clase CasaUrbana
     * @param identificadorInmobiliario Parámetro que define el
     * identificador inmobiliario de una casa urbana
     * @param área Parámetro que define el área de una casa urbana
     * @param dirección Parámetro que define la dirección donde se
     * encuentra localizada una casa urbana
     * @param númeroHabitaciones Parámetro que define el número de
     * habitaciones que tiene una casa urbana
     * @param númeroBaños Parámetro que define el número de baños
     * que tiene una casa urbana
     * @param númeroPisos Parámetro que define el número de pisos
     * que tiene una casa urbana
     */
    public CasaUrbana(int identificadorInmobiliario, int área, String
        dirección, int númeroHabitaciones, int númeroBaños, int
        númeroPisos) {
        // Invoca al constructor de la clase padre
        super(identificadorInmobiliario, área, dirección,
            númeroHabitaciones, númeroBaños, númeroPisos);
    }

    /**
     * Método que muestra en pantalla los datos de una casa urbana
     */
    void imprimir() {
        super.imprimir(); // Invoca al método imprimir de la clase padre
    }
}
```

Clase: ApartamentoFamiliar

```
package Inmuebles;

/**
 * Esta clase denominada ApartamentoFamiliar modela un tipo
 * específico de apartamento con atributos como el valor por área y el
 * valor de la administración.
 * @version 1.2/2020
 */
```

```
public class ApartamentoFamiliar extends Apartamento {
    // Atributo que identifica el valor por área de un apartamento familiar
    protected static double valorArea = 2000000;
    /* Atributo que identifica el valor de la administración de un
       apartamento familiar */
    protected int valorAdministración;

    /**
    * Constructor de la clase ApartamentoFamiliar
    * @param identificadorInmobiliario Parámetro que define el
    * identificador inmobiliario de un apartamento familiar
    * @param área Parámetro que define el área de un apartamento familiar
    * @param dirección Parámetro que define la dirección donde se
    * encuentra localizado un apartamento familiar
    * @param númeroHabitaciones Parámetro que define el número de
    * habitaciones que tiene un apartamento familiar
    * @param númeroBaños Parámetro que define el número de baños
    * que tiene un apartamento familiar
    * @param valorAdministración Parámetro que define el valor de la
    * administración de un apartamento familiar
    */
    public ApartamentoFamiliar(int identificadorInmobiliario, int área,
        String dirección, int númeroHabitaciones, int númeroBaños, int
        valor Administración) {
        // Invoca al constructor de la clase padre
        super(identificadorInmobiliario, área, dirección,
            númeroHabitaciones, númeroBaños);
        this.valorAdministración = valorAdministración;
    }

    /**
    * Método que muestra en pantalla los datos de un apartamento familiar
    */
    void imprimir() {
        super.imprimir(); // Invoca al método imprimir de la clase padre
        System.out.println("Valor de la administración = $" +
            valorAdministración);
        System.out.println();
    }
}
```

Clase: Apartaestudio

```
package Inmuebles;

/**
 * Esta clase denominada Apartaestudio modela un tipo específico de
 * apartamento que tiene una sola habitación.
 * @version 1.2/2020
 */
public class Apartaestudio extends Apartamento {
    // Atributo que identifica el valor por área de un apartaestudio
    protected static double valorArea = 1500000;

    /**
    * Constructor de la clase Apartaestudio
    * @param identificadorInmobiliario Parámetro que define el
    * identificador inmobiliario de un apartaestudio
    * @param área Parámetro que define el área de un apartaestudio
    * @param dirección Parámetro que define la dirección donde se
    * encuentra localizado un apartaestudio
    * @param númeroHabitaciones Parámetro que define el número de
    * habitaciones que tiene un apartaestudio
    * @param númeroBaños Parámetro que define el número de baños
    * que tiene un apartaestudio
    */
    public Apartaestudio(int identificadorInmobiliario, int área, String
        dirección,
                int númeroHabitaciones, int númeroBaños) {
        // Invoca al constructor de la clase padre
        // Los apartaestudios tienen una sola habitación y un solo baño
        super(identificadorInmobiliario, área, dirección, 1, 1);
    }

    /**
    * Método que muestra en pantalla los datos de un apartaestudio
    */
    void imprimir() {
        super.imprimir(); // Invoca al método imprimir de la clase padre
        System.out.println();
    }
}
```

Clase: CasaConjuntoCerrado

```
package Inmuebles;

 /**
 * Esta clase denominada CasaConjuntoCerrado modela un tipo
 * específico de casa urbana que se encuentra en un conjunto cerrado
 * con atributos como el valor por área, valor de la administración y
 * valores booleanos para especificar si tiene piscina y campos deportivos.
 * @version 1.2/2020
 */
public class CasaConjuntoCerrado extends CasaUrbana {
    // Atributo que define el valor por área de una casa en conjunto cerrado
    protected static double valorArea = 2500000;
    /* Atributo que define el valor de administración de una casa en
        conjunto cerrado */
    protected int valorAdministración;
    // Atributo que define si una casa en conjunto cerrado tiene piscina
    protected boolean tienePiscina;
    /* Atributo que define si una casa en conjunto cerrado tiene campos
        deportivos */
    protected boolean tieneCamposDeportivos;

    /**
    * Constructor de la clase CasaConjuntoCerrado
    * @param identificadorInmobiliario Parámetro que define el
    * identificador inmobiliario de una casa en conjunto cerrado
    * @param área Parámetro que define el área de una casa en conjunto
    * cerrado
    * @param dirección Parámetro que define la dirección donde se
    * encuentra localizada una casa en conjunto cerrado
    * @param númeroHabitaciones Parámetro que define el número de
    * habitaciones que tiene una casa en conjunto cerrado
    * @param númeroBaños Parámetro que define el número de baños
    * que tiene una casa en conjunto cerrado
    * @param númeroPisos Parámetro que define el número de pisos
    * que tiene una casa en conjunto cerrado
    * @param valorAdministración Parámetro que define el valor de
    * administración para una casa en conjunto cerrado
    * @param tienePiscina Parámetro que define si una casa en conjunto
    * cerrado tiene o no piscina
    * @param tieneCamposDeportivos Parámetro que define si una casa
    * en conjunto cerrado tiene o no campos deportivos
    */
```

```
    public CasaConjuntoCerrado(int identificadorInmobiliario, int área,
        String dirección, int númeroHabitaciones, int númeroBaños,
        int númeroPisos, int valorAdministración, boolean tienePiscina,
        boolean tieneCamposDeportivos) {
        // Invoca al constructor de la clase padre
        super(identificadorInmobiliario, área, dirección,
            númeroHabitaciones, númeroBaños, númeroPisos);
        this.valorAdministración = valorAdministración;
        this.tienePiscina = tienePiscina;
        this.tieneCamposDeportivos = tieneCamposDeportivos;
    }
```

```
    /**
    * Método que muestra en pantalla los datos de una casa en conjunto
    * cerrado
    */
    void imprimir() {
        super.imprimir(); // Invoca al método imprimir de la clase padre
        System.out.println("Valor de la administración = " +
            valorAdministración);
        System.out.println("Tiene piscina? = " + tienePiscina);
        System.out.println("Tiene campos deportivos? = " +
            tieneCamposDeportivos);
        System.out.println();
    }
}
```

Clase: CasaIndependiente

```
package Inmuebles;
```

```
/**
* Esta clase denominada CasaIndependiente modela un tipo específico
* de casa urbana que no está en conjunto cerrado y es completamente
* independiente de otras casas. Tiene un atributo estático para
* especificar un valor del área del inmueble.
* @version 1.2/2020
*/
public class CasaIndependiente extends CasaUrbana {
    // Atributo que identifica el valor por área de una casa independiente
    protected static double valorArea = 3000000;
```

```
    /**
     * Constructor de la clase CasaIndependiente
     * @param identificadorInmobiliario Parámetro que define el
     * identificador inmobiliario de una casa independiente
     * @param área Parámetro que define el área de una casa independiente
     * @param dirección Parámetro que define la dirección donde se
     * encuentra localizada una casa independiente
     * @param númeroHabitaciones Parámetro que define el número de
     * habitaciones que tiene una casa independiente
     * @param númeroBaños Parámetro que define el número de baños
     * que tiene una casa independiente
     * @param númeroPisos Parámetro que define el número de pisos
     * que tiene una casa independiente
     */
    public CasaIndependiente(int identificadorInmobiliario, int área,
        String dirección, int númeroHabitaciones, int númeroBaños, int
        númeroPisos) {
        // Invoca al constructor de la clase padre
        super(identificadorInmobiliario, área, dirección,
            númeroHabitaciones, númeroBaños, númeroPisos);
    }

    /**
     * Método que muestra en pantalla los datos de una casa independiente
     */
    void imprimir() {
        super.imprimir(); // Invoca al método imprimir de la clase padre
        System.out.println();
    }
}
```

Clase: Local

```
package Inmuebles;

/**
 * Esta clase denominada Local modela un tipo específico de inmueble
 * que no está destinado para la vivienda que tiene como atributos un
 * tipo que especifica si es un local interno o que da a la calle.
 * @version 1.2/2020
 */
public class Local extends Inmueble {
```

```
    enum tipo {INTERNO,CALLE}; /* Tipo de inmueble especificado
        como un valor enumerado */
    protected tipo tipoLocal; /* Atributo que identifica el tipo de
        inmueble */

    /**
    * Constructor de la clase Local
    * @param identificadorInmobiliario Parámetro que define el
    * identificador inmobiliario de un local
    * @param área Parámetro que define el área de un local
    * @param dirección Parámetro que define la dirección donde se
    * encuentra localizado un local
    * @param tipoLocal Parámetro que define el tipo de local (interno o
    * que da a la calle)
    */
    public Local(int identificadorInmobiliario, int área, String dirección,
        tipo tipoLocal) {
        // Invoca al constructor de la clase padre
        super(identificadorInmobiliario, área, dirección);
        this.tipoLocal = tipoLocal;
    }

    /**
    * Método que muestra en pantalla los datos de un local
    */
    void imprimir() {
        super.imprimir(); // Invoca al método imprimir de la clase padre
        System.out.println("Tipo de local = " + tipoLocal);
    }
}
```

Clase: LocalComercial

```
package Inmuebles;

/**
 * Esta clase denominada LocalComercial modela un tipo específico de
 * Local destinado para un uso comercial con atributos como el valor
 * por área y el centro comercial al cual pertenece.
 * @version 1.2/2020
 */
public class LocalComercial extends Local {
```

```
    // Atributo que identifica el valor por área de un local comercial
    protected static double valorArea = 3000000;
    /* Atributo que identifica el centro comercial donde está ubicado el
       local comercial */
    protected String centroComercial;

    /**
    * Constructor de la clase LocalComercial
    * @param identificadorInmobiliario Parámetro que define el
    * identificador inmobiliario de un local comercial
    * @param área Parámetro que define el área de un local comercial
    * @param dirección Parámetro que define la dirección donde se
    * encuentra localizado un local comercial
    * @param tipoLocal Parámetro que define el tipo de local comercial
    * (interno o que da a la calle)
    * @param centroComercial Parámetro que define el nombre del
    * centro comercial donde está ubicado el local comercial
    */
    public LocalComercial(int identificadorInmobiliario, int área, String
        dirección, tipo tipoLocal, String centroComercial) {
        // Invoca al constructor de la clase padre
        super(identificadorInmobiliario, área, dirección, tipoLocal);
        this.centroComercial = centroComercial;
    }

    /**
    * Método que muestra en pantalla los datos de un local comercial
    */
    void imprimir() {
        super.imprimir(); // Invoca al método imprimir de la clase padre
        System.out.println("Centro comercial = " + centroComercial);
        System.out.println();
    }
}
```

Clase: Oficina

```
package Inmuebles;

/**
 * Esta clase denominada Oficina modela un tipo específico de local
 * con atributos como el valor por área y un valor booleano para
 * determinar si pertenece o no al gobierno.
 * @version 1.2/2020
 */
public class Oficina extends Local {
    // Atributo que identifica el valor por área de una oficina
    protected static double valorArea = 3500000;
    // Atributo que identifica si una oficina pertenece o no al gobierno
    protected boolean esGobierno;

    /**
    * Constructor de la clase Oficina
    * @param identificadorInmobiliario Parámetro que define el
    * identificador inmobiliario de una oficina
    * @param área Parámetro que define el área de una oficina
    * @param dirección Parámetro que define la dirección donde se
    * encuentra localizada una oficina
    * @param tipoLocal Parámetro que define el tipo de una oficina
    * (interna o que da a la calle)
    * @param esGobierno Parámetro que define un valor booleano para
    * determinar si la oficina es del gobierno o no
    */
    public Oficina(int identificadorInmobiliario, int área, String
        dirección, tipo tipoLocal, boolean esGobierno) {
        // Invoca al constructor de la clase padre
        super(identificadorInmobiliario, área, dirección, tipoLocal);
        this.esGobierno = esGobierno;
    }

    /**
    * Método que muestra en pantalla los datos de una oficina
    */
```

```
    void imprimir() {
        super.imprimir(); // Invoca al método imprimir de la clase padre
        System.out.println("Es oficina gubernamental = " + esGobierno);
        System.out.println();
    }
}
```

Clase: Prueba

```
package Inmuebles;

/**
 * Esta clase prueba diferentes inmuebles, se calcula su precio de
 * acuerdo al área y se muestran sus datos en pantalla
 * @version 1.2/2020
 */
public class Prueba {

    /**
     * Método main que crea dos inmuebles, calcula su precio de
     * acuerdo al área y se muestran sus datos en pantalla
     */
    public static void main(String args[]) {
        ApartamentoFamiliar apto1 = new
            ApartamentoFamiliar(103067,120,
                "Avenida Santander 45-45",3,2,200000);
        System.out.println("Datos apartamento");
        apto1.calcularPrecioVenta(apto1.valorArea);
        apto1.imprimir();

        System.out.println("Datos apartamento");
        Apartaestudio aptestudio1 = new
            Apartaestudio(12354,50,"Avenida Caracas 30-15",1,1);
        aptestudio1.calcularPrecioVenta(aptestudio1.valorArea);
        aptestudio1.imprimir();
    }
}
```

Diagrama de clases

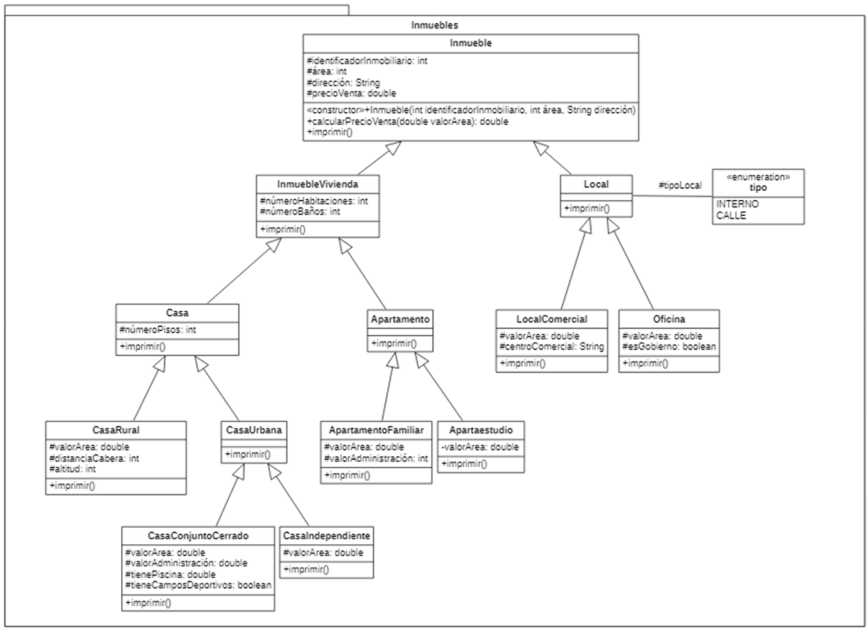


Figura 4.4. Diagrama de clases del ejercicio 4.2.

Explicación del diagrama de clases

El diagrama de clases UML presenta una jerarquía de clases donde la clase raíz es Inmueble. La clase Inmueble tiene dos clases hijas: InmuebleVivienda y Local. La clase InmuebleVivienda tiene dos clases hijas: Casa y Apartamento. A su vez, la clase Casa tiene dos clases hijas: CaraRural y CasaUrbana y la clase CasaRural no tiene clases hijas. La clase CasaUrbana tiene dos clases hijas: CasaConjuntoCerrado y CasaIndependiente y la clase Apartamento tiene dos clases hijas: ApartamentoFamiliar y Apartaestudio. Estas dos últimas clases no tienen clases hijas. Finalmente, la clase Local tiene dos clases hijas: LocalComercial y Oficina, las cuales no tienen clases hijas.

La relación de herencia se expresa en UML utilizando una línea continua en cuyo extremo se encuentra un triángulo. La clase adyacente al triángulo es la clase padre y la clase en el otro extremo es la hija.

Todos los atributos de las clases son protegidos y se identifican con el símbolo (#) en la definición del atributo, lo cual significa que pueden ser accedidos por objetos de la misma clase o de sus clases descendientes.

También se puede observar que la jerarquía de clases está incluida dentro de un paquete denominado "Inmuebles", el cual se representa en UML como una carpeta.

Diagrama de objetos

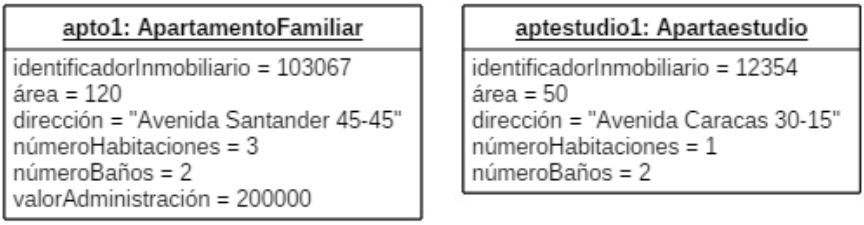


Figura 4.5. Diagrama de objetos del ejercicio 4.2.

Ejecución del programa

```
Datos apartamento
Identificador inmobiliario = 103067
Area = 120
Dirección = Avenida Santander 45-45
Precio de venta = $2.4E8
Número de habitaciones = 3
Número de baños = 2
Valor de la administración = $200000

Datos apartamento
Identificador inmobiliario = 12354
Area = 50
Dirección = Avenida Caracas 30-15
Precio de venta = $7.5E7
Número de habitaciones = 1
Número de baños = 1
```

Figura 4.6. Ejecución del programa del ejercicio 4.2.

Ejercicios propuestos

- Definir un paquete denominado TiendaMascotas, el cual contiene una jerarquía de animales. Mascota es la clase raíz. Los animales que tiene la tienda pueden ser perros y gatos. Los perros se categorizan en perros grandes, medianos y pequeños. A su vez, los perros pequeños pueden ser de las siguientes razas: caniche, *yorkshire terrier*, *schnauzer* y chihuahua. Los perros medianos: collie, dálmata, *bulldog*, galgo y sabueso. Por último, las razas de perros grandes: pastor alemán, *doberman* y *rotweiller.* Los gatos se categorizan en gatos sin pelo, gatos de pelo largo y gatos de pelo corto. Las razas de gatos sin pelo pueden ser: esfinge, elfo y *donskoy*. Los gatos de pelo largo: angora, himalayo, balinés y somalí. Finalmente, los gatos de pelo corto: azul ruso, británico, *manx* y *devon rex.*

  Todos estos animales tienen un nombre, una edad y un color. Los perros tienen atributos adicionales como peso y un atributo booleano para determinar si muerde o no. Todos los perros tienen un método estático denominado "sonido" que imprime en pantalla "Los perros ladran". Los gatos tienen atributos adicionales como: altura de salto y longitud de salto. Todos los gatos tienen un método estático denominado "sonido" que imprime en pantalla "Los gatos maúllan y ronronean".

Ejercicio 4.3. Invocación implícita de constructor heredado

En una relación de herencia en Java, cuando en una clase hija se define un constructor que no invoca al constructor de la clase padre, dicho constructor es invocado en forma implícita (Schildt y Skrien, 2013).

Por lo tanto, si se tienen dos clases relacionadas por medio de la herencia:

```
class ClassA{
        ClassA {…}  //Constructor de la clase A
}
class ClassB extends ClassA {
        ClassB {…} //Constructor de la clase B
}
```

Si el constructor de la clase B no invoca al constructor de su clase padre (clase A) utilizando la referencia *super*, el constructor de la clase A se invocará de todos modos.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Comprender la herencia de constructores.
- Comprender la invocación implícita de constructores heredados.

Enunciado: clase padre DispositivoInformático y clase hija Tableta

¿Cuál es el resultado de la ejecución del siguiente programa (adaptado de Jain y Mangal, 2019)?

Solución

Clase: DispositivoInformático

```
package Informática;
/**
 * Esta clase denominada DispositivoInformático modela un equipo
 * informático y cuenta con un único atributo: marca, el cual
 * inicialmente tiene el valor "Acer".
 * @version 1.2/2020
 */
class DispositivoInformático {
    String marca = "Acer"; /* Atributo que identifica la marca del
        dispositivo informático */
    /**
    * Constructor de la clase DispositivoInformático que imprime en
    * pantalla la marca del dispositivo informático
    */
    DispositivoInformático() {
        System.out.println("Marca = " + marca);
    }
}
```

Clase: Tableta

```
package Informática;

 /**
 * Esta clase denominada Tableta modela un tipo específico de equipo
 * informático
 * @version 1.2/2020
 */
class Tableta extends DispositivoInformático {

    /**
    * Constructor de la clase Tableta que imprime en pantalla la marca
    * de la tableta
    */
    Tableta(String marca) {
        System.out.println("Marca = " + marca); /* ¿Qué imprimirá al ser
            ejecutado el constructor? */
    }
}
```

Clase: Prueba

```
package Informática;

/**
 * Esta clase prueba las clase Tableta instanciando un objeto de esta clase
 * @version 1.2/2020
 */
class Prueba {

    /**
    * Método main que crea una tableta con el parámetro "Dell". ¿Qué
    * se imprimirá en pantalla cuando se invoque al constructor de la clase?
    * Tener en cuenta que Tableta es una subclase de DispositivoInformático
    */
    public static void main(String[] args) {
        Tableta tableta = new Tableta("Dell");
    }
}
```

Diagrama de clases

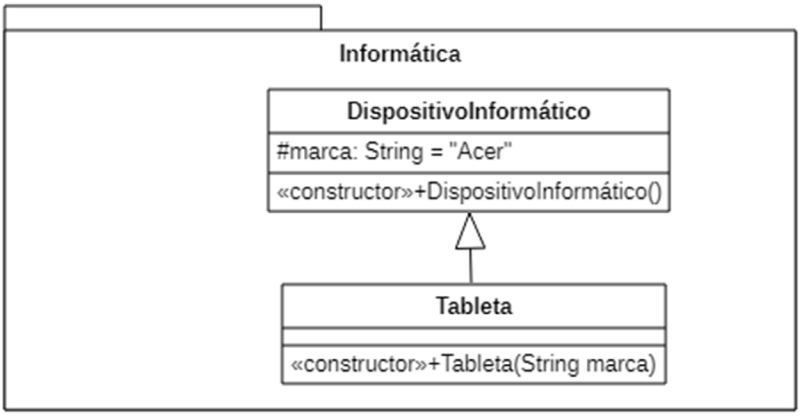


Figura 4.7. Diagrama de clases del ejercicio 4.3.

Explicación del diagrama de clases

Se ha definido un paquete denominado Informática que tiene dos clases relacionadas por la herencia. La clase DispositivoInformático es la clase padre y Tableta, la hija. La clase DispositivoInformático define un atributo protegido (identificado con el símbolo #) denominado marca con un valor inicial "Acer" y un constructor. La clase hija Tableta en su constructor redefine el valor del atributo heredado, aunque esto no se observa en el diagrama UML.

Diagrama de objetos

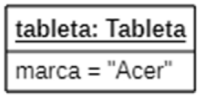


Figura 4.8. Diagrama de objetos del ejercicio 4.3.

Ejecución del programa

```
Marca = Acer
Marca = Dell
```

Figura 4.9. Ejecución del programa del ejercicio 4.3.

El constructor de la clase hija Tableta imprime en pantalla la marca que pasó como parámetro en el constructor, en este caso "Dell". Sin embargo, el constructor de la clase padre se invoca en forma implícita en primer lugar. Por ello, primero imprime marca = "Acer" y luego las instrucciones del constructor de la clase hija: marca = "Dell".

Ejercicios propuestos

- Desarrollar el código del siguiente diagrama en la figura 4.10 .

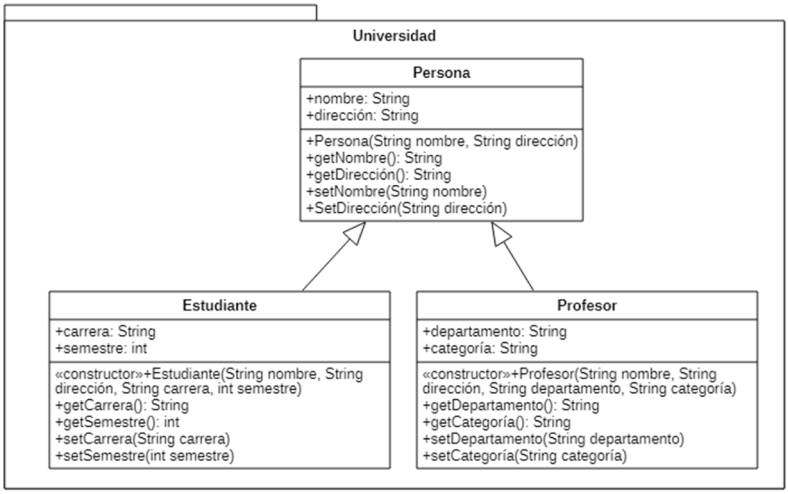


Figura 4.10. Diagrama de clases del ejercicio propuesto.

Ejercicio 4.4. Polimorfismo

El polimorfismo es la capacidad de un objeto para adoptar muchas formas. El uso más común del polimorfismo ocurre cuando se utiliza una referencia a una clase padre para referirse a un objeto de una clase hija (Vozmediano, 2017).

La única forma posible de acceder a un objeto es a través de una variable de referencia. Una variable de referencia puede ser de un solo tipo. Una vez declarado, el tipo de una variable de referencia no cambia. La variable de referencia se puede reasignar a otros objetos siempre que no se declare

final. El tipo de la variable de referencia determinará los métodos que se pueden invocar en el objeto.

Una variable de referencia puede referirse a cualquier objeto de su tipo declarado o cualquier subtipo de su tipo declarado, es decir, que una variable de un cierto tipo puede cambiar su contenido en tiempo de ejecución, siempre y cuando su contenido real sea un objeto de la clase hija o descendiente del tipo de la variable declarada inicialmente.

Objetivo de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para comprender el concepto de polimorfismo relacionado con la herencia.

Enunciado: clase padre Profesor y clase hija ProfesorTitular

¿Cuál es el resultado de la ejecución del siguiente programa (adaptado de Bajracharya, 2019)?

Clase: Profesor

```
package Profesores;
/**
 * Esta clase denominada Profesor es una superclase que representa un
 * profesor genérico.
 * @version 1.2/2020
 */
public class Profesor {
    /**
     * Método que imprime en pantalla un texto específico identificando
     * que el objeto es un Profesor
     */
    protected void imprimir() {
        System.out.println("Es un profesor.");
    }
}
```

Clase: ProfesorTitular

```
package Profesores;

/**
 * Esta clase denominada ProfesorTitular es una subclase de Profesor
 * @version 1.2/2020
 */
public class ProfesorTitular extends Profesor {

    /**
     * Método que sobreescribe el método imprimir heredado de la clase
     * padre Profesor
     */
    protected void imprimir() {
        System.out.println("Es un profesor titular.");
    }
}
```

Clase: Prueba

```
package Profesores;

 /**
 * Esta clase prueba las clase Profesor y ProfesorTitular utilizando el
 * polimorfismo
 * @version 1.2/2020
 */
public class Prueba {

    /**
     * Método main que crea un Profesor pero instanciando la clase
     * ProfesorTitular. ¿Qué se imprimirá en pantalla?
     */
    public static void main(String[] args)  {
        Profesor profesor1 = new ProfesorTitular();
        profesor1.imprimir();
    }
}
```

Diagrama de clases

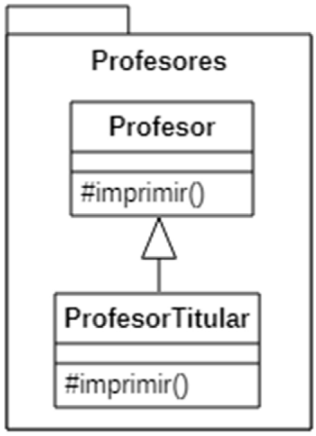


Figura 4.11. Diagrama de clases del ejercicio 4.4.

Explicación del diagrama de clases

Se ha definido un paquete denominado Profesores que tiene dos clases relacionadas por la herencia. La clase Profesor es la clase padre y ProfesorTitular es la hija. La clase Profesor define un método protegido (identificado con el símbolo #) denominado imprimir. La clase hija ProfesorTitular redefine el método imprimir, por ello, se coloca nuevamente en el tercer compartimiento de la clase hija.

Diagrama de objetos

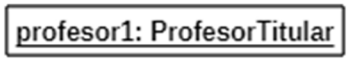


Figura 4.12. Diagrama de objetos del ejercicio 4.4.

Ejecución del programa

```
Es un profesor titular.
```

Figura 4.13. Ejecución del programa del ejercicio 4.4.

En el método *main* de la clase Prueba se declara una variable de tipo Profesor que cuando se instancia tiene almacenado un ProfesorTitular. Cuando se invoca el método *imprimir*, el polimorfismo detecta que la variable realmente contiene un ProfesorTitular y, por ello, ejecutará el método imprimir de dicha clase y no el método heredado de la clase padre Profesor.

Ejercicios propuestos

- ¿Cuál es el resultado de la ejecución del siguiente programa basado en el ejercicio anterior?

```
public class Prueba {
    public static void main(String[] args)  {
        Profesor profesor1 = new ProfesorTitular();
        Profesor profesor2 = (Profesor) profesor1;
        profesor2.imprimir();
    }
}
```

Ejercicio 4.5. Conversión descendente

En una relación de herencia, una instancia de la clase padre puede contener una referencia a otro objeto de la clase padre o sus descendientes. Esta compatibilidad se denomina conversión ascendente (Samoylov, 2019).

También se puede realizar la asignación de una variable de la clase padre a una variable de la clase hija, siempre que la variable de la clase padre guarde una referencia a un objeto de la clase hija o sus descendientes. Esta conversión hay que hacerla en forma explícita mediante el proceso de *casting*. A esta compatibilidad se le denomina conversión descendente (Samoylov, 2019).

El proceso de *casting* tiene el siguiente formato:

ClasePadre variable1 = (ClaseHija) variable;

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Comprender el cambio de contenidos de un objeto que referencia a otros objetos relacionados por medio de la herencia.
- Comprender el concepto de conversión descendente.

Enunciado: clase padre Profesor y clase hija ProfesorTitular

¿Cuál es el resultado de la ejecución del siguiente programa (adaptado de Kumar, 2019)?

Solución

Clase: Profesor

```
package Profesores;
/**
 * Esta clase denominada Profesor es una superclase que representa un
 * profesor genérico
 * @version 1.2/2020
 */
public class Profesor {
    /**
     * Método que imprime en pantalla un texto específico identificando
     * que el objeto es un Profesor
     */
    protected void imprimir() {
        System.out.println("Es un profesor.");
    }
}
```

Clase: ProfesorTitular

```
package Profesores;
/**
 * Esta clase denominada ProfesorTitular es una subclase de Profesor
 * @version 1.2/2020
 */
```

```
public class ProfesorTitular extends Profesor {

    /**
     * Método que sobreescribe el método imprimir heredado de la clase
     * padre Profesor
     */
    protected void imprimir() {
        System.out.println("Es un profesor titular.");
    }
}
```

Clase: Prueba2

```
package Profesores;

 /**
 * Esta clase prueba las clases Profesor y ProfesorTitular utilizando la
 * conversión descendente
 * @version 1.2/2020
 */
public class Prueba2 {

    /**
     * Método main que crea un ProfesorTitular pero instanciando la
     * clase Profesor. ¿Qué se imprimirá en pantalla? ¿Compilará el
     * programa?
     */
    public static void main(String[] args)  {
        ProfesorTitular objeto = new Profesor();
        objeto.imprimir();
    }
}
```

Diagrama de clases

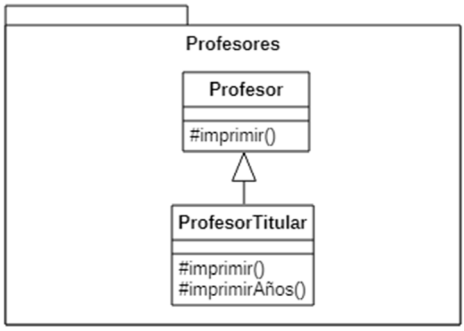


Figura 4.14. Diagrama de clases del ejercicio 4.5.

Explicación del diagrama de clases

El diagrama de clases UML es el mismo del ejercicio anterior. Se ha definido un paquete denominado Profesores que tiene dos clases relacionadas por la herencia. La clase Profesor es la clase padre y ProfesorTitular es la clase hija. La clase Profesor define un método protegido (identificado con el símbolo #) denominado *imprimir*. La clase hija ProfesorTitular redefine el método *imprimir*, por ello, se coloca nuevamente en el tercer compartimiento de la clase hija.

Diagrama de objetos

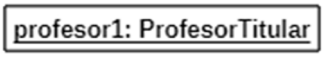


Figura 4.15. Diagrama de objetos del ejercicio 4.5.

Ejecución del programa

El programa no compila. Este resultado se debe a que en el método *main* de la clase Prueba se está realizando una conversión ascendente: en un objeto declarado como perteneciente a la clase hija se está instanciando y asignando un objeto de la clase padre, es decir, a una variable declarada ProfesorTitular (denominada objeto) se le está asignado un Profesor. Este tipo de asignaciones no se permiten. Por lo tanto, se genera un error de compilación así: "tipos incompatibles, no se puede convertir ProfesorTitular a Profesor", ya que no todos los profesores son profesores titulares.

Ejercicios propuestos

- ¿Cuál es el resultado de la ejecución del siguiente programa basado en el ejercicio anterior?:

```
package Profesores;

public class Prueba {
    public static void main(String[] args)  {
        Profesor profesor1 = new ProfesorTitular();
        ProfesorTitular profesor2 = (Profesor) profesor1;
        profesor2.imprimir();
    }
}
```

Ejercicio 4.6. Métodos polimórficos

Los métodos polimórficos se caracterizan porque están definidos en clases relacionadas entre sí por medio de la herencia (Arroyo-Díaz, 2019c). Los métodos se heredan de una clase padre a otra hija.

En las clases hijas, dichos métodos a veces han sido redefinidos y, por lo tanto, son distintos entre sí y obtienen resultados diferentes al ser ejecutados. Es requisito obligatorio que los métodos tengan el mismo nombre.

Objetivo de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para invocar métodos polimórficos en clases relacionadas por medio de la herencia.

Enunciado: clase padre Profesor y clase hija ProfesorTitular

¿Cuál es el resultado de la ejecución del siguiente programa (adaptado de Kumar, 2019)?

Solución

Clase: Profesor

```
package Profesores;

/**
 * Esta clase denominada Profesor es una superclase que representa un
 * profesor genérico
 * @version 1.2/2020
 */
public class Profesor {

    /**
     * Método que imprime en pantalla un texto específico identificando
     * que el objeto es un Profesor
     */
    protected void imprimir() {
        System.out.println("Es un profesor.");
    }
}
```

Clase: ProfesorTitular

```
package Profesores;

/**
 * Esta clase denominada ProfesorTitular es una subclase de Profesor
 * @version 1.2/2020
 */
public class ProfesorTitular extends Profesor {
    /* Atributo que identifica la cantidad de años que el profesor ha sido
       titular */
    int años = 0;

    /**
     * Método que sobreescribe el método imprimir heredado de la clase
     * padre Profesor
     */
    protected void imprimir() {
        System.out.println("Es un profesor titular.");
    }
}
```

```
    /**
     * Método que imprime en pantalla la cantidad de años que tiene un
     * profesor siendo titular
     */
    protected void imprimirAños() {
        System.out.println("Años = " + años);
    }
}
```

Clase: Prueba3

```
package Profesores;

/**
 * Esta clase prueba las clases Profesor y ProfesorTitular utilizando
 * métodos polimórficos
 * @version 1.2/2020
 */
public class Prueba3 {

    /**
     * Método main que crea un ProfesorTitular pero en un objeto tipo
     * Profesor.
     * ¿Qué se imprimirá en pantalla? ¿Compilará el programa?
     */
    public static void main(String[] args)  {
        Profesor profesor1 = new ProfesorTitular();
        profesor1.imprimirAños();
    }
}
```

Diagrama de clases

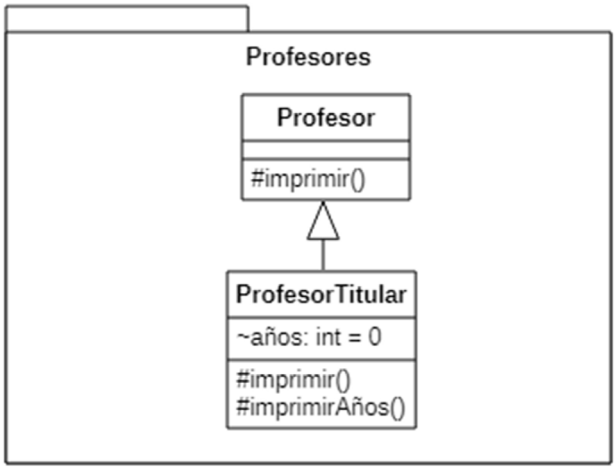


Figura 4.16. Diagrama de clases del ejercicio 4.6.

Explicación del diagrama de clases

El diagrama de clases UML es muy similar al ejemplo anterior. Se ha definido un paquete denominado Profesores que tiene dos clases relacionadas por la herencia. La clase Profesor es la clase padre y ProfesorTitular es la clase hija. El único cambio es que la clase ProfesorTitular tiene un atributo denominado años de tipo entero (con un valor inicial cero) y un nuevo método protegido (identificado con el símbolo #): imprimirAños. La clase Profesor define un método protegido denominado *imprimir*. La clase hija ProfesorTitular redefine el método *imprimir*, por ello, se coloca nuevamente en el tercer compartimiento de la clase hija.

Diagrama de objetos

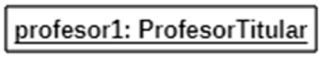


Figura 4.17. Diagrama de objetos del ejercicio 4.6.

Ejecución del programa

El programa no compila. El programa presenta este comportamiento debido a que el método *main* en la clase Prueba ha definido una conversión ascendente que es válida en Java: en un objeto declarado de un tipo de la clase padre se puede instanciar y asignar un objeto de la clase hija. Por la tanto, la primera instrucción compilará ya que al objeto de tipo Profesor (profesor1) se le asigna a un objeto de tipo ProfesorTitular. Sin embargo, la siguiente instrucción es la que hace que el programa no compile, ya que la variable profesor1 es un Profesor y se está invocando el método imprimirAños, el cual no está incluido en la clase Profesor sino en su clase hija ProfesorTitular.

Con esto se muestra que el polimorfismo se aplica correctamente de clases hijas a clases padres, pero es obligatorio que ambas clases tengan los mismos métodos.

Ejercicios propuestos

- ¿Cuál es el resultado de la ejecución del siguiente programa basado en el ejercicio anterior?

```
import java.util.*;

public class Prueba {
    Vector profesores;

    public static void main(String[] args) {
        Prueba prueba = new Prueba();
        prueba.profesores = new Vector();
        Profesor profesor1 = new Profesor();
        ProfesorTitular profesor2 = new ProfesorTitular();
        prueba.profesores.add(profesor1);
        prueba.profesores.add(profesor2);
        for(int i = 0; i < prueba.profesores.size(); i++) {
            Profesor p = (Profesor) prueba.profesores.elementAt(i);
            p.imprimir();
        }
    }
}
```

Ejercicio 4.7. Clases abstractas

La abstracción es el proceso para ocultar los detalles de implementación de una clase y mostrar solo la funcionalidad al usuario (Rumpe, 2016). La abstracción permite concentrarse en qué hace un objeto en lugar de cómo lo hace.

Una clase abstracta no permite que se realicen instancias de dicha clase. Las clases abstractas puede tener métodos abstractos y no abstractos; constructores y métodos estáticos, y métodos finales que obligarán a la subclase a no cambiar el cuerpo del método (Cadenhead, 2017).

Una clase abstracta se define anteponiendo la palabra reservada *abstract* en la definición de la clase:

```
abstract Class nombreClase {
        bloque de instrucciones
}
```

Objetivo de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para definir una jerarquía de herencia con clases abstractas.

Enunciado: jerarquía de clases de animales

Se tiene una jerarquía taxonómica con los siguientes animales:

- Animal es la clase raíz con los atributos: sonidos, alimentos, hábitat y nombre científico (todos de tipo *String*). Esta clase tiene los siguientes métodos abstractos:
  - public abstract *String* getNombreCientífico()
  - public abstract *String* getSonido()
  - public abstract *String* getAlimentos()
  - public abstract *String* getHábitat()
- Los cánidos y los felinos son subclases de Animal.
- Los perros son cánidos, su sonido es el ladrido, su alimentación es carnívora, su hábitat es doméstico y su nombre científico es *Canis lupus familiaris*.

- Los lobos son cánidos, su sonido es el aullido, su alimentación es carnívora, su hábitat es el bosque y su nombre científico es *Canis lupus*.
- Los leones son felinos, su sonido es el rugido, su alimentación es carnívora, su hábitat es la pradera y su nombre científico es *Panthera leo*.
- Los gatos son felinos, su sonido es el maullido, su alimentación son los ratones, su hábitat es doméstico y su nombre científico es *Felis silvestris catus*.

Además, se requiere en una clase de prueba para desarrollar un método *main* que genere un *array* de animales y la pantalla debe mostrar los valores de sus atributos.

Solución

Clase: Animal

```
package Animales;
/**
 * Esta clase abstracta denominada Animal modela un animal genérico
 * que cuenta con atributos como un sonido, alimentos que consume,
 * un hábitat y un nombre científico.
 * @version 1.2/2020
 */
public abstract class Animal {
    protected String sonido; /* Atributo que identifica el sonido emitido
        por un animal */
    protected String alimentos; /* Atributo que identifica los alimentos
        que consume un animal */
    protected String hábitat; /* Atributo que identifica el hábitat de un
        animal */
    protected String nombreCientífico; /* Atributo que identifica el
        nombre científico de un animal */

    /**
    * Método abstracto que permite obtener el nombre científico del animal
    * @return El nombre científico del animal
    */
```

```
    public abstract String getNombreCientífico();
    /**
     * Método abstracto que permite obtener el sonido producido por el
     * animal
     * @return El sonido producido por el animal
     */
    public abstract String getSonido();
    /**
     * Método abstracto que permite obtener los alimentos que consume
     * un animal
     * @return Los alimentos que consume el animal
     */
    public abstract String getAlimentos();
    /**
     * Método abstracto que permite obtener el hábitat de un animal
     * @return El hábitat del animal
     */
    public abstract String getHábitat();
}
```

Clase: Animal

```
package Animales;
/**
 * Esta clase abstracta denominada Cánido modela esta familia de
 * animales. Es una subclase de Animal.
 * @version 1.2/2020
 */
public abstract class Cánido extends Animal {
}
```

Clase: Perro

```
package Animales;
/**
 * Esta clase concreta denominada Perro es una subclase de Cánido.
 * @version 1.2/2020
 */
public class Perro extends Cánido {
```

```
    /**
     * Método que devuelve un String con el sonido de un perro
     * @return Un valor String con el sonido de un perro: "Ladrido"
     */
    public String getSonido() {
        return "Ladrido";
    }

    /**
     * Método que devuelve un String con los alimentos de un perro
     * @return Un valor String con la alimentación de un perro: "Carnívoro"
     */
    public String getAlimentos() {
        return "Carnívoro";
    }

    /**
     * Método que devuelve un String con el hábitat de un perro
     * @return Un valor String con el hábitat de un perro: "Doméstico"
     */
    public String getHábitat() {
        return "Doméstico";
    }

    /**
     * Método que devuelve un String con el nombre científico de un perro
     * @return Un valor String con el nombre científico de un perro:
     * "Canis lupus familiaris"
     */
    public String getNombreCientífico() {
        return "Canis lupus familiaris";
    }
}
```

Clase: Lobo

```
package Animales;

/**
 * Esta clase concreta denominada Lobo es una subclase de Cánido.
 * @version 1.2/2020
 */
public class Lobo extends Cánido {
```

```
    /**
    * Método que devuelve un String con el sonido de un lobo
    * @return Un valor String con el sonido de un lobo: "Aullido"
    */
    public String getSonido() {
        return "Aullido";
    }

    /**
    * Método que devuelve un String con los alimentos de un lobo
    * @return Un valor String con el tipo de alimentación de un lobo:
    * "Carnívoro"
    */
    public String getAlimentos() {
        return "Carnívoro";
    }

    /**
    * Método que devuelve un String con el hábitat de un lobo
    * @return Un valor String con el hábitat de un lobo: "Bosque"
    */
    public String getHábitat() {
        return "Bosque";
    }

    /**
    * Método que devuelve un String con el nombre científico de un lobo
    * @return Un valor String con el nombre científico de un lobo:
    * "Canis lupus"
    */
    public String getNombreCientífico() {
        return "Canis lupus";
    }
}
```

Clase: Felino

```
package Animales;

/**
 * Esta clase abstracta denominada Felino modela esta familia de
 * animales. Es una subclase de Animal.
 * @version 1.2/2020
 */
public abstract class Felino extends Animal {
}
```

Clase: León

```
package Animales;

/**
 * Esta clase concreta denominada León es una subclase de Felino.
 * @version 1.2/2020
 */
public class León extends Felino {

    /**
    * Método que devuelve un String con el sonido de un león
    * @return Un valor String con el sonido de un león: "Rugido"
    */
    public String getSonido() {
        return "Rugido";
    }

    /**
    * Método que devuelve un String con los alimentos de un león
    * @return Un valor String con la alimentación de un león: "Carnívoro"
    */
    public String getAlimentos() {
        return "Carnívoro";
    }

    /**
    * Método que devuelve un String con el hábitat de un león
    * @return Un valor String con el hábitat de un león: "Praderas"
    */
```

```
    public String getHábitat() {
        return "Praderas";
    }

    /**
    * Método que devuelve un String con el nombre científico de un león
    * @return Un valor String con el nombre científico de un león:
    * "Panthera leo"
    */
    public String getNombreCientífico() {
        return "Panthera leo";
    }
}
```

Clase: Gato

```
package Animales;

/**
 * Esta clase concreta denominada Gato es una subclase de Felino.
 * @version 1.2/2020
 */
public class Gato extends Felino {

    /**
    * Método que devuelve un String con el sonido de un gato
    * @return Un valor String con el sonido de un gato: "Maullido"
    */
    public String getSonido() {
        return "Maullido";
    }

    /**
    * Método que devuelve un String con los alimentos de un gato
    * @return Un valor String con la alimentación de un gato: "Ratones"
    */
    public String getAlimentos() {
        return "Ratones";
    }

    /**
    * Método que devuelve un String con el hábitat de un gato
    * @return Un valor String con el hábitat de un gato: "Doméstico"
    */
```

```
    public String getHábitat() {
        return "Doméstico";
    }

    /**
    * Método que devuelve un String con el nombre científico de un gato
    * @return Un valor String con el nombre científico de un gato:
    * "Felis silvestris catus"
    */
    public String getNombreCientífico() {
        return "Felis silvestris catus";
    }
}
```

Clase: Prueba

```
package Animales;

/**
 * Esta clase prueba diferentes animales y sus métodos.
 * @version 1.2/2020
 */
public class Prueba {

    /**
    * Método main que crea un array de varios animales y para cada uno
    * muestra en pantalla su nombre científico, su sonido, alimentos y
    * hábitat
    */
    public static void main(String[] args) {
        Animal[] animales = new Animal[4]; /* Define un array de cuatro
            elementos de tipo Animal */
        animales[0] = new Gato();
        animales[1] = new Perro();
        animales[2] = new Lobo();
        animales[3] = new León();

        for (int i = 0; i < animales.length; i++) { /* Recorre el array de
            animales */
            System.out.println(animales[i].getNombreCientífico());
            System.out.println("Sonido: " + animales[i].getSonido());
            System.out.println("Alimentos: " + animales[i].
                getAlimentos());
```

```
                System.out.println("Hábitat: " + animales[i].getHábitat());
                System.out.println();
            }
        }
    }
```

Diagrama de clases

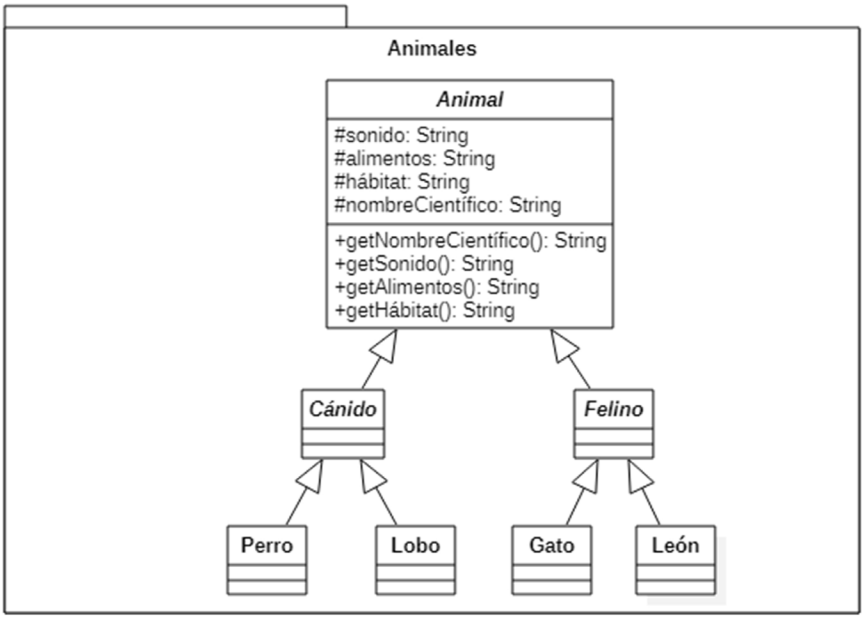


Figura 4.18. Diagrama de clases del ejercicio 4.7.

Explicación del diagrama de clases

El diagrama de clases UML define una jerarquía de clases donde la clase abstracta Animal es la raíz de la jerarquía y es una clase abstracta que se identifica en UML colocando el *nombre de la* clase *en cursiva*. Es una clase abstracta debido a que es muy genérica y lo que interesa es obtener instancias más específicas de un animal. La clase Animal tiene cuatro atributos protegidos (identificados con el símbolo #): sonido, alimentos, hábitat y nombre científico. A su vez, la clase Animal tiene métodos *get* públicos (identificados con el símbolo +) para obtener los valores de dichos atributos.

La clase Animal tiene dos subclases: Cánido y Felino. Ambas son clases muy genéricas y amplias. Por ello, se especifican como abstractas en el diagrama de clases. Las clases Perro, Lobo, Gato y León son clases concretas que sí se pueden instanciar. Las clases Perro y Lobo son subclases de Cánido y las clases Gato y León, de Felino. Por lo tanto, por medio de la herencia estas clases Perro, Lobo, Gato y Perro heredan tanto los atributos (sonidos, alimentos, hábitat y nombre científico) y los métodos *get* de la clase Animal.

Diagrama de objetos

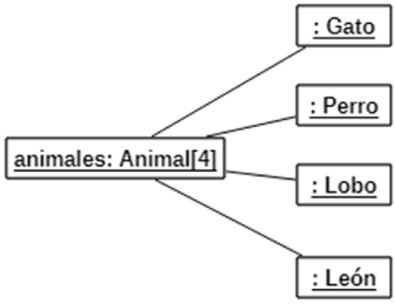


Figura 4.19. Diagrama de objetos del ejercicio 4.7.

Ejecución del programa

```
Felis silvestris catus
Sonido: Maullido
Alimentos: Ratones
Hábitat: Doméstico

Canis lupus familiaris
Sonido: Ladrido
Alimentos: Carnívoro
Hábitat: Doméstico

Canis lupus
Sonido: Aullido
Alimentos: Carnívoro
Hábitat: Bosque

Panthera leo
Sonido: Rugido
Alimentos: Carnívoro
Hábitat: Praderas
```

Figura 4.20. Ejecución del programa del ejercicio 4.7.

Ejercicios propuestos

- Definir una clase abstracta denominada Numérica que tenga los siguientes métodos abstractos:
  - public *String* toString(): convierte el número a *String*.
  - public *boolean* equals (Object ob): compara el objeto con el parámetro.
  - public Numérica sumar(Numérica número): retorna la suma de los dos números.
  - public Numérica restar(Numérica número): retorna la resta de los dos números.
  - public Numérica multiplicar(Numérica número): retorna la multiplicación de los dos números.
  - public Numérica dividir(Numérica número): retorna la división de los dos números.
- Definir una clase Fracción que representa un número fraccionario, el cual hereda de la clase Numérica y tiene dos atributos (tipo *int*)

que representan el numerador y denominador de la fracción. Se deben implementar todos los métodos heredados.
- Crear una clase de prueba que utilice los métodos implementados.

Ejercicio 4.8. Métodos abstractos

Un método abstracto no tiene implementación, por lo tanto, no tiene cuerpo o código, sin llaves, y seguido de un punto y coma (;) (Clark, 2017). Debe tener la palabra reservada *abstract* en la signatura del método:

abstract void método(parámetros);

Si una clase incluye métodos abstractos, la clase debe declararse abstracta. Cuando una clase abstracta tiene subclases, estas deben proporcionar implementaciones para todos los métodos abstractos. Si no lo hacen, también deben declararse abstractas.

Los métodos abstractos se pueden aplicar cuando no se conoce cómo será implementado un método en la clase donde se está definiendo. Debido a que el método está definido en una clase bastante general, es posible que no se conozca, en ese momento, la forma en que se implementará el método. Solamente cuando se hayan definido clases más especializadas (subclases) se contará con un mejor conocimiento para definir en forma explícita el cuerpo o texto del método definido en las clases antecesoras como abstractos.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:
- Definir métodos abstractos en una clase abstracta.
- Implementar métodos abstractos en subclases.

Enunciado: jerarquía de herencia de Ciclista

En una carrera ciclística, un equipo está conformado por un conjunto de ciclistas y se identifica por el nombre del equipo (tipo *String*), la suma de los tiempos de carrera de sus ciclistas en minutos (atributo estático) y país del equipo. Sus atributos deben ser privados.

Un ciclista es una clase abstracta que se describe con varios atributos: identificador (de tipo *int*), nombre del ciclista y tiempo acumulado de carrera (en minutos, con valor inicial cero). Los atributos deben ser privados. Un ciclista tiene un método abstracto *imprimir*Tipo que devuelve un *String*.

Los ciclistas se clasifican de acuerdo con su especialidad (sus atributos deben ser privados y sus métodos protegidos). Estas especialidades no son clases abstractas y heredan los siguientes aspectos de la *clase Ciclista*:

- Velocista: tiene nuevos atributos como potencia promedio (en vatios) y velocidad promedio en *sprint* (Km/h) (ambos de tipo *double*).
- Escalador: tiene nuevos atributos como aceleración promedio en subida (m/s^2) y grado de rampa soportada (grados) (ambos de tipo *float*).
- Contrarrelojista: tiene un nuevo atributo, velocidad máxima (km/h).

Definir clases y métodos para el ciclista y sus clases hijas para realizar las siguientes acciones:

- Constructores para cada clase (deben llamar a los constructores de la clase padre en las clases donde se requiera).
- Métodos *get* y *set* para cada atributo de cada clase.
- Imprimir los datos de un ciclista. Debe invocar el método de la clase padre e imprimir los valores de los atributos propios.
- Método *imprimirTipo* que devuelve un *String* con el texto "Es un *xxx*". Donde *xxx* es la clase a la que pertenece.

La clase Equipo debe tener los siguientes métodos protegidos:

- Métodos *get* y *set* para cada atributo de la clase.
- Imprimir los datos del equipo en pantalla.
- Añadir un ciclista a un equipo.
- Calcular el total de tiempos de los ciclistas del equipo (suma de los tiempos de carrera de sus ciclistas, su atributo estático).
- Listar los nombres de todos los ciclistas que conforman el equipo.
- Dado un identificador de un ciclista por teclado, es necesario *imprimir* en pantalla los datos del ciclista. Si no existe, debe aparecer el mensaje correspondiente.

En una clase de prueba, en un método *main* se debe crear un equipo y agregar ciclistas de diferentes tipos.

Solución

Clase: Equipo

```
package CarreraCiclística;

import java.util.*;
 /**
 * Esta clase denominada Equipo modela un equipo de ciclismo
 * conformado por un vector de ciclistas. La clase tiene atributos como
 * el nombre del equipo, el país del equipo y el tiempo total acumulado
 * del equipo
 * @version 1.2/2020
 */
public class Equipo {
    private String nombre; /* Atributo que define el nombre del equipo
        de ciclismo */
    private static double totalTiempo; /* Atributo que define el tiempo
        total obtenido por el equipo */
    private String país; /* Atributo que define el país al que pertenece el
        equipo */
    Vector listaCiclistas; // Atributo que define un vector de objetos ciclista

    /**
    * Constructor de la clase Equipo
    * @param nombre Parámetro que define el nombre del equipo
    * @param país Parámetro que define el país del equipo
    */
    public Equipo(String nombre, String país) {
        this.nombre = nombre;
        this.país =  país;
        totalTiempo = 0; // Se inicializa el tiempo del equipo en cero
        listaCiclistas = new Vector(); /* Se crea el vector de ciclistas que
            conforma el equipo */
    }
```

```
/**
* Método que devuelve el nombre del equipo
* @return El nombre del equipo
*/
public String getNombre() {
    return nombre;
}

/**
* Método que establece el nombre de un equipo
* @param Parámetro que especifica el nombre de un equipo
*/
public void setNombre(String nombre) {
    this.nombre = nombre;
}

/**
* Método que devuelve el país del equipo
* @return El país del equipo
*/
private String getPaís() {
    return país;
}

/**
* Método que establece el país de un equipo
* @param Parámetro que especifica el país de un equipo
*/
private void setPaís(String país) {
    this.país =  país;
}

/**
* Método que añade un ciclista al vector de ciclistas de un equipo
*/
void añadirCiclista(Ciclista ciclista) {
    listaCiclistas.add(ciclista); // Se agrega el ciclista al vector de ciclistas
}

/**
* Método que muestra en pantalla los nombres de los ciclistas que
* conforman un equipo
*/
```

```
    void listarEquipo() {
        /* Se recorre el vector de ciclistas y para cada elemento se
           imprime el nombre del ciclista */
        for (int i = 0; i < listaCiclistas.size(); i++) {
            Ciclista c = (Ciclista) listaCiclistas.elementAt(i); /* Se aplica
               casting para extraer el elemento */
            System.out.println(c.getNombre());
        }
    }

    /**
    * Método que busca un ciclista ingresado por teclado
    */
    void buscarCiclista() {
        Scanner sc = new Scanner(System.in); /* Se solicita texto
           ingresado por teclado */
        String nombreCiclista = sc.next();
        for (int i = 0; i < listaCiclistas.size(); i++) { /* Se recorre el vector
           de ciclistas */
            Ciclista c = (Ciclista) listaCiclistas.elementAt(i); /* Se obtiene
               un elemento del vector */
            if (c.getNombre().equals(nombreCiclista)) { /* Si el nombre
               ingresado se encuentra */
                System.out.println(c.getNombre());
            }
        }
    }

    /**
    * Método que calcula el tiempo total de un equipo acumulando el
    * tiempo obtenido por cada uno de sus ciclistas
    */
    void calcularTotalTiempo() {
        for (int i = 0; i < listaCiclistas.size(); i++) { // Se recorre el vector
            Ciclista c = (Ciclista) listaCiclistas.elementAt(i); /* Se obtiene
               un elemento del vector */
            // Se acumula el tiempo del ciclista en el tiempo del equipo
            totalTiempo = totalTiempo + c.getTiempoAcumulado();
        }
    }
```

```
    /**
     * Método que muestra en pantalla los datos de un equipo
     */
    void imprimir() {
        System.out.println("Nombre del equipo = " + nombre);
        System.out.println("País = " + país);
        System.out.println("Total tiempo del equipo = " + totalTiempo);
    }
}
```

Clase: *Ciclista*

```
package CarreraCiclística;

/**
 * Esta clase abstracta denominada Ciclista posee como atributos un
 * identificador, un nombre y un tiempo acumulado en una carrera
 * ciclística.
 * @version 1.2/2020
 */
public abstract class Ciclista {
    private int identificador; /* Atributo que define el identificador de
        un ciclista */
    private String nombre; // Atributo que define el nombre del ciclista
    private int tiempoAcumulado = 0; /* Atributo que define el tiempo
        acumulado de un ciclista */

    /**
     * Constructor de la clase Ciclista
     * @param identificador Parámetro que define el identificador de un
     * ciclista
     * @param nombre Parámetro que define el nombre completo de un
     * ciclista
     */
    public Ciclista(int identificador, String nombre) {
        this.identificador = identificador;
        this.nombre = nombre;
    }

    /**
     * Método abstracto que muestra en pantalla el tipo específico de un
     * ciclista
     * @return Tipo de ciclista
     */
```

```
abstract String imprimirTipo();

/**
* Método que devuelve el identificador de un ciclista
* @return El identificador de un ciclista
*/
protected int getIdentificador() {
    return identificador;
}

/**
* Método que establece el identificador de un ciclista
* @param Parámetro que especifica el identificador de un ciclista
*/
protected void setIdentificador() {
    this.identificador = identificador;
}

/**
* Método que devuelve el nombre de un ciclista
* @return El nombre de un ciclista
*/
```

```
protected String getNombre() {
    return nombre;
}

/**
* Método que establece el nombre de un ciclista
* @param Parámetro que especifica el nombre de un ciclista
*/
protected void setNombre(String nombre) {
    this.nombre = nombre;
}

/**
* Método que devuelve el puesto que ocupa un ciclista en la
* posición general de la competencia
* @return El puesto del ciclista en la posición general
*/
protected int getPosiciónGeneral(int posiciónGeneral) {
    return posiciónGeneral;
}

/**
* Método que establece el puesto que ocupa un ciclista en la
* posición general
* @param Parámetro que especifica el puesto que ocupa un ciclista
* en la posición general
*/
protected void setPosiciónGeneral(int posiciónGeneral) {
    posiciónGeneral = posiciónGeneral;
}

/**
* Método que devuelve el tiempo acumulado de un ciclista en una
* competencia
* @return El tiempo acumulado de un ciclista en una competencia
*/
protected int getTiempoAcumulado() {
    return tiempoAcumulado;
}

/**
* Método que establece el tiempo acumulado por un ciclista
* @param Parámetro que especifica el tiempo acumulado por un ciclista
*/
```

```
    protected void setTiempoAcumulado(int tiempoAcumulado) {
        this.tiempoAcumulado = tiempoAcumulado;
    }

    /**
    * Método muestra en pantalla los datos de un ciclista
    */
    protected void imprimir() {
        System.out.println("Identificador = " + identificador);
        System.out.println("Nombre = " + nombre);
        System.out.println("Tiempo Acumulado = " +
            tiempoAcumulado);
    }
}
```

Clase: Velocista

```
package CarreraCiclística;

/**
* Esta clase denominada Velocista es un tipo de Ciclista caracterizado
* por poseer gran potencia y alta velocidad punta en esfuerzos cortos.
* Posee nuevos atributos como la potencia promedio y la velocidad
* promedio
* @version 1.2/2020
*/
public class Velocista extends Ciclista {
    private double potenciaPromedio; /* Atributo que define la potencia
        promedio de un velocista */
    private double velocidadPromedio; /* Atributo que define la
        velocidad promedio de un velocista */

    /**
    * Constructor de la clase Velocista
    * @param identificador Parámetro que define el identificador de un
    * velocista
    * @param nombre Parámetro que define el nombre de un velocista
    * @param potenciaPromedio Parámetro que define la potencia
    * promedio de un velocista
    * @param velocidadPromedio Parámetro que define la velocidad
    * promedio de un velocista
    */
```

```
public Velocista(int identificador, String nombre, double
    potenciaPromedio, double velocidadPromedio) {
    super(identificador, nombre);
    potenciaPromedio = potenciaPromedio;
    this.velocidadPromedio = velocidadPromedio;
}

/**
* Método que devuelve la potencia promedio de un velocista
* @return La potencia promedio de un velocista
*/
protected double getPotenciaPromedio() {
    return potenciaPromedio;
}

/**
* Método que establece la potencia promedio de un velocista
* @param Parámetro que especifica la potencia promedio de un
* velocista
*/
protected void setPotenciaPromedio(double potenciaPromedio) {
    this.potenciaPromedio = potenciaPromedio;
}

/**
* Método que devuelve la velocidad promedio de un velocista
* @return La velocidad promedio de un velocista
*/
protected double getvelocidadPromedio() {
    return velocidadPromedio;
}

/**
* Método que establece la velocidad promedio de un velocista
* @param Parámetro que especifica la velocidad promedio de un
* velocista
*/
protected void setVelocidadPromedio(double velocidadPromedio) {
    this.velocidadPromedio = velocidadPromedio;
}
```

```
    /**
     * Método que muestra en pantalla los datos de un velocista
     */
    protected void imprimir() {
        super.imprimir(); // Invoca al método imprimir de la clase padre
        System.out.println("Potencia promedio = " + potenciaPromedio);
        System.out.println("Velocidad promedio = " +
            velocidadPromedio);
    }

    /**
     * Método que devuelve el tipo de ciclista
     * @return Un valor String con el texto "Es un velocista"
     */
    protected String imprimirTipo() {
        return "Es un velocista";
    }
}
```

Clase: Escalador

```
package CarreraCiclística;

/**
 * Esta clase denominada Escalador es un tipo de Ciclista que se
 * encuentra mejor adaptado y se destaca cuando las carreteras son en
 * ascenso, ya sea en colinas o montañas. Posee nuevos atributos como
 * su aceleración promedio y el grado de rampa que soporta
 * @version 1.2/2020
 */
public class Escalador extends Ciclista {
    // Atributo que define la acelaración promedio de un escalador
    private double aceleraciónPromedio;
    // Atributo que define el grado de rampa soportado por un escalador
    private double gradoRampa;

    /**
     * Constructor de la clase Escalador
     * @param identificador Parámetro que define el identificador de un
     * escalador
```

```
* @param nombre Parámetro que define el nombre de un escalador
* @param aceleraciónPromedio Parámetro que define la aceleración
* promedio de un escalador
* @param gradoRampa Parámetro que define el grado de rampa de
* un escalador
*/
public Escalador(int identificador, String nombre, double
    aceleraciónPromedio, double gradoRampa) {
    super(identificador, nombre);
    this.aceleraciónPromedio = aceleraciónPromedio;
    this.gradoRampa = gradoRampa;
}

/**
* Método que devuelve la aceleración promedio de un escalador
* @return La aceleración promedio de un escalador
*/
protected double getAceleraciónPromedio() {
    return aceleraciónPromedio;
}

/**
* Método que establece la aceleración promedio de un escalador
* @param Parámetro que especifica la aceleración promedio de un
* escalador
*/
protected void setAceleraciónPromedio(double
    aceleraciónPromedio) {
    this.aceleraciónPromedio = aceleraciónPromedio;
}

/**
* Método que devuelve el grado de rampa soportado de un escalador
* @return El grado de rampa soportado de un escalador
*/
protected double getGradoRampa() {
    return gradoRampa;
}

/**
* Método que establece el grado de rampa soportado por un escalador
* @param Parámetro que especifica el grado de rampa soportado
* por un escalador
*/
```

```
    protected void setGradoRampa(double gradoRampa) {
        this.gradoRampa = gradoRampa;
    }

    /**
    * Método que muestra en pantalla los datos de un escalador
    */
    protected void imprimir() {
        super.imprimir(); // Invoca el método imprimir de la clase padre
        System.out.println("Aceleración promedio = " +
            aceleraciónPromedio);
        System.out.println("Grado de rampa = " + gradoRampa);
    }

    /**
    * Método que devuelve el tipo de ciclista
    * @return Un valor String con el texto "Es un escalador"
    */
    protected String imprimirTipo() {
        return "Es un escalador";
    }
}
```

Clase: Contrarrelojista

```
package CarreraCiclística;

/**
* Esta clase denominada Contrarrelojista es un tipo de Ciclista que se
* encuentra mejor adaptado a las etapas contrarreloj. Posee un nuevo
* atributo: su velocidad máxima
* @version 1.2/2020
*/
public class Contrarrelojista extends Ciclista {
    // Atributo que define la velocidad máxima de un contrarrelojista
    private double velocidadMáxima;

    /**
    * Constructor de la clase Escalador
    * @param identificador Parámetro que define el identificador de un
    * contrarrelojista
    * @param nombre Parámetro que define el nombre de un
    * contrarrelojista
```

```
     * @param velocidadMáxima Parámetro que define la velocidad
     * máxima de un contrarrelojista
     */
    public Contrarrelojista(int identificador, String nombre, double
        velocidadMáxima) {
        super(identificador, nombre);
        this.velocidadMáxima = velocidadMáxima;
    }

    /**
     * Método que devuelve la velocidad máxima de un contrarrelojista
     * @return La velocidad máxima de un contrarrelojista
     */
    protected double getVelocidadMáxima() {
        return velocidadMáxima;
    }

    /**
     * Método que establece la velocidad máxima de un contrarrelojista
     * @param Parámetro que especifica la velocidad máxima de un
     * contrarrelojista
     */
    protected void setVelocidadMáxima(double velocidadMáxima) {
        this.velocidadMáxima = velocidadMáxima;
    }

    /**
     * Método que muestra en pantalla los datos de un contrarrelojista
     */
    protected void imprimir() {
        super.imprimir(); // Invoca el método imprimir de la clase padre
        System.out.println("Aceleración promedio = " +
            velocidadMáxima);
    }

    /**
     * Método que devuelve el tipo de ciclista
     * @return Un valor String con el texto "Es un constrarrelojista"
     */
    protected String imprimirTipo() {
        return "Es un constrarrelojista";
    }
}
```

Clase: Prueba

```
package CarreraCiclística;

/**
 * Esta clase prueba diferentes acciones realizadas por un equipo ciclístico
 * y sus diferentes corredores
 * @version 1.2/2020
 */
public class Prueba {

    /**
     * Método main que crea un equipo. Luego, crea un escalador, un
     * velocista y un contrarrelojista. Estos tipos de ciclistas son añadidos
     * al equipo. Se asignan tiempos a cada ciclista para al final obtener el
     * tiempo total del equipo
     */
    public static void main(String args[]) {
        Equipo equipo1 = new Equipo("Sky","Estados Unidos");
        Velocista velocista1 = new Velocista(123979, "Geraint Thomas",
            320, 25);
        Escalador escalador1 = new Escalador(123980, "Egan Bernal",
            25, 10);
        Contrarrelojista contrarrelojista1 = new Contrarrelojista(123981,
            "Jonathan Castroviejo", 120);
        equipo1.añadirCiclista(velocista1);
        equipo1.añadirCiclista(escalador1);
        equipo1.añadirCiclista(contrarrelojista1);
        velocista1.setTiempoAcumulado(365);
        escalador1.setTiempoAcumulado(385);
        contrarrelojista1.setTiempoAcumulado(370);
        equipo1.calcularTotalTiempo();
        equipo1.imprimir();
        equipo1.listarEquipo();
    }
}
```

Diagrama de clases

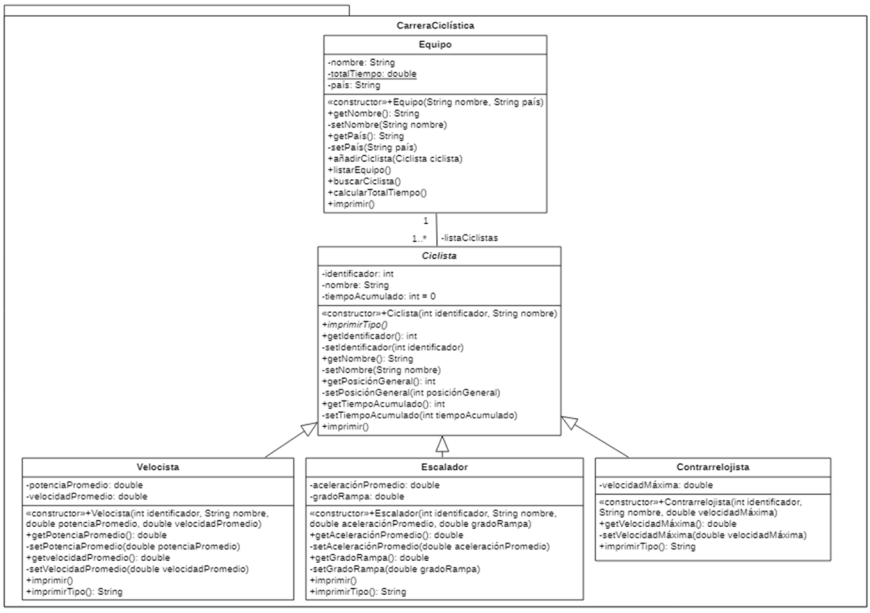


Figura 4.21. Diagrama de clases del ejercicio 4.8.

Explicación del diagrama de clases

El diagrama de clases UML define un paquete denominado "CarreraCiclística". Un Equipo tiene tres atributos privados (-): el nombre del equipo, el país del equipo y el total de tiempo obtenido por el equipo. Este último atributo es estático; por lo tanto, será compartido por todos los objetos de tipo Ciclista y en UML se indica con el texto subrayado. La clase Equipo tiene un constructor, métodos *get* y *set* para cada uno de sus atributos y métodos públicos para añadir un ciclista al equipo; listar los ciclistas del equipo; buscar un ciclista; calcular el tiempo del equipo e imprimir en pantalla los datos del equipo.

Un equipo está conformado por varios ciclistas. Esta relación se representa en UML con una relación de asociación, la cual puede ser de 1 a muchos, lo que indica que un equipo tiene una lista de ciclistas que es una colección específica de ciclistas, en este caso un vector de ciclistas.

Los ciclistas pueden ser de diferente tipo como se puede observar en la jerarquía de herencia del diagrama, donde la clase *Ciclista* es una clase abstracta (su nombre se presenta en cursiva) y las clases Velocista, Escalador y Contrarrelojista son clases hijas de *Ciclista*.

La clase *Ciclista* tiene tres atributos privados: identificador, nombre y tiempo acumulado inicializado con el valor cero. También posee un constructor, un método abstracto denominado imprimirTipo, el cual debe ser implementado en las clases hijas, métodos *get* y *set* para cada uno de sus atributos y un método imprimir que muestra los datos de un ciclista en pantalla.

Cada una de las clases hijas de *Ciclista* poseen atributos y métodos específicos de acuerdo con su tipo. Estas clases no son abstractas y, por lo tanto, podrán ser instanciadas en objetos específicos.

En primer lugar, la clase Velocista tiene como nuevos atributos privados la potencia y velocidad promedio.

En segundo lugar, la clase Escalador tiene como nuevos atributos privados la aceleración promedio y el grado de rampa soportado.

Finalmente, la clase Contrarrelojista tiene como nuevo atributo privado la velocidad máxima. Las clases Velocista, Escalador y Contrarrelojista cuentan con un constructor y métodos *get*, *set*, *imprimir* datos del contrarrelojista e implementa el método abstracto imprimirTipo.

Diagrama de objetos

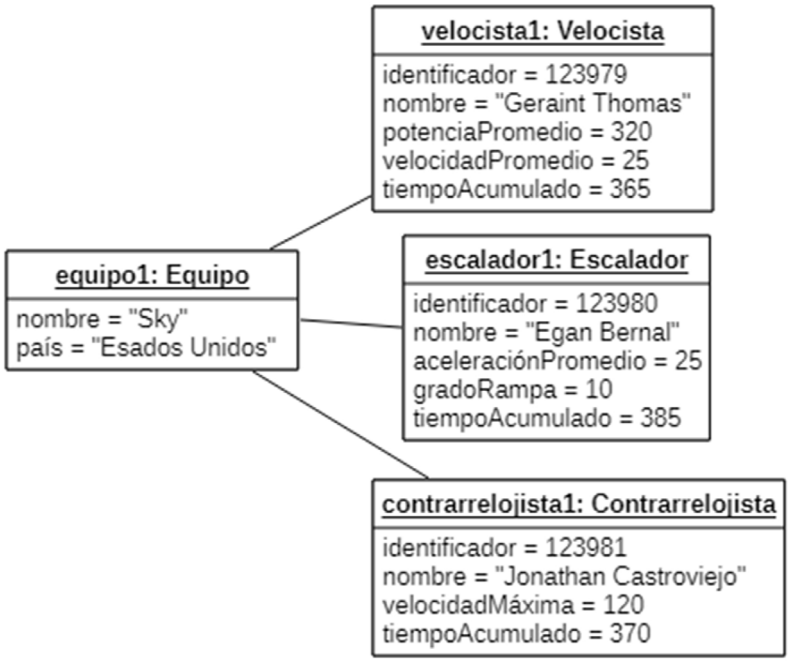


Figura 4.22. Diagrama de objetos del ejercicio 4.8.

Ejecución del programa

```
Nombre del equipo = Sky
País = Estados Unidos
Total tiempo del equipo = 1120.0
Geraint Thomas
Egan Bernal
Jonathan Castroviejo
```

Figura 4.23. Ejecución del programa del ejercicio 4.8.

Ejercicios propuestos

- Realizar el ejercicio 2.4 sobre figuras geométricas definiendo una clase abstracta denominada *Figura geométrica* con los métodos abstractos para calcular el área y el perímetro de la figura. Las subclases Círculo, Rectángulo y Cuadrado deben heredar de la clase

Figura geométrica. La clase Triángulo rectángulo debe ser una subclase de Triángulo.

Ejercicio 4.9. Operador *instaceof*

El operador de *instanceof* de Java se utiliza para evaluar si un objeto es una instancia de una clase particular o de sus subclases (API Java, 2020). Devuelve un valor verdadero o falso. Si se aplica el operador con cualquier variable que tenga un valor nulo, devuelve falso.

El formato del operador es el siguiente:

objeto instanceof Clase

Todos los objetos en Java son instancias de la superclase *Object*. Si se aplica el operador *instanceof* al tipo *Object*, siempre devuelve verdadero.

Objetivo de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para aplicar el operador *instanceof* y determinar si un objeto es instancia de una clase particular.

Enunciado: clase Médico con jerarquía de herencia

Un médico posee como atributo su nombre y métodos *get* y *set* de dicho atributo. Los pediatras y ortopedistas son dos tipos de médicos. Estas dos subclases se modelan como subclases de médico.

Los pediatras pueden ser pediatras neurólogos o pediatras psicológicos. Esta tipología se modela como un atributo enumerado de pediatra. De igual manera, los ortopedistas cuentan con un atributo enumerado para describir su tipo que puede ser maxilofacial o pediátrica. Estas subclases cuentan con métodos *get* y *set*.

En una clase de prueba, en su método *main*, se solicita crear un vector con objetos tanto de tipo pediatra como ortopedistas. Luego, en pantalla se debe mostrar qué tipo de objeto está ubicado en cada posición del vector.

Las clases deben estar contenidas en un paquete denominado Medicina.

Solución

Clase: Médico

```
package Medicina;

/**
 * Esta clase denominada Médico modela un médico con un solo
 * atributo: su nombre
 * @version 1.2/2020
 */
class Médico {
    String nombre; // Atributo que define el nombre de un médico

    /**
     * Método que devuelve el nombre de un médico
     * @return El nombre del médico
     */
    String getNombre() {
        return nombre;
    }

    /**
     * Método que estable el nombre de un médico
     * @param nombre Parámetro que define el nombre de un médico
     */
    void setNombre(String nombre) {
        this.nombre = nombre;
    }
}
```

Clase: Médico

```
package Medicina;

/**
 * Esta clase denominada Pediatra es una subclase de Médico que
 * cuenta con un atributo tipo que determina si el pediatra es pediatra
 * neurólogo o psicólogo
 * @version 1.2/2020
 */
public class Pediatra extends Médico {
    // Valor enumerado de define tipologías de un pediatra
```

```
    enum tipología {NEUROLOGO, PSICOLOGO};
    tipología tipo; // Atributo que define el tipo de pediatra

    /**
    * Método que devuelve el tipo de pediatra
    * @return El tipo de pediatra
    */
    void setTipología(tipología tipo) {
        this.tipo = tipo;
    }

    /**
    * Método que establece el tipo de pediatra
    * @param nombre Parámetro que define el tipo de pediatra
    */
    tipología getTipología() {
        return tipo;
    }
}
```

Clase: Ortopedista

```
package Medicina;

/**
* Esta clase denominada Ortopedista es una subclase de Médico que
* cuenta con un atributo tipo que determina si el ortopedista es
* maxilofacial o pediátrico
* @version 1.2/2020
*/
public class Ortopedista extends Médico {
    // Valor enumerado para definir diferentes tipo de ortopedista
    enum tipología {MAXILOFACIAL, PEDIÁTRICA};
    tipología tipo; // Atributo que define el tipo de ortopedista

    /**
    * Método que estable el tipo de ortopedista
    * @param nombre Parámetro que define el tipo de ortopedista
    */
    void setTipología(tipología tipo) {
        this.tipo = tipo;
    }
```

```
    /**
     * Método que devuelve el tipo de ortopedista
     * @return El tipo de ortopedista
     */
    tipología getTipología() {
        return tipo;
    }
}
```

Clase: Prueba

```
package Medicina;
import java.util.*;

/**
 * Esta clase prueba diferentes acciones realizadas por la clase Pediatra y
 * Ortopedista
 * @version 1.2/2020
 */
public class Prueba {

    /**
     * Método main que crea un vector de médicos y luego agrega un
     * médico, un ortopedista y un pediatra al vector de médicos. Luego,
     * se recorre el vector de médicos para determinar qué tipo de
     * médico es cada elemento y obteniendo su posición en el vector
     */
    public static void main(String[] args)  {
        Vector listaMédicos = new Vector();
        Médico médico1 = new Médico();
        listaMédicos.add(médico1);
        Ortopedista ortopedista1 = new Ortopedista();
        listaMédicos.add(ortopedista1);
        Pediatra pediatra1 = new Pediatra();
        listaMédicos.add(pediatra1);

        for (int i = 0; i < listaMédicos.size(); i++) {
            // Se debe realizar un proceso de casting con la clase padre
            Médico a = (Médico) listaMédicos.elementAt(i);
            if (a instanceof Ortopedista) { /* Determina si el elemento es
                un ortopedista */
```

```
                System.out.println("El objeto en el indice "+ i + " es de la
                    clase Ortopedista");
                continue;
            }
            if (a instanceof Pediatra) { /* Determina si el elemento es un
                pediatra */
                System.out.println("El objeto en el indice " + i + " es de la
                    clase Pediatra");
                continue;
            }
            if (a instanceof Médico) { /* Determina si el elemento es un
                médico */
                System.out.println("El objeto en el indice "+i + " es de la
                    clase Médico");
                continue;
            }
        }
    }
}
```

Diagrama de clases

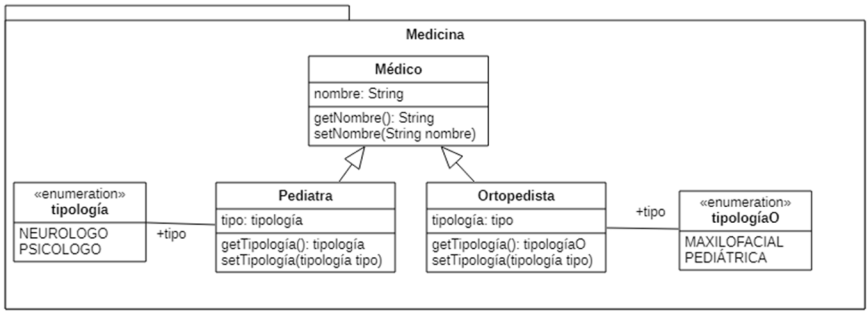


Figura 4.24. Diagrama de clases del ejercicio 4.9.

Explicación del diagrama de clases

El diagrama de clases UML define un paquete denominado "Medicina". Dentro del paquete se define una pequeña jerarquía de clases con la clase Médico como clase raíz. La clase Médico tiene un solo atributo: nombre del médico. La clase Médico tiene métodos *get* y *set* para obtener y establecer

el nombre del médico. La clase Médico tiene dos subclases: Pediatra y Ortopedista. Ambos tienen un atributo denominado tipo, el cual es un valor enumerado con dos posibles valores.

Los atributos enumerados se representan en el diagrama de clases UML como clases con el estereotipo <<*enumeration*>>. Cada clase está asociada a la clase enumerada. El atributo enumerado denominado "tipo" de la clase Pediatra asume dos valores: NEURÓLOGO o PSICÓLOGO. El atributo enumerado denominado "tipo" de la clase Pediatra asume dos valores: MAXILOFACIAL o PEDIÁTRICA.

Diagrama de objetos

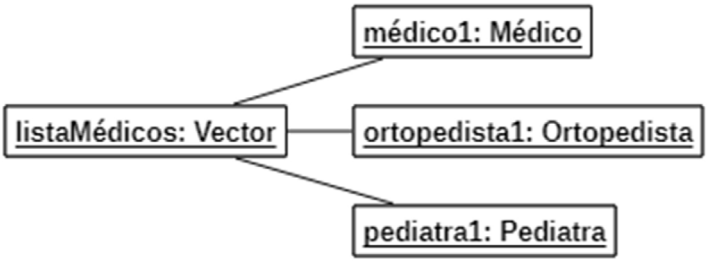


Figura 4.25. Diagrama de objetos del ejercicio 4.9.

Ejecución del programa

```
El objeto en el indice 0 es de la clase Médico
El objeto en el indice 1 es de la clase Ortopedista
El objeto en el indice 2 es de la clase Pediatra
```

Figura 4.26. Ejecución del programa del ejercicio 4.9.

Ejercicios propuestos

- Cuál es el resultado de la ejecución del siguiente programa:

public class *Array* {

```
    public static void main(String[] args) {
        int[] arrayInt = new int[5];
        float[] arrayFloat = new float[5];
        Integer[] arrayObjetosInt = new Integer[5];
        System.out.println(arrayInt instanceof Object);
        System.out.println(arrayInt instanceof int[]);
        System.out.println(arrayFloat instanceof Object);
        System.out.println(arrayFloat instanceof float[]);
        System.out.println(arrayObjetosInt instanceof Object);
        System.out.println(arrayObjetosInt instanceof Object[]);
        System.out.println(arrayObjetosInt instanceof Integer[]);
    }
}
```

Ejercicio 4.10. Interfaces

Algunas veces se necesitan objetos que compartan una colección de métodos, que no pertenezcan a la misma jerarquía de herencia (Joy, Bracha, Steele, Buckley y Gosling, 2013). Las interfaces son la solución.

Al igual que las clases abstractas, las interfaces no se pueden instanciar (Schildt, 2018). Por lo tanto, las interfaces no deben tener constructores.

Una interfaz se define de la siguiente manera:

```
interface nombreInterface {
        nombreMétodo(parámetros);
}
```

Las interfaces son similares a las clases abstractas en que sus métodos son completamente abstractos y pueden contener atributos, pero deben ser estáticos y finales. Los métodos de una interfaz son siempre públicos.

Una clase implementa una interfaz por medio de la palabra clave *implements*:

class nombreClase implements InterfazA

La clase debe implementar todos los métodos de la interfaz. Si no se implementa por lo menos un método abstracto, la clase debe ser abstracta.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Comprender el concepto de interfaz.
- Definir métodos en una interfaz.
- Definir clases que implementan una interfaz.

Enunciado: jerarquía de herencia de Mamífero con interfaz

Hacer un programa que implemente las siguientes clases y métodos relacionados con una jerarquía taxonómica de animales.

- Los mamíferos son una clase abstracta con el método abstracto amamantar crías, que no devuelve nada.
- Las ballenas son mamíferos e implementan el método abstracto heredado, la pantalla muestra un mensaje indicando que ellas amamantan a sus crías.
- Los animales ovíparos son una interfaz con el método poner huevos.
- El ornitorrinco es un mamífero que pone huevos. Por lo tanto, es una clase que hereda de mamífero e implementa la interfaz Ovíparo. El método heredado de la clase padre muestra en pantalla que el ornitorrinco amamanta a sus crías, y el método implementado desde la interfaz muestra en pantalla que pone huevos.

Generar un método *main* donde se crean una ballena y un ornitorrinco y se invocan los métodos heredados e implementados.

Solución

Clase: Mamífero

```
package Mamíferos;

/**
* Esta clase abstracta denominada Mamífero modela una clase de
* vertebrado que amamanta a sus crías.
* @version 1.2/2020
*/
```

```
public abstract class Mamífero {
    /**
     * Método abstracto que presenta que los mamíferos amamantan a
     * sus crías
     */
    abstract void amamantarCrías();
}
```

Clase: Ballena

```
package Mamíferos;
/**
 * Esta clase denominada Ballena modela un tipo específico de mamífero
 * marino.
 * @version 1.2/2020
 */
public class Ballena extends Mamífero {
    /**
     * Método que implementa el método abstracto amamantarCrías
     * heredado de la clase Mamífero que define un texto específico sobre
     * la ballena que amamanta crías
     */
    void amamantarCrías() {
        System.out.println("La ballena amamanta a sus crías.");
    }
}
```

Interface: Ovíparo

```
package Mamíferos;
/**
 * Esta interfaz denominada Ovíparo modela un animal que pone
 * huevos pero que no está relacionado con otras clases por medio de la
 * herencia.
 * @version 1.2/2020
 */
```

```
public interface Ovíparo {
    /**
     * Método abstracto que presenta que los ovíparos pueden poner huevos
     */
    public void ponerHuevos();
}
```

Clase: Ornitorrinco

```
package Mamíferos;
/**
 * Esta clase denominada Ornitorrinco modela un tipo de Mamífero y a
 * su vez implementa la interfaz Ovíparo
 * @version 1.2/2020
 */
public class Ornitorrinco extends Mamífero implements Ovíparo {
    /**
     * Método que implementa el método abstracto amamantarCrías
     * heredado de la clase Mamífero que define un texto específico sobre
     * el ornitorrinco que amamanta crías
     */
    public void amamantarCrías() {
        System.out.println("El ornitorrinco amamanta a sus crías.");
    }
    /**
     * Método que implementa el método ponerHuevos de la interfaz
     * Ovíparo que define un texto específico sobre el ornitorrinco que
     * puede poner huevos
     */
    public void ponerHuevos() {
        System.out.println("El ornitorrinco pone huevos.");
    }
}
```

Clase: Prueba

```
package Mamíferos;

/**
 * Esta clase prueba diferentes acciones realizadas por los mamíferos y
 * sus clases específicas Ballena y Ornitorrinco
 * @version 1.2/2020
 */
public class Prueba {

    /**
     * Método main que crea una ballena y un ornitorrinco e invoca los
     * métodos heredados e implementados
     */
    public static void main(String args[]) {
        Ballena mobyDick = new Ballena(); // Crea una ballena
        mobyDick.amamantarCrías(); /* Invoca el método heredado de la
            clase Mamífero */
        Ornitorrinco ornitorrinco1 = new Ornitorrinco(); /* Crea un
            ornitorrinco */
        ornitorrinco1.amamantarCrías(); /* Invoca el método heredado
            de la clase Mamífero */
        ornitorrinco1.ponerHuevos(); /* Invoca el método implementado
            de la interfaz Ornitorrinco */
    }
}
```

Diagrama de clases

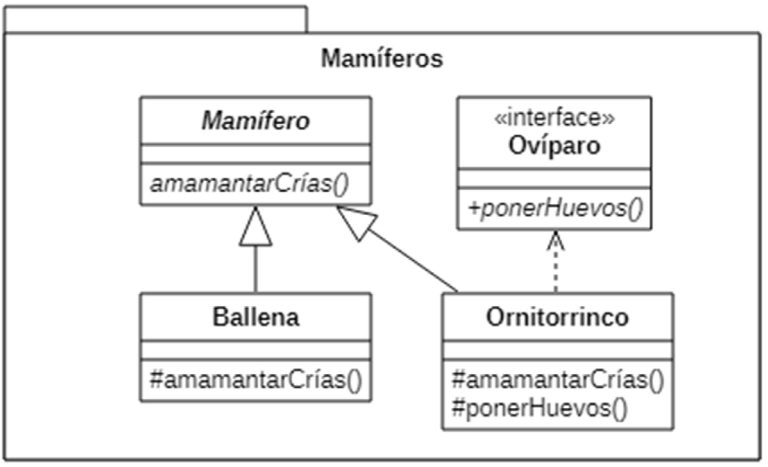


Figura 4.27. Diagrama de clases del ejercicio 4.10.

Explicación del diagrama de clases

El diagrama de clases UML define un paquete denominado "Mamíferos". Dentro del paquete se define una pequeña jerarquía de clases con la clase Mamífero como clase padre. La clase Mamífero contiene un método abstracto denominado "*amamantarCrías*", el cual se representa en UML con el texto en cursiva. La clase Mamífero cuenta con dos subclases: Ballena y Ornitorrinco, las cuales deben implementar el código del método abstracto definido en la clase padre; por ello, aparece el método "*amamantarCrías*" en ambas clases.

La relación de herencia se expresa mediante una línea continua que termina con un triángulo en un extremo que relaciona la clase padre y la clase hija, la clase que se vincula con el triángulo es la clase padre y la clase del otro extremo es la hija.

También se ha definido una interfaz denominada "Ovíparo" por medio de la notación de clase, pero con el estereotipo <<*interface*>> en el nombre de la clase. La interfaz tiene un método llamado "*ponerHuevos*" el cual es implícitamente abstracto (en el diagrama aparece con el texto en cursiva). La clase Ornitorrinco implementa la interfaz mediante una línea discontinua con punta en flecha que representa una relación de "realización" en UML: la clase Ornitorrinco realiza la clase Ovíparo. Para que la clase

Ornitorrinco implemente efectivamente la interfaz Ovario debe agregarle código al método "ponerHuevos", el cual aparece en el compartimiento de métodos de la clase.

Diagrama de objetos

Figura 4.28. Diagrama de objetos del ejercicio 4.10.

Ejecución del programa

```
La ballena amamanta a sus crías.
El ornitorrinco amamanta a sus crías.
El ornitorrinco pone huevos.
```

Figura 4.29. Ejecución del programa del ejercicio 4.10.

Ejercicios propuestos

- Agregar a la solución anterior una nueva interfaz denominada Volador que representa un animal que vuela. Dicha interfaz tiene un método volar que muestra en pantalla que un animal vuela. Además, agregar una nueva subclase de Mamífero llamada Murciélago que a su vez implementa la clase Volador. En la clase Prueba instanciar un murciélago e invocar los métodos heredados e implementados.

Ejercicio 4.11. Interfaces múltiples

Java no soporta la herencia múltiple. Sin embargo, con el uso de interfaces se permite una forma de simulación o implementación limitada de la herencia múltiple (Horstmann, 2018).

Una clase puede implementar más de una interfaz utilizando el siguiente formato:

nombreClase implements InterfazA, interfazB

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Simular la herencia múltiple a través de interfaces.
- Definir clases que permitan simultáneamente implementar métodos heredados y métodos de una interfaz.

Enunciado: clase Vehículo con jerarquía de herencia e implementación de varias interfaces

Un vehículo posee una velocidad actual y una velocidad máxima (ambas en km/h). Un vehículo tiene dos métodos abstractos:

- El método acelerar permite incrementar la velocidad actual sumándole la velocidad pasada como parámetro. La velocidad actualizada no puede superar la velocidad máxima.
- El método frenar permite disminuir la velocidad actual restándole la velocidad pasada como parámetro. La velocidad actualizada no puede ser negativa.
- El método *imprimir* muestra en pantalla la velocidad actual y la velocidad máxima del vehículo.

Existen dos tipos de vehículos: vehículos terrestres y acuáticos. En primer lugar, los vehículos terrestres tienen como nuevos atributos: la cantidad de llantas y el uso del vehículo (militar o civil). En segundo lugar, los vehículos acuáticos tienen como nuevos atributos su tipo (de superficie o submarino) y capacidad de pasajeros.

A su vez, existen dos interfaces: Motor y Vela. La clase Vehículo terrestre implementa la interfaz Motor. La clase vehículo acuático implementa la interfaz Vela.

La interfaz Motor tiene el siguiente método:

- calcularRevolucionesMotor(*int* fuerza, *int* radio): el número de revoluciones se calcula como la multiplicación de la fuerza del motor por su radio.

La interfaz Vela tiene los siguientes métodos:

- recomendarVelocidad(*int* velocidadViento): si la velocidad del viento es mayor a 80 km/h es muy alta, se recomienda no salir a navegar

y, por lo tanto, la velocidad actual debe ser cero. Si la velocidad del viento es menor a 10 km/h es muy baja y tampoco se recomienda salir a navegar.

Se debe definir un método *main* que cree una camioneta y una moto acuática e invoque los métodos de cada clase e imprima sus resultados en pantalla.

Solución

Clase: Vehículo

```
package Vehículos;

/**
 * Esta clase abstracta denominada Vehículo modela un medio de
 * locomoción que permite el traslado de un lugar a otro de personas o
 * cosas. Cuenta con atributos como velocidad actual y velocidad máxima.
 * @version 1.2/2020
 */
public abstract class Vehículo {
    int velocidadActual; /* Atributo que identifica la velocidad actual de
        un vehículo */
    int velocidadMáxima; /* Atributo que identifica la velocidad máxima
        permitida a un vehículo */

    /**
    * Constructor de la clase Vehículo
    * @param velocidadActual Parámetro que define la velocidad actual
    * de un vehículo
    * @param velocidadMáxima Parámetro que define la velocidad
    * máxima permitida a un vehículo
    */
    Vehículo(int velocidadActual, int velocidadMáxima) {
        this.velocidadActual = velocidadActual;
        this.velocidadMáxima = velocidadMáxima;
    }
```

```
    /**
     * Método que muestra en pantalla los datos de un vehículo
     */
    void imprimir() {
        System.out.println("Velocidad actual = " + velocidadActual);
        System.out.println("Velocidad máxima = " + velocidadMáxima);
    }

    /**
     * Método abstracto que permite incrementar la velocidad de un
     * vehículo
     * @param velocidad Parámetro que define el incremento de la
     * velocidad de un vehículo
     */
    abstract void acelerar(int velocidad);

    /**
     * Método abstracto que permite decrementar la velocidad de un
     * vehículo
     * @param velocidad Parámetro que define el decremento de la
     * velocidad de un vehículo
     */
    abstract void frenar(int velocidad);
}
```

Clase: Terrestre

```
package Vehículos;

/**
 * Esta clase denominada Terrestre modela un tipo de Vehículo que
 * funciona en el medio terrestre y que implementa la interfaz Motor.
 * @version 1.2/2020
 */
class Terrestre extends Vehículo implements Motor {

    /**
     * Constructor de la clase Terrestre
     * @param velocidadActual Parámetro que define la velocidad actual
     * de un vehículo terrestre
     * @param velocidadMáxima Parámetro que define la velocidad
     * máxima permitida para un vehículo terrestre
     */
```

```
    Terrestre(int velocidadActual, int velocidadMáxima) {
        // Invoca al constructor de la clase padre
        super(velocidadActual, velocidadMáxima);
    }

    /**
    * Implementación del método abstracto acelerar heredado de
    * Vehículo que permite incrementar la velocidad de un vehículo
    * terrestre
    * @param velocidad Parámetro que define el incremento de la
    * velocidad de un vehículo terrestre
    */
    void acelerar(int velocidad) {
        int vel = velocidadActual + velocidad;
        if (vel > velocidadMáxima) { /* La velocidad actualizada no puede
            superar la velocidad máxima */
            System.out.println("Supera la velocidad máxima permitida");
        } else { /* Si no supera la velocidad máxima, se actualiza la
            velocidad actual */
            velocidadActual = vel;
        }
    }

    /**
    * Implementación del método abstracto frenar heredado de Vehículo
    * que permite decrementar la velocidad de un vehículo terrestre
    * @param velocidad Parámetro que define el decremento de la
    * velocidad de un vehículo terrestre
    */
    void frenar(int velocidad) {
        int vel = velocidadActual - velocidad;
        if (vel < 0) { // La velocidad actualizada no puede ser negativa
            System.out.println("La velocidad no puede ser negativa");
        } else { /* Si la velocidad no se vuelve negativa, se actualiza la
            velocidad actual */
            velocidadActual = vel;
        }
    }
```

```
    /**
     * Implementación del método abstracto calcularRevolucionesMotor
     * heredado de Vehículo que calcula las revoluciones de un motor
     * como la multiplicación de su fuerza por su radio
     * @param fuerza Parámetro que define la fuerza del motor de un
     * vehículo
     * @param radio Parámetro que define el radio del motor de un vehículo
     */
    public int calcularRevolucionesMotor(int fuerza, int radio) {
        return (fuerza*radio);
    }
}
```

Interface: Motor

```
package Vehículos;

/**
 * Esta interfaz denominada Motor modela un motor que será
 * implementado por las clases Terrestre y Acuático
 * @version 1.2/2020
 */
interface Motor {

    /**
     * Método abstracto que permite calcular las revoluciones de un
     * motor a partir de la fuerza y radio del motor
     * @param fuerza Parámetro que define la fuerza del motor de un
     * vehículo
     * @param radio Parámetro que define el radio del motor de un
     * vehículo
     */
    int calcularRevolucionesMotor(int fuerza, int radio);
}
```

Interface: Vela

```
package Vehículos;

/**
 * Esta interfaz denominada Vela modela una superficie utilizada para
 * propulsar una embarcación mediante la acción del viento sobre ella.
 * La interfaz será implementada por la clase Acuático
 * @version 1.2/2020
 */
public interface Vela {

    /**
     * Método abstracto que recomienda una determinada velocidad del
     * vehículo de acuerdo a la velocidad del viento
     * @param velocidadViento Parámetro que define la velocidad del
     * viento que influenciará en la velocidad actual del vehículo
     */
    void recomendarVelocidad(int velocidadViento);
}
```

Clase: Acuático

```
package Vehículos;

/**
 * Esta clase denominada Acuático modela un tipo de Vehículo que
 * funciona en el medio acuático y que implementa las interfaces Motor
 * y Vela.
 * @version 1.2/2020
 */
public class Acuático extends Vehículo implements Motor, Vela {

    /**
     * Constructor de la clase Acuático
     * @param velocidadActual Parámetro que define la velocidad actual
     * de un vehículo acuático
     * @param velocidadMáxima Parámetro que define la velocidad
     * máxima permitida para un vehículo acuático
     */
    public Acuático(int velocidadActual, int velocidadMáxima) {
        // Invoca al constructor de la clase padre
        super(velocidadActual, velocidadMáxima);
    }
```

```
/**
* Implementación del método abstracto acelerar heredado de
* Vehículo que permite incrementar la velocidad de un vehículo
* acuático
* @param velocidad Parámetro que define el incremento de
* velocidad de un vehículo acuático
*/
void acelerar(int velocidad) {
    int vel = velocidadActual + velocidad;
    if (vel > velocidadMáxima) { /* La velocidad actualizada no puede
        superar la velocidad máxima */
        System.out.println("Supera la velocidad máxima permitida");
    } else { /* Si no supera la velocidad máxima, se actualiza la
        velocidad actual */
        velocidadActual = vel;
    }
}

/**
* Implementación del método abstracto frenar heredado de Vehículo
* que permite decrementar la velocidad de un vehículo acuático
* @param velocidad Parámetro que define el decremento de
* velocidad de un vehículo acuático
*/
void frenar(int velocidad) {
    int vel = velocidadActual – velocidad;
    if (vel < 0) { // La velocidad actualizada no puede ser negativa
        System.out.println("La velocidad no puede ser negativa");
    } else { /* Si la velocidad no se vuelve negativa, se actualiza la
        velocidad actual */
        velocidadActual = vel;
    }
}

/**
* Implementación del método abstracto calcularRevolucionesMotor
* heredado de Vehículo que calcula las revoluciones de un motor
* como la multiplicación de su fuerza por su radio
* @param fuerza Parámetro que define la fuerza que tiene el motor
* de un vehículo acuático
* @param radio Parámetro que define el radio de un motor de un
* vehículo acuático
*/
```

```
    public int calcularRevolucionesMotor(int fuerza, int radio) {
        return (fuerza*radio);
    }

    /**
    * Implementación de método abstracto recomendarVelocidad
    * proveniente de la interfaz Vela que recomienda una determinada
    * velocidad del vehículo de acuerdo a la velocidad del viento
    * @param velocidadViento Parámetro que define la velocidad del
    * viento que influenciará la velocidad actual del vehículo
    */
    public void recomendarVelocidad(int velocidadViento) {
        if ( velocidadViento > 80 || velocidadViento < 10) {
            velocidadActual = 0;
        }
    }
}
```

Clase: Prueba

```
package Vehículos;

/**
* Esta clase prueba diferentes acciones realizadas por las clases Terrestre
* y Acuático que son subclases de vehículos.
* @version 1.2/2020
*/
public class Prueba {

    /**
    * Método main que crea un vehículo terrestre y un vehículo
    * acuático. Además, prueba diferentes métodos de estas clases al
    * acelerar el vehículo terrestre; al calcular las revoluciones de motor
    * de un vehículo acuático; y al recomendar la velocidad de acuerdo
    * a la velocidad del viento.
    */
    public static void main(String args[]) {
        Terrestre camioneta = new Terrestre(100, 250);
        System.out.println("Camioneta");
        camioneta.imprimir();
        camioneta.acelerar(50);
        System.out.println("Nueva Velocidad actual= " + camioneta.
            velocidadActual);
```

```
            Acuático motoAcuática = new Acuático(50, 110);
            System.out.println("Moto acuática");
            motoAcuática.imprimir();
            System.out.println("Revoluciones del motor = " +
                    motoAcuática.calcularRevolucionesMotor(1200, 2));
            motoAcuática.recomendarVelocidad(20);
        }
    }
```

Diagrama de clases

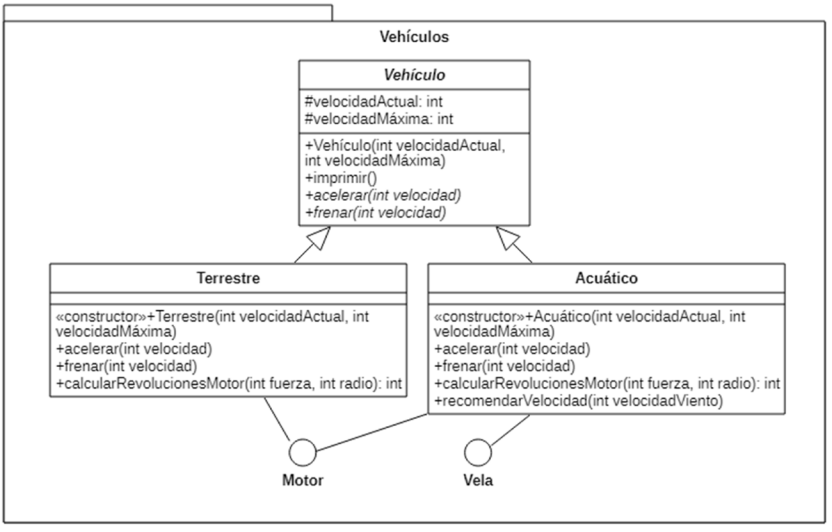


Figura 4.30. Diagrama de clases del ejercicio 4.11.

Explicación del diagrama de clases

Se ha definido un paquete denominado "Vehículos", el cual tiene una jerarquía de clases cuya clase raíz es la clase denominada "Vehículo". La clase Vehículo tiene dos subclases: Terrestre y Acuático. Estas relaciones de herencia se expresan en UML por medio de una línea continua que termina en un triángulo vinculado con la clase padre.

El diagrama también muestra dos interfaces: Motor y Vela, que se expresan por medio de una notación gráfica alternativa a la presentada en el ejercicio 4.10; en lugar de usar la notación de clase con el estereotipo <<*interface*>> se utiliza la notación de un círculo y una línea continua relacionada

con la clase que implementa la interfaz. En el diagrama, la clase Terrestre implementa la interfaz Motor y la clase Acuática implementa las interfaces Motor y Vela. Por lo tanto, estas clases deben implementar los métodos abstractos definidos en las interfaces, para la clase Terrestre se implementa el método calcularRevolucionesMotor y para la clase Acuático se implementan los métodos calcularRevolucionesMotor y recomendarVelocidad.

La clase *Vehículo* es una clase abstracta (presenta el nombre en cursiva) que posee dos atributos protegidos: la velocidad máxima y la velocidad actual (ambos de tipo entero). También cuenta con un constructor, un método imprimir y dos métodos abstractos: acelerar y frenar. Para identificar que son métodos abstractos sus nombres aparecen en cursiva.

La clase *Vehículo* tiene dos clases hijas: Terrestre y Acuático, las cuales heredan los atributos y métodos de la clase padre. Ambas clases implementan los métodos abstractos de la clase padre: acelerar y frenar.

Diagrama de objetos

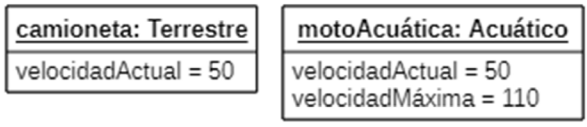


Figura 4.31. Diagrama de objetos del ejercicio 4.11.

Ejecución del programa

```
Camioneta
Velocidad actual = 100
Velocidad máxima = 250
Nueva Velocidad actual= 150
Moto acuática
Velocidad actual = 50
Velocidad máxima = 110
Revoluciones del motor = 2400
```

Figura 4.32. Ejecución del programa del ejercicio 4.11.

Ejercicios propuestos

- Agregar a la solución anterior una clase denominada VehículoAéreo, la cual tiene los métodos despegar, aterrizar y volar que muestran en pantalla la acción que están realizando.

  Agregar también dos nuevas interfaces:
  - Reactor que representa un motor de reacción. Esta interfaz tiene dos métodos encender y apagar.
  - Alas que representa las alas de un vehículo aéreo. Dicha interfaz tiene dos métodos soltar y subir tren de aterrizaje.

  La clase VehículoAéreo debe implementar estas interfaces. Los métodos encender y apagar de Reactor muestran en pantalla que el reactor está encendido y apagado, respectivamente. Los métodos soltar y subir tren de aterrizaje muestran en pantalla dichas acciones.

  En la clase de Prueba instanciar un vehículo aéreo e invocar los métodos heredados e implementados.

Ejercicio 4.12. Herencia de interfaces

Una interface puede heredar de otras interfaces, de igual manera que una clase hereda de otras clases. De la misma forma que en las clases, se utiliza la palabra reservada *extends* para determinar cuál interfaz hereda los métodos de la interfaz padre (Flanagan y Evans, 2019).

El formato de herencia de interfaces es el siguiente:

Interface interfazA extends interfazB, interfazC { … }

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Definir relaciones de herencia entre interfaces.
- Implementar el código de los métodos de las interfaces en clases específicas.

Enunciado: interfaz MatchDeportivo y sus interfaces heredadas

Se requiere desarrollar un programa que defina una interfaz denominada MatchDeportivo que represente un encuentro entre dos equipos deportivos. La interfaz cuenta con dos métodos: setEquipoLocal y setEquipoVisitante que no retornan nada, pero cada uno recibe como parámetro un *String* con el nombre del equipo.

La interfaz posee dos interfaces heredadas:

- Partido de fútbol: define dos nuevos métodos, setGolesEquipoLocal(*int* marcador) que permite asignar la cantidad de goles que realiza un equipo de fútbol que juega como local y setGolesEquipoVisitante(*int* marcador) para asignar la cantidad de goles que realiza un equipo de fútbol que juega como visitante. Ambos métodos no devuelven valores. La interfaz cuenta con un atributo que identifica la duración de un partido de fútbol, la cual tiene un valor de 90 minutos.
- Partido de baloncesto: define dos nuevos métodos, setCestasEquipoLocal(*int* marcador) que permite asignar la cantidad de cestas que realiza un equipo de baloncesto que juega como local y setCestasEquipoVisitante(*int* marcador) que permite asignar la cantidad de cestas que realiza un equipo de baloncesto que juega como visitante. Ambos métodos no devuelven valores. La interfaz cuenta con un atributo que identifica la duración de un partido de baloncesto, la cual tiene un valor de 40 minutos.

Se debe definir una clase PartidoFútbolLigaEspañola que implementa la interfaz Partido de fútbol, tiene los atributos: equipoLocal y equipoVisitante (ambos de tipo *String*) y golesEquipoLocal y golesEquipoVisitante (ambos de tipo *int*). Además, cuenta con métodos *get* y *set* para cada atributo y un método imprimirMarcador que *imprime* el marcador obtenido en el partido por los dos equipos.

Solución

Interface: MatchDeportivo

```
package Deportes;

/**
 * Esta interfaz denominada MatchDeportivo modela un encuentro
 * deportivo con dos métodos abstractos para establecer el equipo local
 * y el equipo visitante.
 * @version 1.2/2020
 */
public interface MatchDeportivo {
    /**
     * Método abstracto que establece el nombre del equipo local en un
     * encuentro deportivo
     * @param nombreEquipo Parámetro que define el nombre del
     * equipo local del encuentro deportivo
     */
    void setEquipoLocal(String nombreEquipo);

    /**
     * Método abstracto que establece el nombre del equipo visitante en
     * un encuentro deportivo
     * @param nombreEquipo Parámetro que define el nombre del
     * equipo visitante del encuentro deportivo
     */
    void setEquipoVisitante(String nombreEquipo);
}
```

Interface: PartidoFútbol

```
package Deportes;

/**
 * Esta interfaz denominada PartidoFútbol modela un encuentro
 * deportivo específico como un partido de fútbol. La interfaz hereda de
 * la interfaz padre MatchDeportivo. Tiene un atributo para la duración
 * del partido y define a su vez los métodos abstractos de la interfaz padre.
 * @version 1.2/2020
 */
```

```
package Deportes;

    /**
    * Esta interfaz denominada PartidoFútbol modela un encuentro
    * deportivo específico como un partido de fútbol. La interfaz hereda
    * de la interfaz padre MatchDeportivo. Tiene un atributo para la
    * duración del partido y define a su vez los métodos abstractos de la
    * interfaz padre.
    * @version 1.2/2020
    */
public interface PartidoFútbol extends MatchDeportivo {
    /* Atributo final que representa la duración de un partido de fútbol
        en minutos */
    static final int duraciónPartidoFútbol = 90;

    /**
    * Método abstracto que establece la cantidad de goles que marcó el
    * equipo local en el partido de fútbol
    * @param marcador Parámetro que define el marcador en goles del
    * equipo local en el partido de fútbol
    */
    void setGolesEquipoLocal(int marcador);

    /**
    * Método abstracto que establece la cantidad de goles que marcó el
    * equipo visitante en el partido de fútbol
    * @param marcador Parámetro que define el marcador en goles del
    * equipo visitante en el partido de fútbol
    */
    void setGolesEquipoVisitante(int marcador);
}
```

Interface: PartidoBaloncesto

```
package Deportes;

/**
* Esta interfaz denominada PartidoBaloncesto modela un encuentro
* deportivo específico como un partido de baloncesto. La interfaz
* hereda de la interfaz padre MatchDeportivo. Tiene un atributo para la
* duración del partido y define a su vez los métodos abstractos de la
* interfaz padre.
* @version 1.2/2020
*/
```

```
public interface PartidoBaloncesto extends MatchDeportivo {
    // Atributo final que representa la duración de un partido en minutos
    static final int duraciónPartidoBaloncesto = 40;

    /**
    * Método abstracto que establece la cantidad de cestas que marcó el
    * equipo local en el partido de baloncesto
    * @param marcador Parámetro que define el marcado obtenido en
    * cestas por el equipo local en el partido de baloncesto
    */
    void setCestasEquipoLocal(int marcador);

    /**
    * Método abstracto que establece la cantidad de cestas que marcó el
    * equipo visitante en el partido de baloncesto
    * @param marcador Parámetro que define el marcador obtenido en
    * cestas por el equipo visitante en el partido de baloncesto
    */
    void setCestasEquipoVisitante(int marcador);
}
```

Clase: PartidoFútbolLigaEspañola

```
package Deportes;

/**
* Esta clase denominada PartidoFútbolLigaEspañola modela un partido
* de fútbol de la liga española. La clase tiene atributos como el nombre
* del equipo local, el nombre del equipo visitante, la cantidad de goles
* marcados por el equipo local y la cantidad de goles marcados por el
* equipo visitante.
* @version 1.2/2020
*/
public class PartidoFútbolLigaEspañola implements PartidoFútbol {
    /* Atributo que identifica el nombre del equipo local en un partido de
       fútbol de la liga española */
    String equipoLocal;
    /* Atributo que identifica el nombre del equipo visitante en un partido
       de fútbol de la liga española */
    String equipoVisitante;
    /* Atributo que identifica la cantidad de goles realizados por el equipo
       local en un partido de fútbol de la liga española */
    int golesEquipoLocal;
```

```
/* Atributo que identifica la cantidad de goles realizados por el equipo
    visitante en un partido de fútbol de la liga española */
int golesEquipoVisitante;

/**
* Implementación del método abstracto heredado de la interfaz
* MatchDeportivo que establece el nombre del equipo local del
* partido de fútbol
* @param nombreEquipo Parámetro que define el nombre del
* equipo local del partido de fútbol
*/
public void setEquipoLocal(String nombreEquipo) {
    this.equipoLocal = nombreEquipo;
}

/**
* Implementación del método abstracto de la interfaz
* MatchDeportivo que establece el nombre del equipo visitante del
* partido de fútbol
* @param nombreEquipo Parámetro que define el nombre del
* equipo visitante del partido de fútbol
*/
public void setEquipoVisitante(String nombreEquipo) {
    this.equipoVisitante = nombreEquipo;
}

/**
* Implementación del método abstracto heredado de la interfaz
* PartidoFútbol que establece la cantidad de goles que marcó el
* equipo local en el partido de fútbol
* @param marcador Parámetro que define el marcador obtenido en
* goles por el equipo local en el partido de fútbol
*/
public void setGolesEquipoLocal(int marcador) {
    this.golesEquipoLocal = marcador;
}

/**
* Implementación del método abstracto heredado de la interfaz
* PartidoFútbol que establece la cantidad de goles que marcó el
* equipo visitante en el partido de fútbol
* @param marcador Parámetro que define el marcador obtenido en
* goles por el equipo visitante en el partido de fútbol
*/
```

```
    public void setGolesEquipoVisitante(int marcador) {
        this.golesEquipoVisitante = marcador;
    }

    /**
    * Método que muestra en pantalla el marcador de un partido de
    * fútbol de la liga española
    */
    void imprimirMarcador() {
        System.out.println("Equipo local: " + equipoLocal + ": " +
            golesEquipoLocal + " - Equipo visitante: " + equipoVisitante +
            ": " + golesEquipoVisitante);
    }
}
```

Clase: Prueba

```
package Deportes;

/**
* Esta clase prueba diferentes acciones realizadas por la clase
* PartidoFútbolLigaEspañola, la cual implementa varias interfaces
* @version 1.2/2020
*/
public class Prueba {

    /**
    * Método main que crea un partido de fútbol de la liga española y
    * establece un marcador para dicho partido, mostrándolo luego en
    * pantalla
    */
    public static void main(String args[]) {
        PartidoFútbolLigaEspañola partido = new
            PartidoFútbolLigaEspañola();
        System.out.println("Duración del partido =" +
                PartidoFútbolLigaEspañola.duraciónPartidoFútbol);
        partido.setEquipoLocal("Real Madrid");
        partido.setEquipoVisitante("Barcelona");
        partido.setGolesEquipoLocal(3);
        partido.setGolesEquipoVisitante(3);
        partido.imprimirMarcador();
    }
}
```

Diagrama de clases

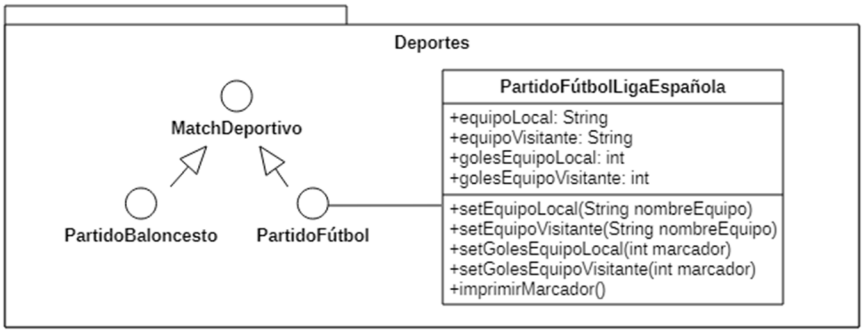


Figura 4.33. Diagrama de clases del ejercicio 4.12.

Explicación del diagrama de clases

Se ha definido un paquete denominado "Deportes", el cual presenta una jerarquía de interfaces con la interfaz raíz llamada MatchDeportivo y dos interfaces hijas: PartidoBaloncesto y PartidoFútbol. Las interfaces están relacionadas por medio de relaciones de herencia que se representan en UML con líneas continuas que finalizan con un triángulo en el extremo de la interfaz padre.

En el diagrama de clases se ha incluido una clase denominada PartidoFútbolLigaEspañola que implementa la interfaz PartidoFútbol. Al implementar esta interfaz, la clase PartidoFútbolLigaEspañola debe implementar los métodos setGolesEquipoLocal y setGolesEquipoVisitante (provenientes de la interfaz MatchDeportivo) y los métodos setEquipoLocal y setEquipoVisitante (provenientes de la interfaz PartidoFútbol).

La clase PartidoFútbolLigaEspañola tiene cuatro atributos: equipo local y visitante (de tipo *String*) y goles del equipo local y visitante (de tipo *int*). También cuenta con métodos *set* para establecer los valores de dichos atributos.

Diagrama de objetos

Figura 4.34. Diagrama de objetos del ejercicio 4.12.

Ejecución del programa

```
Duración del partido =90
Equipo local: Real Madrid: 3 - Equipo visitante: Barcelona: 3
```

Figura 4.35. Ejecución del programa del ejercicio 4.12.

Ejercicios propuestos

Agregar a la solución anterior, una clase PartidoBaloncestoLigaColombiana con nuevos métodos, que implementen los métodos requeridos por la interfaz PartidoBaloncesto. Modificar la clase de Prueba para instanciar un objeto de dicho tipo.

Capítulo 5
Relaciones de asociación, agregación y composición

El propósito general del quinto capítulo es entender que la programación orientada a objetos establece un modelado de la solución algorítmica como un conjunto de clases que se relacionan entre sí, enviándose mensajes entre ellas para cumplir los requisitos de un programa. En UML se establecen diferentes relaciones entre las clases de un programa y existen diferentes tipos de relaciones, cada una ellas con una sintaxis y semántica particular. A su vez, dicho modelado de relaciones se expresa en Java con unas estructuras de programación concretas. Este capítulo tiene siete ejercicios sobre relaciones de asociación, agregación y composición entre clases.

- **Ejercicio 5.1. Relación de asociación**
  El primer tipo de relación entre clases que se presenta es la relación de asociación, la cual es genérica y tiene una notación gráfica particular junto con características como multiplicidad, roles y navegabilidad. Para entender el concepto de asociación entre clases se plantea el primer ejercicio.

- **Ejercicio 5.2. Relación de composición**
  Otro tipo de relación entre clases, con un significado más específico, es la relación de composición. Dicha relación expresa una fuerte relación entre el todo y sus partes, es decir, si el todo se destruye, sus partes contenedoras también. El

segundo ejercicio está orientado a entender y aplicar la relación de composición entre clases.

- **Ejercicio 5.3. Composición con partes múltiples**
  Una clase que representa el todo puede estar conformada por múltiples partes, es decir, un objeto está conformado por varias partes. El tercer ejercicio propone un problema que usa las relaciones de composición con partes múltiples.

- **Ejercicio 5.4. Composición múltiple**
  A su vez, una clase que representa el todo puede estar conformada por múltiples partes y dichas partes puede aparecer instanciadas más de una vez. El cuarto ejercicio presenta un problema que implementa la composición múltiple entre clases.

- **Ejercicio 5.5. Relación de agregación**
  Otra relación entre clases, similar a la relación de composición, pero con una semántica de la relación más débil, es la relación de agregación. En este tipo de relación, las partes pueden seguir existiendo pese a que el todo en que están incluidas llegue a ser destruido. El quinto ejercicio presenta un problema para aplicar este concepto.

- **Ejercicio 5.6. Diferencias entre agregación y composición**
  Es necesario aclarar las diferencias sintácticas y semánticas entre las relaciones de agregación y composición. El sexto ejercicio propone un problema para entender las diferencias entre estos dos tipos de relaciones entre clases.

- **Ejercicio 5.7. Significado de la relación de agregación**
  El último ejercicio propone un enunciado adicional para implementar correctamente la relación de agregación entre clases, que es diferente de las relaciones de asociación y composición.

Los diagramas UML utilizados en este capítulo son los diagramas de clase y de objetos. Los paquetes incorporados en los diagramas de clase permiten agrupar elementos UML y en los ejercicios se organizarán las clases en un único paquete debido a la pequeña cantidad de clases que tienen las soluciones presentadas. Los diagramas de clase modelan la estructura estática de los programas. Así, los diagramas de clase de los ejercicios presentados,

generalmente, estarán conformados por una o varias clases relacionadas por medio de relaciones de asociación, agregación o composición e incorporan notaciones para identificar tipos de atributos y métodos. Los diagramas de objeto identifican los objetos creados en el método *main* del programa.

Ejercicio 5.1. Relación de asociación

La asociación se refiere a la relación entre dos clases que se establece a través de sus objetos. Esta define cómo los objetos están relacionados entre sí y cómo cada uno está utilizando su funcionalidad. Cada objeto tiene su propio ciclo de vida y no tiene ningún propietario. Los objetos asociados se pueden crear y eliminar de forma independiente.

La relación de asociación indica que una clase conoce y mantiene una referencia a otra clase. La relación puede ser bidireccional con cada clase sosteniendo una referencia a la otra (Booch, Rumbaugh y Jacobson, 2017). Si dos clases están asociadas en forma unidireccional, en la clase donde se origina la dirección (navegabilidad de la asociación) existirá un atributo con referencia a la otra clase. Así:

```
class A {
        ClassB objetoClaseB;
}
```

El valor de multiplicidad máxima en un extremo de una asociación puede ser cualquier número entero mayor o igual que 1, aunque los valores más comunes son 1 o * (Booch, Rumbaugh y Jacobson, 2017). Cuando el valor es 1, la asociación es única para la clase en el extremo opuesto, cuando el valor es igual o mayor a 2, se dice que es múltiple.

Por otro lado, las asociaciones individuales son más fáciles de implementar que las asociaciones múltiples. Para almacenar la única instancia posible de una asociación única, usualmente, se emplea un atributo que referencia a la clase objetivo, pero para implementar una asociación múltiple se debe utilizar algún tipo de colección de objetos (*arrays* o vectores).

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Identificar y modelar relaciones de asociación entre clases.
- Implementar relaciones de asociación a través de código.

Enunciado: clase Peaje y jerarquía de herencia de vehículos

Se requiere desarrollar un programa que modele un sistema de peaje. Las estaciones de peaje tienen un nombre, departamento en que están ubicadas y un valor numérico que representa el valor total de peaje recolectado.

Los vehículos que llegan a un peaje tienen una placa (tipo *String*). El peaje cobra diferentes valores de peaje según el tipo de vehículo. Si es un carro, el valor del peaje es de $10 000. Si es una moto, $5000. Si es un camión, el valor del peaje depende del número de ejes, se cobra $5000 por cada eje.

Se requiere que la estación de peaje calcule el valor del peaje de cada vehículo que llegue y el total de peajes recolectados. Así, al finalizar, el sistema debe *imprimir* en pantalla un listado con los vehículos que llegaron al peaje y el total acumulado.

Solución

Clase: Vehículo

```
package SistemaPeaje;

/**
 * Esta clase abstracta denominada Vehículo modela diferentes tipos de
 * vehículos que pueden llegar a un peaje. Un vehículo posee como
 * único atributo su placa.
 * @version 1.2/2020
 */
abstract public class Vehículo {
    String placa; // Atributo que define la placa de un vehículo

    /**
     * Constructor de la clase Vehículo
     * @param placa Parámetro que define la placa de un vehículo
     */
```

```
    Vehículo(String placa) {
        this.placa = placa;
    }
}
```

Clase: Carro

```
package SistemaPeaje;

/**
 * Esta clase denominada Carro modela un tipo específico de Vehículo
 * que llega a un peaje. Tiene un atributo estático que representa su
 * valor del peaje en $10000.
 * @version 1.2/2020
 */
public class Carro extends Vehículo {
    // Atributo estático que identifica el valor de peaje a pagar por un carro
    static int valorPeaje = 10000;

    /**
    * Constructor de la clase Carro
    * @param placa Parámetro que define la placa de un carro
    */
    Carro(String placa) {
        super(placa); // Invoca al constructor de la clase padre
    }

    /**
    * Método que devuelve el valor del peaje para un carro
    * @return El valor del peaje para un carro
    */
    public int getValorPeaje() {
        return valorPeaje;
    }

    /**
    * Método que establece el valor del peaje para un carro
    * @param Parámetro que define el valor del peaje para un carro
    */
    static void setValorPeaje(int valorPeaje) {
        valorPeaje = valorPeaje;
    }
```

```
    /**
     * Método que muestra en pantalla la placa y el valor del peaje de un
     * carro
     */
    void imprimit() {
        System.out.println("Placa = " + placa);
        System.out.println("Valor del peaje = " + valorPeaje);
    }
}
```

Clase: Moto

```
package SistemaPeaje;

/**
 * Esta clase denominada Moto modela un tipo específico de Vehículo
 * que llega a un peaje. Tiene un atributo estático que representa su
 * valor del peaje en $5000.
 * @version 1.2/2020
 */
public class Moto extends Vehículo {
    // Atributo estático que identifica el valor de peaje a pagar por una moto
    static int valorPeaje = 5000;

    /**
     * Constructor de la clase Moto
     * @param placa Parámetro que define la placa de una moto
     */
    Moto(String placa) {
        super(placa); // Invoca al constructor de la clase padre
    }

    /**
     * Método que devuelve el valor del peaje para una moto
     * @return El valor del peaje para una moto
     */
    public int getValorPeaje() {
        return valorPeaje;
    }
```

```
    /**
     * Método que establece el valor del peaje para una moto
     * @param Parámetro que define el valor del peaje para una moto
     */
    static void setValorPeaje(int valorPeaje) {
        valorPeaje = valorPeaje;
    }

    /**
     * Método que muestra en pantalla la placa y el valor del peaje de
     * una moto
     */
    void imprimit() {
        System.out.println("Placa = " + placa);
        System.out.println("Valor del peaje = " + valorPeaje);
    }
}
```

Clase: Camión

```
package SistemaPeaje;

/**
 * Esta clase denominada Camión modela un tipo específico de Vehículo
 * que llega a un peaje. Tiene un atributo para representar el número de
 * ejes que tiene un camión y un atributo estático que representa su
 * valor del peaje en $5000 por cada eje.
 * @version 1.2/2020
 */
public class Camión extends Vehículo {
    // Atributo estático que identifica el valor de peaje a pagar por un camión
    static int valorPeajeEje = 5000;
    // Atributo que identifica el número de ejes que tiene un camión
    int númeroEjes;

    /**
     * Constructor de la clase Camión
     * @param placa Parámetro que define la placa de un camión
     * @param númeroEjes Parámetro que define el número de ejes de
     * un camión
     */
```

```
    Camión(String placa, int númeroEjes) {
        super(placa); // Invoca al constructor de la clase padre
        this.númeroEjes = númeroEjes;
    }

    /**
    * Método que devuelve el valor del peaje para un camión
    * @return El valor del peaje para un camión
    */
    public int getValorPeajeEje() {
        return valorPeajeEje;
    }

    /**
    * Método que establece el valor del peaje para un camión
    * @param Parámetro que define el valor del peaje para un camión
    */
    static void setValorPeajeEje(int valorPeajeEje) {
        valorPeajeEje = valorPeajeEje;
    }

    /**
    * Método que muestra en pantalla la placa, el número de ejes y el
    * valor del peaje de un camión
    */
    void imprimit() {
        System.out.println("Placa = " + placa);
        System.out.println("Número de ejes = " + númeroEjes);
        System.out.println("Valor del peaje = " + valorPeajeEje);
    }
}
```

Clase: Peaje

```
package SistemaPeaje;
import java.util.*;

/**
 * Esta clase denominada Peaje modela una estación de peaje de una
 * carretera. Tiene los atributos nombre de la estación de peaje, el
 * departamento al que pertenece la estación, un vector de Vehículos
 * que llegan a la estación y total de dinero en peajes recibido. Además
 * cuenta con tres atributos estáticos para calcular el total de carros,
 * motos y camiones que llegan al peaje.
```

```
 * @version 1.2/2020
 */
public class Peaje {
    // Atributo que identifica el nombre de una estación de peaje
    String nombre;
    /* Atributo que identifica el nombre del departamento donde está
       ubicada la estación de peaje */
    String departamento;
    /* Atributo que identifica el conjunto de vehículos que llega a la
       estación de peaje */
    Vector vehículos;
    /* Atributo que identifica el total de dinero recolectado por la estación
       de peaje */
    int totalPeaje = 0;

    /* Atributo que identifica el total de camiones que llegó a la estación
       de peaje */
    static int totalCamiones = 0;
    /* Atributo que identifica el total de motos que llegó a la estación de
       peaje */
    static int totalMotos = 0;
    /* Atributo que identifica el total de carros que llegó a la estación de
       peaje */
    static int totalCarros = 0;

    /**
    * Constructor de la clase Peaje
    * @param nombre Parámetro que define el nombre de la estación de
    * peaje
    * @param departamento Parámetro que define el departamento
    * donde se encuentra localizado el peaje
    */
    Peaje(String nombre, String departamento) {
        this.nombre = nombre;
        this.departamento = departamento;
        vehículos = new Vector(); // Crea el vector de vehículos
    }
```

```
/**
* Método que devuelve el nombre de la estación de peaje
* @return El nombre de la estación de peaje
*/
public String getNombre() {
    return nombre;
}

/**
* Método que establece el nombre de la estación de peaje
* @param nombre Parámetro que define el nombre de la estación de
* peaje
*/
private void setNombre(String nombre) {
    this.nombre = nombre;
}

/**
* Método que devuelve el departamento donde está localizada la
* estación de peaje
* @return El departamento donde está localizada la estación de
* peaje
*/
public String getDepartamento() {
    return departamento;
}

/**
* Método que establece el departamento donde está localizada la
* estación de peaje
* @param departamento Parámetro que define el departamento
* donde está localizada la estación de peaje
*/
private void setDepartamento(String departamento) {
    this.departamento = departamento;
}

/**
* Método que permite añadir un vehículo al vector de vehículos de
* la estación de peaje
* @param vehículo Parámetro que define el vehículo a agregar al
* vector de vehículos de la  estación de peaje
```

```
*/
public void añadirVehículo(Vehículo vehículo) {
    vehículos.add(vehículo);
}

/**
* Método que permite calcular el peaje de un vehículo que llega a la
* estación de peaje
* @param Parámetro que define el vehículo que llega a la estación
* de peaje
*/
public int calcularPeaje(Vehículo vehículo) {
    if (vehículo instanceof Carro) { /* Si el vehículo que llegó es un
        carro */
        totalCarros++; // Actualiza el total de carros que llega al peaje
        totalPeaje += Carro.valorPeaje; /* Actualiza el total de dinero
            del peaje */
        return Carro.valorPeaje; // Retorna el dinero pagado por un carro
    } else if (vehículo instanceof Moto) { /* Si el vehículo que llegó es
        una moto */
        totalMotos++;  // Actualiza el total de motos que llega al peaje
        totalPeaje += Moto.valorPeaje; /* Actualiza el total de dinero
            del peaje */
        return Moto.valorPeaje; // Retorna el dinero pagado por una moto
    } else if (vehículo instanceof Camión) { /* Si el vehículo que llegó
        es un camión */
        totalCamiones++;  /* Actualiza el total de camiones que llega
            al peaje */
        Camión camión = (Camión) vehículo; /* Obtiene un objeto
            Camión */
        /* Calcula el peaje del camión y actualiza total de dinero del
            peaje */
        totalPeaje += camión.númeroEjes * camión.valorPeajeEje;
        // Retorna el dinero pagado por un camión
        return camión.númeroEjes * camión.valorPeajeEje;
    } else return -1; // Si llega otro tipo de objeto
}

/**
* Método que muestra en pantalla los datos del peaje, el total de
* vehículos que llegó al peaje discriminado por tipo y el total de
* dinero recaudado por la estación de peaje
*/
```

```
    public void imprimir() {
        System.out.println("Peaje = " + getNombre());
        System.out.println("Ubicación = " + getDepartamento());
        System.out.println("Total de carros = " + totalCarros);
        System.out.println("Total de motos = " + totalMotos);
        System.out.println("Total de camiones = " + totalCamiones);
        int totalVehículos = totalCarros + totalMotos +totalCamiones;
        System.out.println("Total de vehículos = " + totalVehículos);
        System.out.println("Dinero total = $" + totalPeaje);
    }
}
```

Clase: Prueba

```
package SistemaPeaje;

/**
 * Esta clase prueba diferentes acciones realizadas por una estación de
 * peaje donde llegan diferentes tipos de vehículos y se calcula el total
 * de dinero recaudado.
 * @version 1.2/2020
 */
public class Prueba {

    /**
     * Método main que crea una estación de peaje, luego se van
     * agregando diferentes tipos de vehículos al vector de vehículos de
     * la estación de peaje. Para finalizar, se calcula el total de dinero
     * recaudado por la estación de peaje.
     */
    public static void main(String args[]) {
        Peaje peaje = new Peaje ("La Lizama", "Santander");
        Camión camión1 = new Camión("PKI-889", 2);
        peaje.añadirVehículo(camión1);
        peaje.calcularPeaje(camión1);
        Camión camión2 = new Camión("KLM-123", 3);
        peaje.añadirVehículo(camión2);
        peaje.calcularPeaje(camión2);
        Camión camión3 = new Camión("PQI-234", 4);
        peaje.añadirVehículo(camión3);
        peaje.calcularPeaje(camión3);
        Moto moto1 = new Moto("ABC-123");
        peaje.añadirVehículo(moto1);
```

```
            peaje.calcularPeaje(moto1);
            Moto moto2 = new Moto("XYZ-000");
            peaje.añadirVehículo(moto2);
            peaje.calcularPeaje(moto2);
            Carro carro1 = new Carro("WVC-389");
            peaje.añadirVehículo(carro1);
            peaje.calcularPeaje(carro1);
            Carro carro2 = new Carro("QWE-543");
            peaje.añadirVehículo(carro2);
            peaje.calcularPeaje(carro2);
            peaje.imprimir();
        }
    }
```

Diagrama de clases

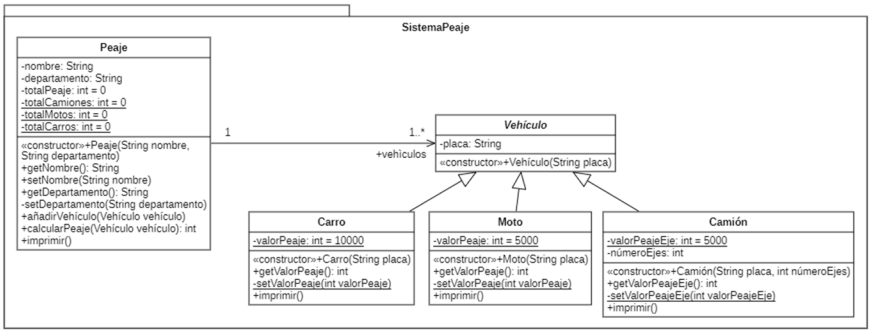


Figura 5.1. Diagrama de clases del ejercicio 5.1.

Explicación del diagrama de clases

Se ha definido un paquete denominado "SistemaPeaje", el cual presenta una jerarquía de clases con una clase raíz denominada *Vehículo*, la cual es abstracta (su nombre se encuentra en cursiva) por ello no habrá instancias de dicha clase porque es muy genérica. Un vehículo cuenta con un único atributo: la placa que lo identifica. Hay tres tipos de vehículos: carro, moto y camión, que sí pueden ser instanciados. Las relaciones de herencia se expresan en UML con líneas continuas que finalizan con un triángulo; la clase adyacente al triángulo representa la clase padre y en el otro extremo se encuentra la hija.

Cada subclase tiene un atributo estático que representa el valor del peaje para dicho tipo de vehículo. En UML, el texto de un atributo estático está subrayado. Cada subclase posee un constructor y métodos para obtener y establecer el valor del peaje (*get* y *set*), además, también tiene un método para mostrar los datos del vehículo en pantalla (*imprimir*). Como el método *set* de valor del peaje cambia un atributo estático, a su vez debe ser estático; por ello, se identifica con el texto subrayado en las tres clases hijas.

La clase Peaje contiene tanto atributos de instancia como estáticos. Los atributos estáticos son compartidos por todos los objetos de la clase. Como ya se ha mencionado, los atributos estáticos para un peaje se identifican porque están subrayados y para el ejercicio son los totales de camiones, motos y carros que llegan al peaje. El resto de los atributos son de instancia: nombre del peaje, departamento del peaje y total de peaje.

La clase Peaje está asociada con la clase *Vehículo* mediante una línea continua que expresa una relación de asociación UML y a nivel de código establecerá un atributo denominado "vehículos" que será un vector de elementos de tipo Vehículo. La multiplicidad de la asociación presentada es de uno a muchos, es decir, que por lo menos, habrá un vehículo en el vector y podrán existir muchos vehículos. Como un vehículo tiene varios tipos, el vector de vehículos puede contener objetos de tipo carro, moto o camión.

Es importante identificar que, en primer lugar, mediante los métodos añadirVehículo de la clase Peaje se van agregando vehículos al vector. En segundo lugar, el método calcularPeaje determina el valor a pagar por un vehículo que llega al peaje. Finalmente, también cuenta con un método para imprimir los datos del peaje en pantalla.

Diagrama de objetos

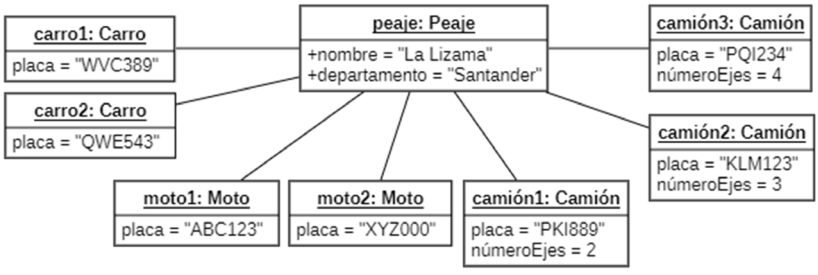


Figura 5.2. Diagrama de objetos del ejercicio 5.1.

Ejecución del programa

```
Peaje = La Lizama
Ubicación = Santander
Total de carros = 2
Total de motos = 2
Total de camiones = 3
Total de vehículos = 7
Dinero total = $75000
```

Figura 5.3. Ejecución del programa del ejercicio 5.1.

Ejercicios propuestos

- Modificar el ejercicio anterior para modelar que una persona conduce un vehículo. La persona tiene los siguientes atributos: nombre, apellidos, número de documento de identidad y fecha de nacimiento. Se requieren métodos para:
  - Asignar un vehículo a una persona. Se debe tener en cuenta que una persona puede tener asignado varios vehículos de diferente tipo.
  - Desasignar un vehículo a una persona. Previamente, es necesario verificar que la persona tiene un vehículo asignado.
  - Al dar el nombre de una persona, se debe generar el total de dinero pagado en peajes por cada vehículo que esta persona tenga asignado.

Ejercicio 5.2. Relación de composición

La relación de composición es un tipo especial de asociación entre clases que representa una fuerte relación entre un todo con sus partes constituyentes. Si la clase que representa el todo se elimina, las clases que las componen también se eliminarán en el sistema (Seidl *et al*., 2015).

En Java, la relación de composición se modela dentro de la clase que representa el todo. Las partes constituyentes deben ser atributos de la clase. Además, dentro del constructor de la clase que representa el todo, se deben instanciar los objetos que representan las partes y asignarlos a sus respectivos atributos.

```
class Todo {
        Parte parte1, parte2;
        Todo() {
                parte1 = new Parte();
                parte2 = new Parte();
        }
}
```

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Definir relaciones de composición entre clases, identificando las clases que representan el todo (contenedoras) y sus partes constituyentes.
- Instanciar las partes de la clase contenedora dentro del constructor de la clase que representa el todo.

Enunciado: clase Círculo y clase Punto

Un Círculo se describe por medio de un punto central y su radio. A su vez un Punto se define por medio de una coordenada *x* y una coordenada *y*. Se requiere que ambas clases tengan sus métodos *get* y *set*.

La clase Círculo debe tener un método imprimir que muestre en pantalla el siguiente mensaje "El centro del círculo es la coordenada *(x, y)*, y tiene un radio de *z*.

Se debe generar una clase de prueba donde en su método *main* se crea un círculo e invoca su método imprimir.

Solución

Clase: Círculo

```
package Composición;
/**
 * Esta clase denominada Círculo modela este tipo de figura geometrica
 * que se caracteriza por tener un centro y un radio como atributos.
 * @version 1.2/2020
 */
```

```
public class Círculo {
    /* Atributo que identifica el punto central donde se encuentra
       localizado un círculo */
    Punto centro;
    int radio; // Atributo que identifica el radio de un círculo

    /**
    * Constructor de la clase Círculo
    * @param x Parámetro que define la coordenada x de un punto que
    * establece el centro del círculo
    * @param y Parámetro que define la coordenada y de un punto que
    * establece el centro del círculo
    * @param radio Parámetro que define el radio del círculo
    */
    public Círculo(int x, int y, int radio) {
        centro = new Punto(x,y); /* Crea un círculo con los valores
            pasados como parámetros */
        this.radio = radio; // Inicializa el atributo radio
    }

    /**
    * Método que devuelve el radio de un círculo
    * @return El radio de un círculo
    */
    int getRadio() {
        return radio;
    }

    /**
    * Método que establece el radio de un círculo
    * @param Parámetro que define el radio de un círculo
    */
    void setRadio(int radio) {
        this.radio = radio;
    }

    /**
    * Método que muestra en pantalla los datos de un círculo
    */
    void imprimir() {
```

```
            System.out.println("El centro del círculo es la coordenada (" +
                centro.getX() + "," + centro.getY() + ") y tiene un radio = " +
                radio);
        }
    }
```

Clase: Punto

```
package Composición;

/**
 * Esta clase denominada Punto que está conformado por una pareja de
 * valores, un valor para la coordenada x y otro para la coordenada y.
 * @version 1.2/2020
 */
public class Punto {
    int x, y; /* Atributos que identifican el valor de las coordenadas (x,y)
        del círculo */

    /**
    * Constructor de la clase Punto
    * @param x Parámetro que define la coordenada x de un punto
    * @param y Parámetro que define la coordenada y de un punto
    */
    Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }

    /**
    * Método que devuelve el valor de la coordenada x de un punto
    * @return El valor de la coordenada x de un punto
    */
    int getX() {
        return x;
    }

    /**
    * Método que establece el valor de la coordenada x de un punto
    * @param Parámetro que define el valor de la coordenada x de un punto
    */
```

```
    void setX(int x) {
        this.x = x;
    }
    /**
    * Método que devuelve el valor de la coordenada y de un punto
    * @return El valor de la coordenada y de un punto
    */
    int getY() {
        return y;
    }
    /**
    * Método que establece el valor de la coordenada y de un punto
    * @param Parámetro que define el valor de la coordenada y de un
    * punto
    */
    void setY(int y) {
        this.y = y;
    }
}
```

Clase: Prueba

```
package Composición;
/**
 * Esta clase prueba la clase Círculo
 * @version 1.2/2020
 */
public class Prueba {
    /**
    * Método main que crea un círculo localizado en el punto (2,3) y
    * con un radio de 5. Luego, imprime los datos del círculo en
    * pantalla.
    */
    public static void main(String args[]) {
        Círculo círculo1 = new Círculo(2,3,5);
        círculo1.imprimir();
    }
}
```

Diagrama de clases

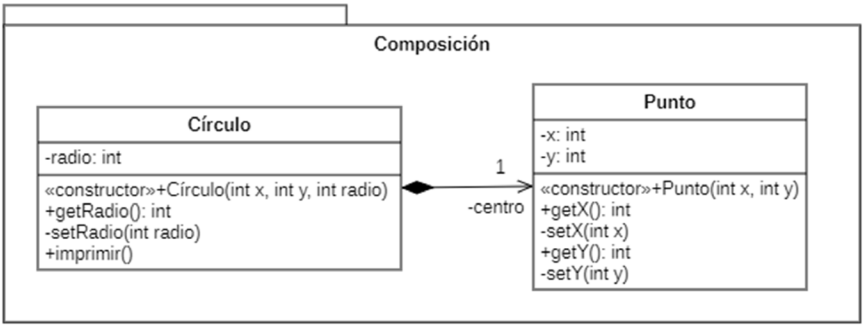


Figura 5.4. Diagrama de clases del ejercicio 5.2.

Explicación del diagrama de clases

Se ha definido un paquete denominado "Composición", que contiene dos clases: Círculo y Punto. La clase Círculo tiene como atributo un radio y otro atributo denominado centro, el cual como es un objeto de tipo Punto, se representará como una relación de composición con dicha clase.

La relación de composición se representa como una línea continua, en un extremo hay un rombo negro. La clase adyacente al rombo es contenedora (representa al todo) y la clase en el otro extremo representa la parte. La relación de composición expresa una fuerte relación entre las dos clases, es decir, si la clase Círculo se elimina, el Punto contenido dentro de la clase también se elimina. Por lo tanto, el punto no puede existir sin un círculo donde esté contenido.

La multiplicidad de la composición es 1, es decir, que un círculo tiene un único punto (su centro). La punta en flecha de la relación de composición es la navegabilidad de la relación, en este ejemplo, significa que el círculo conoce el punto que es su centro, pero el punto desconoce el círculo al cual pertenece.

Ambas clases presentan un constructor y métodos *get* y *set* de sus respectivos atributos.

Diagrama de objetos

Figura 5.5. Diagrama de objetos del ejercicio 5.2.

Ejecución del programa

```
El centro del círculo es la coordenada (2,3) y tiene un radio = 5
```

Figura 5.6. Ejecución del programa del ejercicio 5.2.

Ejercicios propuestos

- Modificar el programa anterior para incluir las siguientes clases:
  - Una clase Recta que contenga exactamente 2 puntos. Se debe definir los siguientes métodos para dicha clase:
    - Un método que calcule la pendiente de la recta.
    - Un método que calcule la longitud de la recta.
  - Una clase Polígono que tenga un nombre y este conformada por mínimo 3 puntos. El orden de los puntos se tiene en cuenta en la especificación del polígono.

Ejercicio 5.3. Composición con partes múltiples

La relación de composición también se puede manifestar con casos donde la parte que representa el todo está compuesto de varias partes. Por lo tanto, el todo debe tener como atributo una colección de partes y en su constructor se va creando cada parte y, estas, a su vez, se van agregando a la colección de partes. El constructor del todo recibe como parámetros la información necesaria para crear cada parte (Hunt, 2013). El formato para la composición con partes múltiples es el siguiente:

```
class Todo  {
Parte colecciónPartes;
        Public Todo(argumentos de las partes) {
                lasPartes = new Parte[númeroPartes];
                for (int i = 0; i < númeroPartes; i++) {
                        lasPartes[i] = new Parte(argumentos de la parte);
                }
        }
}
```

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Definir relaciones de composición donde la clase contenedora tiene muchas partes.
- Instanciar las partes de la clase contenedora dentro del constructor de la clase que representa el todo.

Enunciado: clase Biblioteca y clase Libro

Una biblioteca tiene un nombre (tipo *String*) y una colección de libros. Se requiere que la biblioteca tenga un constructor que reciba como parámetros el nombre de la biblioteca e inicialice la colección de libros. La biblioteca posee los siguientes métodos:

- Añadir libro que permite agregar un libro a la colección de libros de la biblioteca.
- Listar libros que muestra en pantalla los datos de todos los libros de la biblioteca invocando el método imprimir de los libros.

Cada libro tiene los siguientes datos: título (tipo *String*), autor (tipo *String*), año de publicación (tipo *int*), editorial (tipo *String*), referencia bibliográfica (tipo *String*). El libro debe tener un constructor que inicialice sus atributos y un método *imprimir* que muestre los valores de sus atributos en pantalla. Se requiere implementar una clase de prueba que instancie una biblioteca denominada "Biblioteca Nacional" y añada a la biblioteca los siguientes libros de la tabla 5.1.

Tabla 5.1. Libros de la biblioteca

Título del libro	Autor	Año	Editorial	Ref. Bibliográfica
Cien años de soledad	Gabriel García Márquez	1967	Sudamericana	858.67/M566
Rayuela	Julio Cortázar	1963	Sudamericana	863.55/J667
La tía Julia y el escribidor	Mario Vargas Llosa	1977	Seix Barral	868.23/L567

Luego, se deben imprimir en pantalla la colección de libros de la biblioteca.

Solución

Clase: Biblioteca

```
package Biblio;
import java.util.*;

/**
 * Esta clase denominada Biblioteca modela este tipo de establecimiento,
 * el cual tiene un nombre y una colección de libros.
 * @version 1.2/2020
 */
public class Biblioteca {
    // Atributo que identifica el nombre de la biblioteca
    String nombre;
    /* Atributo que identifica la colección de libros de la biblioteca como
       un vector de libros */
    Vector colecciónLibros;

    /**
    * Constructor de la clase Biblioteca
    * @param nombre Parámetro que define el nombre de la biblioteca
    */
    Biblioteca(String nombre) {
        this.nombre = nombre;
        colecciónLibros = new Vector(); // Crea el vector de libros
    }

    /**
    * Método que permite añadir un libro a la biblioteca
    * @param libro Parámetro que define un libro que se agregará a la
    * biblioteca
    */
```

```
    void añadirLibro(Libro libro) {
        colecciónLibros.add(libro); /* Se agrega el libro al vector de libros
            de la biblioteca */
    }

    /**
    * Método que muestra en pantalla los datos de los libros de la biblioteca
    */
    void listarLibros() {
        // Se recorre el vector de libros
        for (int i= 0; i < colecciónLibros.size(); i++) {
            /* Se extrae un elemento del vector en la posición i. Se debe
                realizar un casting para extraer un objeto Libro */
            Libro libro = (Libro) colecciónLibros.elementAt(i);
            libro.imprimir();
        }
    }
}
```

Clase: Libro

```
package Biblio;

/**
 * Esta clase denominada Libro modela los libros que tiene la biblioteca.
 * Cada libro contiene un título, autor, año de publicación, editorial y
 * referencia bibliográfica.
 * @version 1.2/2020
 */
public class Libro {
    String título; // Atributo que identifica el título de un libro
    String autor; // Atributo que identifica el autor de un libro
    int añoPublicación; /* Atributo que identifica el año de publicación
        de un libro */
    String editorial; // Atributo que identifica la editorial que publicó el libro
    String referenciaBibliográfica; /* Atributo que identifica la referencia
        bibligráfica del libro */

    /**
    * Constructor de la clase libro
    * @param título Parámetro que define el título de un libro
    * @param autor Parámetro que define el autor de un libro
```

```
     * @param añoPublicación Parámetro que define el año de
     * publicación de un libro
     * @param editorial Parámetro que define la editorial de un libro
     * @param referenciaBibliográfica Parámetro que define la referencia
     * bibliográfica de un libro
     */
    Libro(String título, String autor, int añoPublicación, String editorial,
            String referenciaBibliográfica) {
        this.título = título;
        this.autor = autor;
        this.añoPublicación = añoPublicación;
        this.editorial = editorial;
        this.referenciaBibliográfica = referenciaBibliográfica;
    }

    /**
     * Método que muestra en pantalla los datos de un libro
     */
    void imprimir() {
        System.out.println("Título del libro = " + título);
        System.out.println("Autor = " + autor);
        System.out.println("Año de publicación = " + añoPublicación);
        System.out.println("Editorial = " + editorial);
        System.out.println("Referencia bibliográfica = " +
            referenciaBibliográfica);
        System.out.println();
    }
}
```

Clase: Prueba

```
package Biblio;

/**
 * Esta clase prueba diferentes acciones realizadas en una biblioteca.
 * @version 1.2/2020
 */
public class Prueba {

    /**
     * Método main que crea una biblioteca y luego se crean varios libros
     * que serán agregados al vector de libros de la biblioteca. Al finalizar,
     * se listan los libros de la biblioteca.
```

```
        */
        public static void main(String args[]) {
            Biblioteca biblioteca1 = new Biblioteca("Biblioteca Nacional");
            Libro libro1 = new Libro("Cien años de soledad", "Gabriel García
                Márquez", 1967, "Sudamericana","858.67/M566");
            biblioteca1.añadirLibro(libro1);
            Libro libro2 = new Libro("Rayuela","Julio
                Cortázar",1963,"Sudamericana","863.55/J667");
            biblioteca1.añadirLibro(libro2);
            Libro libro3 = new Libro("La tía julia y el escribidor", "Mario
                Vargas Llosa", 1977, "Seix Barral","868.23/L567");
            biblioteca1.añadirLibro(libro3);
            biblioteca1.listarLibros();
        }
    }
```

Diagrama de clases

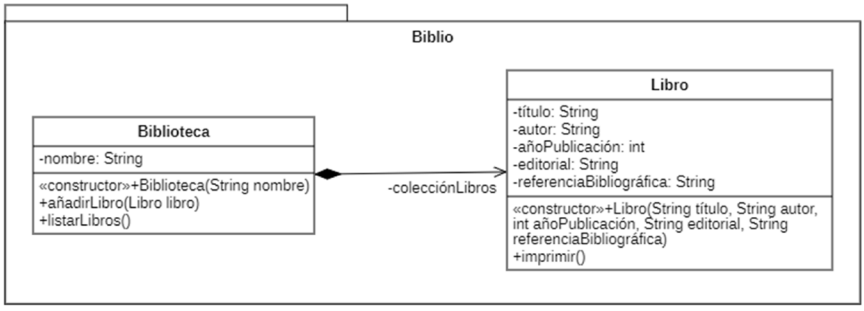


Figura 5.7. Diagrama de clases del ejercicio 5.3.

Explicación del diagrama de clases

Se ha definido un paquete denominado "Biblio", el cual contiene dos clases: Biblioteca y Libro. La clase Biblioteca posee un atributo privado que representa el nombre de la biblioteca (de tipo *String*) y posee un constructor y métodos para añadir un libro a la biblioteca y otro para listar los libros que tiene la biblioteca.

La clase Libro contiene los siguientes atributos: título, autor, año de publicación, editorial y referencia bibliográfica. La clase tiene un constructor y un método para imprimir los datos del libro.

La relación entre la clase Biblioteca y Libro es una relación de composición, expresada en UML como una línea continua que incluye un rombo negro en el extremo de la clase que representa el todo; el extremo contrario representa la clase que es una parte del todo. En este ejercicio, una biblioteca está conformada por una colección de libros. Por ello, el nombre del rol de la asociación se denomina colecciónLibros. La relación de composición es una relación semántica fuerte, si se elimina una biblioteca también se eliminan los libros que contiene. La relación de composición también expresa que no pueden existir libros independientemente de la biblioteca que los contiene.

Diagrama de objetos

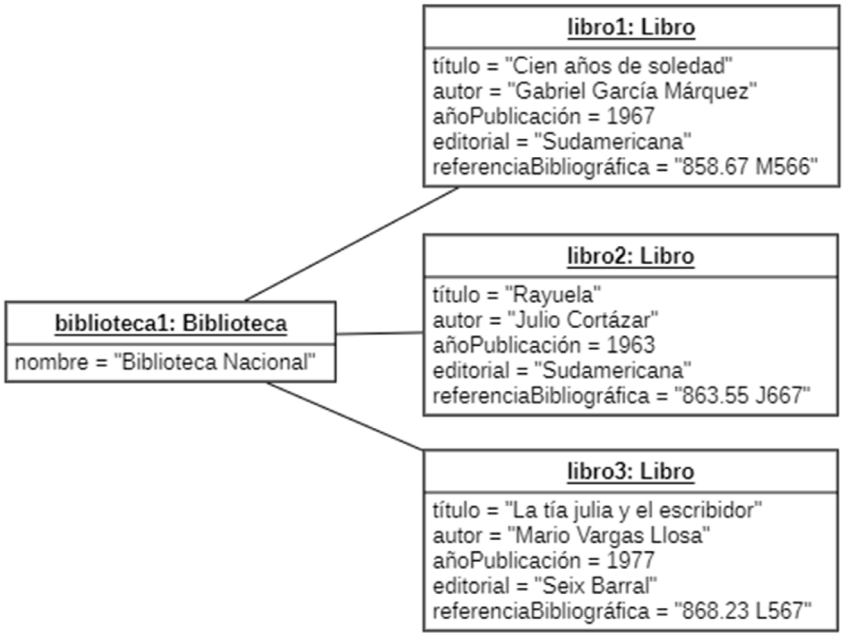


Figura 5.8. Diagrama de objetos del ejercicio 5.3.

Ejecución del programa

```
Título del libro = Cien años de soledad
Autor = Gabriel García Márquez
Año de publicación = 1967
Editorial = Sudamericana
Referencia bibliográfica = 858.67/M566

Título del libro = Rayuela
Autor = Julio Cortázar
Año de publicación = 1963
Editorial = Sudamericana
Referencia bibliográfica = 863.55/J667

Título del libro = La tía julia y el escribidor
Autor = Mario Vargas Llosa
Año de publicación = 1977
Editorial = Seix Barral
Referencia bibliográfica = 868.23/L567
```

Figura 5.9. Ejecución del programa del ejercicio 5.3.

Ejercicios propuestos

- Modificar el programa anterior para que soporte que la biblioteca también pueda contener una colección de revistas. Las revistas contienen los siguientes atributos: nombre de la revista, año de publicación, periodicidad, editorial y país. Las clases Libro y Revista deben ser subclases de una clase denominada Publicación, la cual debe contener atributos compartidos entre las dos clases. Agregar métodos a la clase biblioteca para que se puedan agregar y listar revistas.

Ejercicio 5.4. Composición múltiple

En los dos ejercicios anteriores, se presentó la relación de composición entre solo dos clases (Círculo y Punto en el ejercicio 5.2, y Biblioteca y Libro en el ejercicio 5.3). Esta relación de composición puede aplicarse numerosas veces en el modelado de una solución. Por lo tanto, una clase puede estar compuesta a su vez de muchas clases. De tal manera que, si se elimina la clase que representa el todo, sus clases (que constituyen las partes) también se eliminan.

Las relaciones de composición se usan para representar situaciones en que los objetos están compuestos por múltiples objetos, cada uno con diferentes multiplicidades.

Objetivo de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para definir relaciones de composición; la clase contenedora posee múltiples objetos, cada uno con diferente multiplicidad.

Enunciado: clase Carro y sus partes

Un carro está compuesto por un motor, un chasis, cuatro llantas y una carrocería.

- Los motores tienen un atributo para representan el volumen del motor en litros.
- El chasis tiene un atributo, el tipo de chasis, que es un valor enumerado con valores Independiente o Monocasco.
- La carrocería tiene un atributo que representa el tipo de carrocería, enumerado con valores Independiente, Autoportante o Tubular y un color (tipo *String*).
- Las llantas tienen atributos como: marca (tipo *String*); diámetro del rin, altura y anchura (los últimos tres son tipo *int*).

Cada parte de un carro tiene su correspondiente constructor y un método *imprimir* que muestra los valores de sus atributos en pantalla.

Se requiere en una clase de prueba, desarrollar un método *main* que cree un carro con un motor de 2 litros, un chasis monocasco, una carrocería de color rojo y tipo tubular, 4 llantas de marca "Goodyear", diámetro del rin de 25 pulgadas, altura de 20 pulgadas y anchura de 15 pulgadas. Luego, se deben *imprimir* los datos del carro en pantalla.

Solución

Dato enumerado: tipoChasis

```
package Carro;

/**
 * Se define un valor enumerado que especifica el tipo de chasis
 * de un automóvil que puede ser: INDEPENDIENTE o MONOCASCO
 * @version 1.2/2020
 */
public enum tipoChasis {
    INDEPENDIENTE, MONOCASCO
}
```

Dato enumerado: tipoCarrocería

```
package Carro;

/**
 * Se define un valor enumerado que especifica el tipo de carrocería
 * de un automóvil que puede ser: INDEPENDIENTE,
 * AUTOPARLANTE o TUBULAR
 * @version 1.2/2020
 */
public enum tipoCarrocería {
    INDEPENDIENTE, AUTOPORTANTE, TUBULAR
}
```

Clase: Carro

```
package Carro;

/**
 * Esta clase denominada Carro modela un carro que está compuesto de
 * los siguientes atributos: un motor, un chasis, una carrocería y 4 llantas.
 * @version 1.2/2020
 */
public class Carro {
    Motor motor; // Atributo que identifica el motor de un automóvil
    Chasis chasis; // Atributo que identifica el chasis de un automóvil
```

```
    Llanta[] llantas; /* Atributo que identifica las llantas de un automóvil
        como un array */
    Carrocería carrocería; // Atributo que identifica la carrocería de un
        automóvil

    /**
    * Constructor de la clase Carro
    * @param volumen Parámetro que define el volumen del motor de
    * un automóvil
    * @param claseChasis Parámetro que define el tipo de chasis de un
    * automóvil
    * @param color Parámetro que define el color de un automóvil
    * @param claseCarrocería Parámetro que define el tipo de carrocería
    * de un automóvil
    * @param marca Parámetro que define la marca de las llantas de un
    * automóvil
    * @param diametro Parámetro que define el diámetro de las llantas
    * de un automóvil
    * @param altura Parámetro que define la altura de las llantas de un
    * automóvil
    * @param anchura Parámetro que define la anchura de las llantas de
    * un automóvil
    */
    public Carro(int volumen, tipoChasis claseChasis, String color,
        tipoCarrocería claseCarrocería,
                String marca, int diametro, int altura, int anchura) {
        motor = new Motor(volumen); // Crea el motor de un automóvil
        chasis = new Chasis(claseChasis); // Crea el chasis de un automóvil
        llantas = new Llanta[4]; // Crea un array de 4 llantas
        for (int i = 0; i < llantas.length; i++) {
            // Crea cada una de las cuatro llantas de un automóvil
            llantas[i] = new Llanta(marca,diametro,altura,anchura);
        }
        carrocería = new Carrocería(color,claseCarrocería); /* Crea la
            carrocería de un automóvil */
    }

    /**
    * Método que muestra en pantalla los datos de un automóvil y sus
    * diferentes componentes
    */
    void imprimir() {
        System.out.println("Datos del Carro");
```

```
        System.out.println("Motor - Cilindros = " + motor.volumen);
        System.out.println("Chasis - Tipo = " + chasis.tipo);
        System.out.println("Carrocería - Tipo = " + carrocería.tipo + "
            Color = " + carrocería.color);
        System.out.println("Llantas - Cantidad = " + llantas.length);
        for (int i = 0; i < llantas.length; i++)
            llantas[i].imprimir(); // Imprime los datos de cada llanta
    }
}
```

Clase: Motor

```
package Carro;

/**
 * Esta clase denominada Motor modela el motor de un automóvil y
 * cuenta con un único atributo para identificar el volumen del motor.
 * @version 1.2/2020
 */
public class Motor {
    int volumen; // Atributo que identifica el volumen en litros del motor

    /**
     * Constructor de la clase Motor
     * @param volumen Parámetro que define el volumen de un motor
     */
    Motor(int volumen) {
        this.volumen = volumen;
    }
}
```

Clase: Chasis

```
package Carro;

/**
 * Esta clase denominada Chasis modela el chasis de un automóvil y
 * cuenta con un único atributo para identificar el tipo de chasis, el cual
 * es un valor enumerado.
 * @version 1.2/2020
 */
```

```
public class Chasis {
    tipoChasis tipo; // Atributo enumerado que identifica el tipo de chasis

    /**
     * Constructor de la clase Chasis
     * @param tipo Parámetro que define el tipo de chasis de un automóvil
     */
    Chasis(tipoChasis tipo) {
        this.tipo = tipo;
    }
}
```

Clase: Llanta

```
package Carro;

/**
 * Esta clase denominada Llanta modela la llanta de un automóvil y
 * cuenta con los atributos marca de la llanta, diámetro del rin, altura de
 * la llanta y anchura de la llanta.
 * @version 1.2/2020
 */
public class Llanta {
    String marca; // Atributo que identifica la marca de una llanta
    int diametroRin; // Atributo que identifica el diámetro del rin de una llanta
    int altura; // Atributo que identifica la altura de una llanta
    int anchura; // Atributo que identifica la anchura de una llanta

    /**
     * Constructor de la clase Llanta
     * @param marca Parámetro que define la marca de la llanta de un
     * automóvil
     * @param diametroRin Parámetro que define el diámetro del rin de
     * la llanta de un automóvil
     * @param altura Parámetro que define la altura de la llanta de un
     * automóvil
     * @param anchura Parámetro que define la anchura de la llanta de
     * un automóvil
     */
```

```
    Llanta(String marca, int diametroRin, int altura, int anchura) {
        this.marca = marca;
        this.diametroRin = diametroRin;
        this.altura = altura;
        this.anchura = anchura;
    }

    /**
    * Método que muestra en pantalla los datos de una llanta de un
    * autómovil
    */
    void imprimir() {
        System.out.println("Marca = " + marca);
        System.out.println("Diámetro del rin = " + diametroRin);
        System.out.println("Altura = " + altura);
        System.out.println("Anchura = " + anchura);
    }
}
```

Clase: Carrocería

```
package Carro;

/**
 * Esta clase denominada Carrocería modela la carrocería de un
 * automóvil y cuenta con los atributos tipo de carrocería y color de la
 * carrocería.
 * @version 1.2/2020
 */
public class Carrocería {
    tipoCarrocería tipo; /* Atributo que identifica el tipo de carrocería de
        un automóvil */
    String color; /* Atributo que identifica el color de la carrocería de un
        automóvil */

    /**
    * Constructor de la clase Carrocería
    * @param color Parámetro que define el color de la carrocería de un
    * automóvil
    * @param tipo Parámetro que define el tipo de la carrocería de un
    * automóvil
    */
```

```
    Carrocería(String color, tipoCarrocería tipo) {
        this.color = color;
        this.tipo = tipo;
    }
}
```

Clase: Prueba

```
package Carro;

/**
 * Esta clase prueba la creación de un carro con sus diferentes partes
 * constituyentes.
 * @version 1.2/2020
 */
public class Prueba {

    /**
    * Método main que crea un carro con sus partes constituyentes:
    * chasis, motor, carrocería y  llantas
    */
    public static void main(String args[]) {
        Carro carro = new Carro(2,tipoChasis.
            MONOCASCO,"Rojo",tipoCarrocería.TUBULAR,
            "Goodyear",25,20,15); carro.imprimir();
    }
}
```

Diagrama de clases

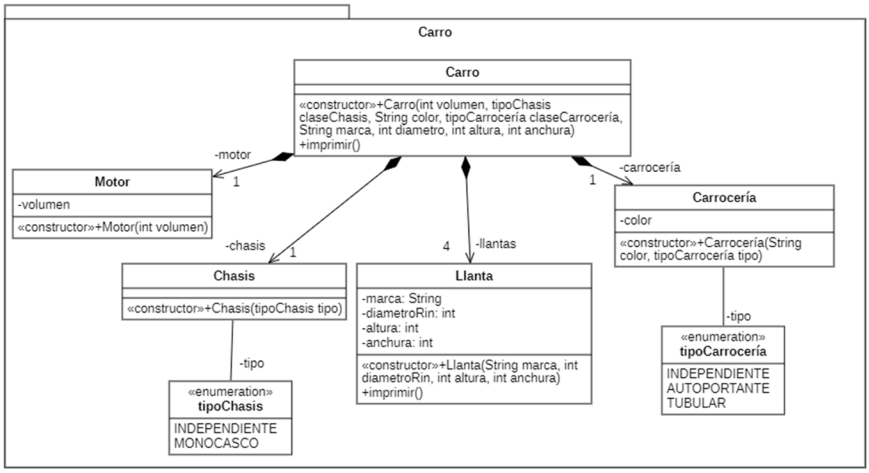


Figura 5.10. Diagrama de clases del ejercicio 5.4.

Explicación del diagrama de clases

Se ha definido un paquete denominado "Carro", que contiene una clase principal llamada Carro. La clase Carro está compuesta de varias clases a través de relaciones de composición múltiple. En este caso, un carro está conformado por los siguientes elementos: un motor, un chasis, una carrocería y cuatro llantas. Las relaciones de composición se denotan como una línea continua donde un extremo presenta un rombo negro adyacente a la clase que representa al todo y en el extremo opuesto, la clase que representa una parte. Por lo tanto, según el diagrama de clases, la clase Carro es el todo y está constituida por sus partes correspondientes. El diagrama especifica la multiplicidad de cada relación donde un carro tiene específicamente un motor, una carrocería, un chasis y cuatro llantas.

Cada clase tiene sus atributos, constructor y métodos, si es el caso. La clase Motor cuenta con un atributo para el volumen del motor y un constructor que inicializa el mismo. La clase Chasis tiene un atributo para identificar el tipo de chasis y un constructor para inicializar dicho atributo. La clase Llanta cuenta con los atributos que identifican la marca, diámetro, altura y anchura de la llanta; posee un constructor para inicializar dichos

atributos y un método para *imprimir* sus datos en pantalla. La clase Carrocería tiene un atributo para identificar el color de la carrocería y otro para identificar el tipo de carrocería, además, de un constructor para inicializar los atributos mencionados.

Se han definido algunos atributos que son datos enumerados representados con la notación de clase UML, pero con el estereotipo <<*enumeration*>>, sus valores constantes se colocan en el comportamiento de los atributos. La clase Chasis tiene un atributo enumerado tipoChasis con los posibles valores de independiente o monocasco. A su vez, la clase Carrocería tiene un atributo enumerado tipoCarrocería con los posibles valores de independiente, autoparlante o tubular.

Diagrama de objetos

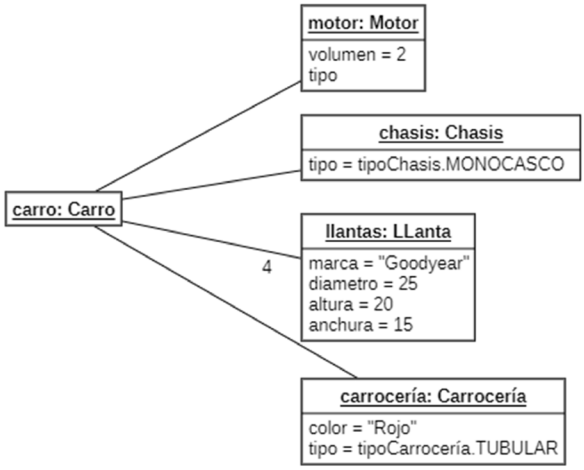


Figura 5.11. Diagrama de objetos del ejercicio 5.4.

Ejecución del programa

```
Datos del Carro
Motor - Cilindros = 2
Chasis - Tipo = MONOCASCO
Carrocería - Tipo = TUBULAR Color = Rojo
Llantas - Cantidad = 4
Marca = Goodyear
Diámetro del rin = 25
Altura = 20
Anchura = 15
Marca = Goodyear
Diámetro del rin = 25
Altura = 20
Anchura = 15
Marca = Goodyear
Diámetro del rin = 25
Altura = 20
Anchura = 15
Marca = Goodyear
Diámetro del rin = 25
Altura = 20
Anchura = 15
```

Figura 5.12. Ejecución del programa del ejercicio 5.4.

Ejercicios propuestos

- Agregar otros componentes que tiene un carro como:
  - Los tres asientos: el asiento del conductor; el asiento del acompañante y el asiento trasero. Los asientos tienen atributos como el tipo de material y si tiene o no funda.
  - Agregar otros elementos de un carro. Por ejemplo, los que componen el *full* equipo como *airbags*, frenos ABS, vidrios eléctricos, espejos eléctricos, *sunroof*, etc. Considerar los atributos que se estimen convenientes.

Ejercicio 5.5. Relación de agregación

La relación de agregación es un tipo de asociación entre clases, pero débil, es decir, si la clase contenedora que representa al todo se elimina, sus partes siguen existiendo independientemente (Booch, Rumbaugh y Jacobson, 2017).

Al igual que la composición, la clase contenedora debe poseer un atributo para cada una de sus partes. Pero la diferencia a nivel de código se encuentra en el constructor de la clase contenedora, donde las partes no se crean en el constructor de la clase, sino que sus partes han sido previamente creadas y se pasan como parámetros en el constructor. Estos parámetros se asignan a los atributos correspondientes en el cuerpo del constructor (Hunt, 2013). El formato de la relación de agregación es:

```
class Todo {
        Parte parte1, parte2;
        Todo(Parte1 p1, Parte2 p2) {
                parte1 = p1;
                parte2 = p2;
        }
}
```

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Diferenciar la relación de agregación de la composición.
- Definir relaciones de agregación entre clases, identificando las clases que representan el todo (contenedoras) y sus partes constituyentes.
- Agregar las partes de la clase contenedora dentro del constructor de la clase que representa el todo.

Enunciado: clase Orden de compra y jerarquía de agregación

Una orden de compra tiene un identificador numérico y está compuesta de un conjunto de líneas de pedido.

Una orden de compra cuenta con los siguientes métodos:

- Un constructor que inicializa sus atributos.
- Métodos *get* y *set* para el atributo identificador.
- Calcular el total de una orden sumando los subtotales de cada línea de pedido.
- *toString*: este método devuelve un *String* que concatena los siguientes atributos: identificador, las líneas de pedido que conforman la orden y el total de la orden.
- Añadir ítem: permite agregar una línea de pedido a la orden.

Una línea de pedido consta de un identificador, la cantidad de producto solicitada y el producto solicitado en sí. Una línea de pedido cuenta con los siguientes métodos:

- Un constructor que inicializa sus atributos.
- Métodos *get* y *set* para sus atributos.
- Calcular el subtotal de la línea de pedido, que es la multiplicación del precio del producto por su cantidad.
- *toString*: este método devuelve un *String* que concatena los atributos: identificador, cantidad y producto solicitado.

Los productos tienen atributos como: identificador, nombre, descripción y precio. Un producto cuenta con los siguientes métodos:

- Un constructor que inicializa sus atributos.
- Métodos *get* y *set* para sus atributos.
- *toString*: este método devuelve un *String* que concatena los atributos: identificador, nombre y precio.

Se debe generar una clase de prueba con un método *main* para crear una orden de compra con los siguientes productos:

- Producto 1:
  - Identificador: 1
  - Nombre: Carpeta
  - Descripción: Carpeta anillada metálica
  - Precio: $1000
- Producto 2:
  - Identificador: 2
  - Nombre: Tinta
  - Descripción: Tinta china de color negro
  - Precio: $3000
- Producto 3:
  - Identificador: 3
  - Nombre: Cinta pegante
  - Descripción: Cinta adhesiva de color transparente
  - Precio: $800

- Producto 4:
  - Identificador: 4
  - Nombre: Cartulina
  - Descripción: Pliego de cartulina 60 x 40 cm
  - Precio: $600

Se debe crear una orden de compra que tenga 5 carpetas, 3 tintas, 2 cintas pegantes y 4 cartulinas.

Solución

Clase: Orden

```
package Agregación;
import java.util.*;

/**
 * Esta clase denominada Orden modela una orden de pedido que está
 * conformada por el identificador de la orden y un vector de líneas de
 * pedido.
 * @version 1.2/2020
 */
class Orden {
    // Atributo que representa el identificador de la orden
    private int identificador;
    /* Atributo que identifica los ítems del pedido por medio de un vector
        de Líneas de Pedido */
    private Vector itemsPedido;

    /**
    * Constructor de la clase Orden
    * @param identificador Parámetro que define el identificador de la
    * orden de pedido
    */
    public Orden(int identificador) {
        this.identificador = identificador;
        itemsPedido = new Vector(); // Crea el vector de líneas de pedido
    }

/**
    * Método que obtiene el identificador de una orden de pedido
```

```
* @return El identificador de una orden de pedido
*/
public int getIdentificador() {
    return identificador;
}

/**
* Método que establece el identificador de una orden de pedido
* @param identificador Parámetro que define el identificador de una
* orden de pedido
*/
public void setIdentificador(int identificador) {
    this.identificador = identificador;
}

/**
* Método que calcula el total de una orden de pedido
* @return El total de una orden de pedido
*/
public int calcularTotalOrden() {
    int totalOrden = 0;
    for(int i = 0; i < itemsPedido.size(); i++) { /* Recorre el vector de
        líneas de pedido */
        // Obtiene cada objeto del vector aplicando casting
        LíneaPedido línea = (LíneaPedido) itemsPedido.elementAt(i);
        // Va totalizando la orden mediante la suma de los subtotales
        totalOrden = totalOrden + línea.calcularSubtotalLíneaPedido();
    }
    return totalOrden;
}

/**
* Método que convierte a String los datos de una orden de pedido
* @return Un String con los datos de una orden de pedido
*/
public String toString() {
    return "Orden [id = " + identificador +"]\n" + "Items del pedido\n"
        + itemsPedido +
            "\nTotal orden = $" + calcularTotalOrden();
}

/**
* Método que permite agregar una línea de pedido a la orden
```

```
     * @param identificador Parámetro que define el identificador de la
     * línea de pedido
     * @param cantidad Parámetro que define la cantidad de producto
     * solicitado en la línea de pedido
     * @param producto Parámetro que define el producto solicitado en
     * la línea de pedido
     */
    public void añadirItem(int identificador, int cantidad, Producto
        producto) {
        // Crea un línea de pedido
        LíneaPedido línea= new LíneaPedido(identificador, cantidad,
            producto);
        itemsPedido.add(línea); /* Añade la línea de pedido creada al
            vector de líneas de pedido */
    }
}
```

Clase: LíneaPedido

```
package Agregación;

/**
 * Esta clase denominada LíneaPedido modela un línea específica de
 * productos que conforman un pedido. Una línea de pedido tiene como
 * atributos un identificador de la línea de pedido, la cantidad de
 * producto a solicitar y el producto concreto solicitado.
 * @version 1.2/2020
 */
class LíneaPedido {
    private int identificador; /* Atributo que representa el identificador
        de una línea de pedido */
    /* Atributo que define la cantidad de producto solicitado en una línea
        de pedido */
    private int cantidad;
    private Producto producto; /* Atributo que define el producto
        solicitado en una línea de pedido */

    /**
     * Constructor de la clase LíneaPedido
```

```
* @param identificador Parámetro que define el identificador de la
* línea de pedido
* @param cantidad Parámetro que define la cantidad de un
* producto solicitado en la línea de pedido
* @param producto Parámetro que define el Producto solicitado en
* la línea de pedido
*/
public LíneaPedido(int identificador, int cantidad, Producto
    producto) {
    this.identificador = identificador;
    this.cantidad = cantidad;
    this.producto = producto;
}

/**
* Método que obtiene el identificador de un producto
* @return El identificador de un producto
*/
public int getIdentificador() {
    return identificador;
}

/**
* Método que establece el identificador de un producto
* @param identificador Parámetro que define el identificador de un
* producto
*/
public void setIdentificador(int identificador) {
    this.identificador = identificador;
}

/**
* Método que obtiene la cantidad de un producto solicitado en una
* línea de pedido
* @return La cantidad de producto solicitado en una línea de pedido
*/
public int getCantidad() {
    return cantidad;
}

/**
```

```
    * Método que establece la cantidad de un producto solicitado en
    * una línea de pedido
    * @param cantidad Parámetro que define la cantidad de un
    * producto solicitado en una línea de pedido
    */
    public void setCantidad(int cantidad) {
        this.cantidad = cantidad;
    }

    /**
    * Método que obtiene el producto solicitado en una línea de pedido
    * @return El producto solicitado en una línea de pedido
    */
    public Producto getProducto() {
        return producto;
    }

    /**
    * Método que establece el producto solicitado en una línea de pedido
    * @param producto Parámetro que define el producto solicitado en
    * una línea de pedido
    */
    public void setProducto(Producto producto) {
        this.producto = producto;
    }

    /**
    * Método que calcula el subtotal de la línea de pedido
    * @return El subtotal de la línea de pedido
    */
    public int calcularSubtotalLíneaPedido() {
        /* El subtotal se calcula como el precio del producto multiplicado
             por su cantidad */
        return cantidad * producto.getPrecio();
    }

    /**
    * Método que convierte a String los datos de una línea de pedido
    * @return Un String con los datos de una línea de pedido
    */
    public String toString() {
```

```
        return "Línea de Pedido [id = " + identificador + ", Cantidad = " +
            cantidad + ", Producto = " + producto + "]\n";
    }
}
```

Clase: Producto

```
package Agregación;

/**
 * Esta clase denominada Producto modela un producto comercial con
 * los atributos: identificador de producto, nombre del producto, una
 * descripción del producto y un precio del producto.
 * @version 1.2/2020
 */
class Producto {
    private int identificador; /* Atributo que representa el identificador
        de un producto */
    private String nombre; // Atributo que identifica el nombre de un producto
    private String descripción; /* Atributo que identifica la descripción
        de un producto */
    private int precio; // Atributo que identifica el precio de un producto

    /**
    * Constructor de la clase Producto
    * @param identificador Parámetro que define el identificador de un
    * producto
    * @param nombre Parámetro que define el nombre de un producto
    * @param descripción Parámetro que define la descripción de un
    * producto
    * @param precio Parámetro que define el precio de un producto
    */
    public Producto(int identificador, String nombre, String descripción,
        int precio) {
        this.identificador = identificador;
        this.nombre = nombre;
        this.descripción = descripción;
        this.precio = precio;
```

```
}

/**
* Método que obtiene el identificador de un producto
* @return El identificador de un producto
*/
public int getIdentificador() {
    return identificador;
}

/**
* Método que establece el identificador de un producto
* @param identificador Parámetro que define el identificador de un
* producto
*/
public void setIdentificador(int identificador) {
    this.identificador = identificador;
}

/**
* Método que obtiene el nombre de un producto
* @return El nombre de un producto
*/
public String getNombre() {
    return nombre;
}

/**
* Método que establece el nombre de un producto
* @param nombre Parámetro que define el nombre de un producto
*/
public void setNombre(String nombre) {
    this.nombre = nombre;
}

/**
* Método que obtiene la descripción de un producto
* @return La descripción de un producto
*/
public String getDescripción() {
    return descripción;
}
```

```
    /**
    * Método que establece la descripción de un producto
    * @param descripción Parámetro que define la descripción de un
    * producto
    */
    public void setDescripción(String descripción) {
        this.descripción = descripción;
    }

    /**
    * Método que obtiene el precio de un producto
    * @return El precio de un producto
    */
    public int getPrecio() {
        return precio;
    }

    /**
    * Método que establece el precio de un producto
    * @param precio Parámetro que define el precio de un producto
    */
    public void setPrecio(int precio) {
        this.precio = precio;
    }

    /**
    * Método que convierte a String los datos de un producto
    * @return Un String con los datos concatenados de un producto
    */
    public String toString() {
        return "id = " + identificador + ", Nombre = " + nombre + ",
            Descripción = " + descripción + ", Precio = $ " + precio;
    }
}
```

Clase: Prueba

```
package Agregación;

/**
* Esta clase prueba la creación de una orden de pedido conformada por
* diferentes línea de pedido con sus productos.
```

```
 * @version 1.2/2020
 */
public class Prueba {

    /**
    * Método main que crea una orden conformada por cuatro
    * productos con sus identificadores y cantidad. Luego, se añaden los
    * cuatro productos como línea de pedido a la orden y se calcula el
    * total de la orden
    */
    public static void main(String[] args) {
        Producto producto1 = new Producto(1, "Carpeta", "Carpeta
            anillada metálica",1000);
        Producto producto2 = new Producto(2, "Tinta", "Tinta china de
            color negro.", 3000);
        Producto producto3 = new Producto(3, "Cinta pegante",
                "Cinta adhesiva de color transparente.",800);
        Producto producto4 = new Producto(4, "Cartulina", "Pliego de
            cartulina 60x40 cms.",600);
        Orden orden = new Orden(1);
        orden.añadirItem(1, 5, producto1);
        orden.añadirItem(2, 3, producto2);
        orden.añadirItem(3, 2, producto1);
        orden.añadirItem(4, 4, producto4);
        orden.calcularTotalOrden();
        System.out.println(orden);
    }
}
```

Diagrama de clases

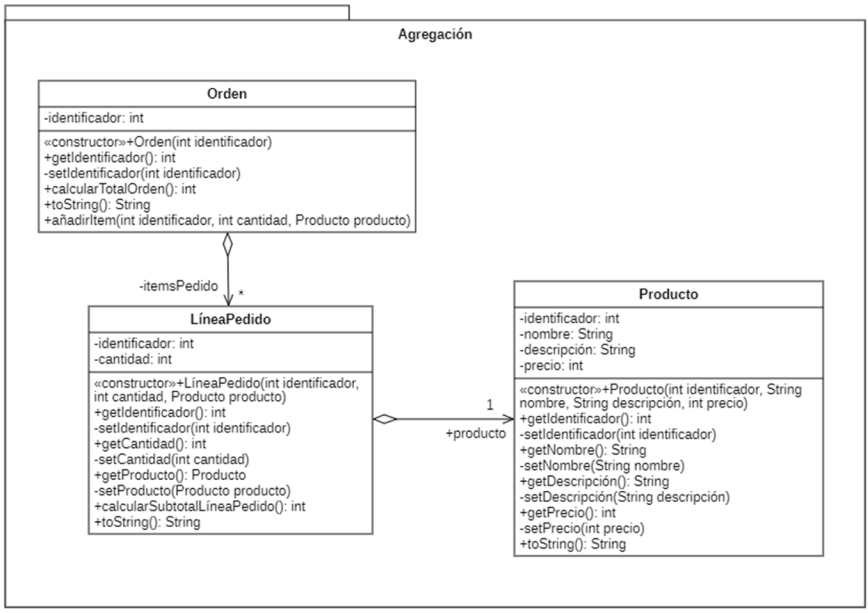


Figura 5.13. Diagrama de clases del ejercicio 5.5.

Explicación del diagrama de clases

Se ha definido un paquete denominado "Agregación" que incluye tres clases para gestionar órdenes de compra. La clase principal es Orden que cuenta con un identificador como atributo, un constructor y varios métodos. Una Orden está constituida por líneas de pedidos, esta relación se manifiesta en el diagrama por medio de una relación de agregación entre la clase Orden y LíneaPedido. Una relación de agregación se denota gráficamente como una línea continua con un rombo blanco adyacente a la clase que representa el todo y en el otro extremo, la parte agregada. La relación de agregación tiene una multiplicidad de muchos (*), esto indica que una orden puede estar constituida por varias líneas de pedido. La relación de agregación también indica una relación semántica débil entre la clase que representa el todo y la parte, es decir, que la parte puede existir independientemente del todo.

A su vez, una línea de pedido, que está conformada por un solo producto, nuevamente es representada mediante una relación de agregación.

La clase contenedora Orden posee un método que permite agregar líneas de pedido a la orden (añadirItem). De tal manera que el vector se va llenando de objetos ítemsPedido. La clase Orden tiene un método para calcular el total de la orden (calcularTotalOrden) y la clase LíneaPedido tiene un método para calcular el subtotal de la línea de pedido (calcularSubtotalLíneaPedido).

La clase Orden tiene como atributo su identificador. La clase LíneaPedido tiene como atributos un identificador y la cantidad de producto a ordenar. Los atributos de la clase Producto son el identificador de cada producto, nombre del producto, descripción y precio del producto. Todas las clases cuentan con métodos *get* y *set* para sus atributos y un método *toString* para convertir los valores de sus atributos a *String*.

Diagrama de objetos

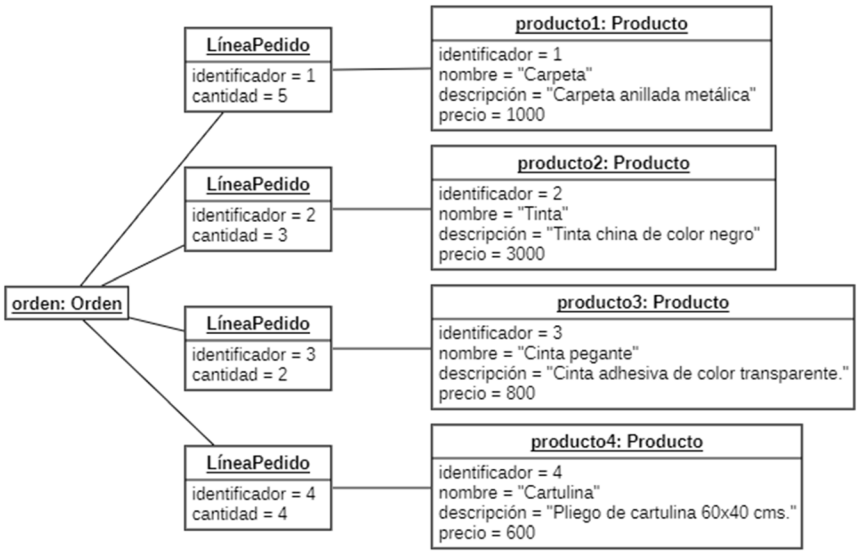


Figura 5.14. Diagrama de objetos del ejercicio 5.5.

Ejecución del programa

```
Orden [id = 1]
Items del pedido
[Línea de Pedido [id = 1, Cantidad = 5, Producto = id = 1, Nombre = Carpeta, Descripción = Carpeta anillada metálica, Precio = $ 1000]
, Línea de Pedido [id = 2, Cantidad = 3, Producto = id = 2, Nombre = Tinta, Descripción = Tinta china de color negro., Precio = $ 3000]
, Línea de Pedido [id = 3, Cantidad = 2, Producto = id = 1, Nombre = Carpeta, Descripción = Carpeta anillada metálica, Precio = $ 1000]
, Línea de Pedido [id = 4, Cantidad = 4, Producto = id = 4, Nombre = Cartulina, Descripción = Pliego de cartulina 60x40 cms., Precio = $ 600]
]
Total orden = $18400
```

Figura 5.15. Ejecución del programa del ejercicio 5.5.

Ejercicios propuestos

- Modificar el programa anterior para que soporte que las órdenes de compra están relacionadas con un único cliente. Los clientes a su vez pueden tener muchas órdenes de compra. Los cuales pueden ser empresas o clientes particulares; las primeras tienen un nombre o razón social, NIT, teléfono y dirección. Los últimos tienen nombre, apellidos, número de documento, teléfono y dirección.

Ejercicio 5.6. Diferencias entre agregación y composición

La relación de agregación entre clases se caracteriza porque las clases que representan las partes pueden ser compartidas entre varias clases contenedoras, de la misma relación de agregación o de varias asociaciones de agregación distintas. La relación de agregación, generalmente, es más frecuente que la composición (Seidl *et al.*, 2015).

En la tabla 5.2 se presentan las diferencias entre la relación de agregación y composición.

Tabla 5.2. Diferencias entre las relaciones de agregación y composición

Características	Agregación	Composición
Diferentes relaciones comparten las partes	Sí	No
Cuando se destruye el todo también se destruyen sus partes	No	Sí
A nivel de código	Las partes se asignan en el constructor del todo	Las partes se instancian en el constructor del todo
Representación en UML	Rombo transparente	Rombo negro

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Diferenciar la relación de agregación de la composición.
- Establecer relaciones de agregación entre clases.
- Expresar relaciones de agregación entre clases por medio de código de Java.

Enunciado: clase Equipo de fútbol y su jerarquía de agregación

Se requiere desarrollar un programa que modele la conformación de un equipo de fútbol teniendo en cuenta las siguientes especificaciones. Un equipo de fútbol tiene atributos como: nombre, país, un técnico, un portero, cuatro defensas, cuatro mediocampos y dos delanteros. Se requieren dos constructores para el equipo: el primero recibe como parámetros el nombre del equipo y del país. El segundo invoca al primero e inicializa los jugadores que conforman el equipo.

Los técnicos y jugadores son personas. Las personas tienen un nombre, apellidos y edad. Además, los jugadores tienen un atributo booleano para representar si son titulares o no en el equipo. Los técnicos tienen como atributos los años de experiencia (tipo *int*) y si son nacionales o extranjeros (tipo *boolean*).

Los jugadores pueden ser porteros, defensas, mediocampos o delanteros. Los porteros tienen como atributo propio la cantidad de goles recibidos y los jugadores mediocampos, el número de asistencias realizadas (tipo *int*). Finalmente, los delanteros tienen como atributo propio la cantidad de goles anotados.

Además, se requiere un método *imprimir* que muestre en pantalla los datos del equipo de fútbol, su técnico y sus jugadores. Se debe realizar una clase de prueba con un método *main* que cree un equipo de fútbol de la Selección Colombia con los siguientes datos:

- Técnico: Carlos Queiroz, 66 años, 30 años de experiencia y es extranjero.
- Portero: David Ospina, 30 años y 10 goles recibidos.

Los defensas son:

- Yerry Mina, 24 años.

- Davison Sánchez, 23 años.
- William Tesillo, 29 años.
- Stefan Medina, 29 años.

Los jugadores del mediocampo son:

- Mateus Uribe, 28 años, 12 asistencias.
- Wilmar Barrios, 25 años, 12 asistencias.
- Juan Guillermo Cuadrado, 31 años, 10 asistencias.
- James Rodríguez, 28 años, 32 asistencias.

Los delanteros son:

- Radamel Falcao García, 33 años, 15 goles.
- Duván Zapata, 28 años, 12 goles.

Todos los jugadores son titulares.

Solución

Clase: *Persona*

```
package Fútbol;

/**
 * Esta clase denominada Persona es una clase abstracta que modela una
 * persona genérica.
 * Una persona tiene un nombre, apellidos y una edad.
 * @version 1.2/2020
 */
abstract public class Persona {
    String nombre; // Atributo que define el nombre de una persona
    String apellidos; // Atributo que define los apellidos de una persona
    int edad; // Atributo que define la edad de una persona

    /**
    * Constructor de la clase Persona
    * @param nombre Parámetro que define el nombre de una persona
    * @param apellidos Parámetro que define los apellidos de una persona
    * @param edad Parámetro que define la edad de una persona
    */
```

```
    public Persona(String nombre, String apellidos, int edad) {
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.edad = edad;
    }
}
```

Clase: *Jugador*

```
package Fútbol;

/**
 * Esta clase denominada Jugador es una clase abstracta que modela un
 * jugador de fútbol.
 * @version 1.2/2020
 */
abstract public class Jugador extends Persona {
    boolean esTitular; /* Atributo que define si un jugador es titular o no
        en un equipo */

    /**
     * Constructor de la clase Jugador
     * @param nombre Parámetro que define el nombre de un jugador
     * @param apellidos Parámetro que define los apellidos de un jugador
     * @param edad Parámetro que define la edad de un jugador
     * @param esTitular Parámetro que define si un jugador es titular o no
     */
    Jugador(String nombre, String apellidos, int edad, boolean esTitular)
        {
        super(nombre, apellidos, edad);
        esTitular = esTitular;
    }
}
```

Clase: Técnico

```
package Fútbol;

/**
 * Esta clase denominada Técnico es una clase que modela un técnico de
 * fútbol.
 * @version 1.2/2020
 */
```

```
public class Técnico extends Persona {
    int añosExperiencia; /* Atributo que define los años de experiencia
        que tiene un técnico */
    boolean esNacional; /* Atributo que define si un técnico es nacional
        o extranjero */

    /**
    * Constructor de la clase Técnico
    * @param nombre Parámetro que define el nombre de un técnico
    * @param apellidos Parámetro que define los apellidos de un técnico
    * @param edad Parámetro que define la edad de un técnico
    * @param añosExperiencia Parámetro que define los años de
    * experiencia dirigiendo equipos de un técnico
    * @param esNacional Parámetro que define si un técnico es nacional
    * o extranjeto
    */
    public Técnico(String nombre, String apellidos, int edad, int
        añosExperiencia, boolean esNacional) {
        super(nombre, apellidos, edad);
        this.añosExperiencia = añosExperiencia;
        this.esNacional = esNacional;
    }

    /**
    * Método que convierte a String los datos de un técnico
    * @return Un String que concatena los datos de un técnico
    */
    public String toString() {
        return "Nombre = " + nombre + ", Apellidos = " + apellidos + ",
            Edad = " + edad + " Años de experiencia = " + añosExperiencia;
    }
}
```

Clase: Portero

```
package Fútbol;

/**
 * Esta clase denominada Portero es una subclase de Jugador que
 * modela un portero de un equipo de fútbol.
 * @version 1.2/2020
 */
```

```
public class Portero extends Jugador {
    int golesRecibidos; /* Atributo que define la cantidad de goles
        recibidos por un portero de fútbol */

    /**
    * Constructor de la clase Portero
    * @param nombre Parámetro que define el nombre de un portero
    * @param apellidos Parámetro que define los apellidos de un portero
    * @param edad Parámetro que define la edad de un portero
    * @param esTitular Parámetro que define si un portero es titular o no
    * @param golesRecibidos Parámetro que define la cantidad de goles
    * recibidos por un portero
    */
    public Portero(String nombre, String apellidos, int edad, boolean
        esTitular, int golesRecibidos) {
        super(nombre, apellidos, edad, esTitular); /* Invoca al constructor
            de la clase padre */
        this.golesRecibidos = golesRecibidos;
    }

    /**
    * Método que convierte a String los datos de un portero
    * @return Un String que concatena los datos de un portero
    */
    public String toString() {
        return "Nombre = " + nombre + ", Apellidos = " + apellidos + ",
            Edad = " + edad + ", Goles recibidos = " + golesRecibidos;
    }
}
```

Clase: Defensa

```
package Fútbol;

/**
 * Esta clase denominada Defensa es una subclase de Jugador que
 * modela un jugador que realiza actividades de defensa en un equipo
 * de fútbol.
 * @version 1.2/2020
 */
```

```
public class Defensa extends Jugador {

    /**
    * Constructor de la clase Defensa
    * @param nombre Parámetro que define el nombre de un defensa
    * @param apellidos Parámetro que define los apellidos de un defensa
    * @param edad Parámetro que define la edad de un defensa
    * @param esTitular Parámetro que define si un defensa es titular o no
    */
    public Defensa(String nombre, String apellidos, int edad , boolean
        esTitular) {
        super(nombre, apellidos, edad, esTitular); /* Invoca al constructor
            de la clase padre */
    }

    /**
    * Método que convierte a String los datos de un defensa
    * @return Un String que concatena los datos de un defensa
    */
    public String toString() {
        return "Nombre = " + nombre + ", Apellidos = " + apellidos + ",
            Edad = " + edad;
    }
}
```

Clase: Mediocampo

```
package Fútbol;

/**
 * Esta clase denominada Mediocampo es una subclase de Jugador que
 * modela un jugador de fútbol que realiza actividades de mediocampo
 * en un equipo de fútbol. Tiene un atributo que representa la cantidad
 * de asistencias que ha realizado.
 * @version 1.2/2020
 */
public class Mediocampo extends Jugador {
    /* Atributo que identifica la cantidad de asistencias que ha realizado
        un mediocampo */
    int númeroAsistencias;
```

```
    /**
     * Constructor de la clase Mediocampo
     * @param nombre Parámetro que define el nombre de un mediocampo
     * @param apellidos Parámetro que define los apellidos de un
     * mediocampo
     * @param edad Parámetro que define la edad de un mediocampo
     * @param esTitular Parámetro que define si un mediocampo es
     * titular o no
     * @param númeroAsistencias Parámetro que define la cantidad de
     * asistencias de un mediocampo
     */
    public Mediocampo(String nombre, String apellidos, int edad,
        boolean esTitular, int númeroAsistencias) {
        super(nombre, apellidos, edad, esTitular); /* Invoca al constructor
            de la clase padre */
        this.númeroAsistencias = númeroAsistencias;
    }

    /**
     * Método que convierte a String los datos de un mediocampo
     * @return Un String que concatena los datos de un mediocampo
     */
    public String toString() {
        return "Nombre = " + nombre + ", Apellidos = " + apellidos + ",
            Edad = " + edad + ", Asistencias = " + númeroAsistencias;
    }
}
```

Clase: Delantero

```
package Fútbol;

/**
 * Esta clase denominada Delantero es una subclase de Jugador que
 * modela un jugador de fútbol que realiza actividades ofensivas en un
 * equipo de fútbol. Tiene un atributo que identifica la cantidad de goles
 * que ha anotado.
 * @version 1.2/2020
 */
public class Delantero extends Jugador {
    int golesAnotados; /* Atributo que identifica la cantidad de goles
        anotados por un delantero */
```

```
    /**
     * Constructor de la clase Delantero
     * @param nombre Parámetro que define el nombre de un delantero
     * @param apellidos Parámetro que define los apellidos de un delantero
     * @param edad Parámetro que define la edad de un delantero
     * @param esTitular Parámetro que define si un delantero es titular o no
     * @param golesAnotados Parámetro que define el número de goles
     * anotados por un delantero
     */
    public Delantero(String nombre, String apellidos,int edad,boolean
        esTitular, int golesAnotados) {
        super(nombre, apellidos, edad, esTitular); /* Invoca al constructor
            de la clase padre */
        this.golesAnotados = golesAnotados;
    }

    /**
     * Método que convierte a String los datos de un delantero
     * @return Un String que concatena los datos de un delantero
     */
    public String toString() {
        return "Nombre = " + nombre + ", Apellidos = " + apellidos + ",
            Edad = " + edad + ", Goles anotados = " + golesAnotados;
    }
}
```

Clase: EquipoFútbol

```
package Fútbol;

/**
 * Esta clase denominada EquipoFútbol es una clase que modela un
 * equipo de fútbol. Un equipo de fútbol tiene un nombre, país, un
 * técnico, un portero, cuatro defensas y dos delanteros.
 * @version 1.2/2020
 */
public class EquipoFútbol {
    String nombre; /* Atributo que identifica el nombre de un equipo de
        fútbol */
    String país; // Atributo que identifica el país de un equipo de fútbol
```

```
    Técnico técnico; /* Atributo que identifica el técnico de un equipo
        de fútbol */
    Portero portero; /* Atributo que identifica el portero de un equipo
        de fútbol
    Defensa[] defensas; /* Atributo que identifica los defensas de un
        equipo de fútbol */
    /* Atributo que identifica los mediocampos de un equipo de fútbol */
    Mediocampo[] mediocampos;
    Delantero[] delanteros; /* Atributo que identifica los delanteros de un
        equipo de fútbol */

    /**
    * Constructor de la clase EquipoFútbol
    * @param nombre Parámetro que define el nombre de un equipo de
    * fútbol
    * @param país Parámetro que define el país del equipo de fútbol
    */
    public EquipoFútbol(String nombre, String país) {
        this.nombre = nombre;
        this.país = país;
    }

    /**
    * Constructor sobrecargado de la clase EquipoFútbol
    * @param nombre Parámetro que define el nombre de un equipo de
    * fútbol
    * @param país Parámetro que define el país del equipo de fútbol
    * @param técnico Parámetro que define el técnico de un equipo de
    * fútbol
    * @param portero Parámetro que define el portero de un equipo de
    * fútbol
    * @param defensas Parámetro que define los defensas de un equipo
    * de fútbol
    * @param mediocampos Parámetro que define los mediocampos de
    * un equipo de fútbol
    * @param delanteros Parámetro que define los delanteros de un
    * equipo de fútbol
    */
    public EquipoFútbol(String nombre, String ciudad, Técnico técnico,
        Portero portero, Defensa[] defensas, Mediocampo[] mediocampos,
            Delantero[] delanteros) {
        this(nombre, ciudad); // Invoca al constructor inicial
```

```
        this.técnico = técnico;
        this.portero = portero;
        this.defensas = defensas;
        this.mediocampos = mediocampos;
        this.delanteros = delanteros;
    }

    /**
    * Método que muestra en pantalla los datos del equipo junto con la
    * información de su técnico, portero, defensas y delanteros
    */
    void imprimir() {
        System.out.println("Nombre del equipo = " + nombre);
        System.out.println("País = " + país);
        System.out.println("Portero [" + portero + "]");
        System.out.println("Defensas");
        for (int i = 0; i < defensas.length; i++) { /* Recorre el array de
            defensas */
            System.out.println(defensas[i]);
        }
        System.out.println("Medicampo");
        for (int j = 0; j < mediocampos.length; j++) { /* Recorre el array
            de mediocampos */
            System.out.println(mediocampos[j]);
        }
        System.out.println("Delanteros");
        for (int k = 0; k < delanteros.length; k++) { /* Recorre el array de
            delanteros */
            System.out.println(delanteros[k]);
        }
    }
}
```

Clase: Prueba

```
package Fútbol;

/**
 * Esta clase prueba la creación de un equipo de fútbol conformado por
 * sus diferentes tipos de jugadores: portero, defensa, mediocampo y
 * delanteros, además de un técnico.
 * @version 1.2/2020
 */
```

```
public class Prueba {
    /**
    * Método main que crea en primer lugar los integrantes específicos
    * de un equipo de fútbol. Luego, compone el equipo con sus
    * integrantes y muestra sus datos en pantalla
    */
    public static void main(String[] args) {
        Técnico técnico = new Técnico("Carlos","Queiroz", 66, 30, false);
            // Crea un técnico
        Portero portero = new Portero("David", "Ospina", 30, true, 10);
            // Crea un portero
        Defensa[] defensas = new Defensa[4]; /* Crea un array de 4
            defensas */
        Mediocampo[] mediocampos = new Mediocampo[4]; /* Crea un
            array de 4 mediocampos */
        Delantero[] delanteros = new Delantero[2]; /* Crea un array de 2
            delanteros */

        // Crea los jugadores específicos de acuerdo a su tipo
        defensas[0] = new Defensa("Yerry", "Mina", 24, true);
        defensas[1] = new Defensa("Davison", "Sánchez", 23, true);
        defensas[2] = new Defensa("William", "Tesillo", 29, true);
        defensas[3] = new Defensa("Stefan", "Medina", 29, true);
        mediocampos[0] = new Mediocampo("Mateus", "Uribe", 28, true,
            12);
        mediocampos[1] = new Mediocampo("Wilmar", "Barrios", 25,
            true, 12);
        mediocampos[2] = new Mediocampo("Juan Guillermo",
            "Cuadrado", 31, true, 10);
        mediocampos[3] = new Mediocampo("James", "Rodríguez", 28,
            true, 32);
        delanteros[0] = new Delantero("Radamel Falcao", "García", 33,
            true, 15);
        delanteros[1] = new Delantero("Duvan", "Zapata", 28, true, 12);
        /* Crea el equipo pasando como parámetros cada jugador creado
            anteriormente */
        EquipoFútbol equipo = new EquipoFútbol("Selección mayores",
            "Colombia",
                técnico, portero, defensas, mediocampos, delanteros);
        equipo.imprimir(); // Muestra los datos del equipo de fútbol
    }
}
```

Diagrama de clases

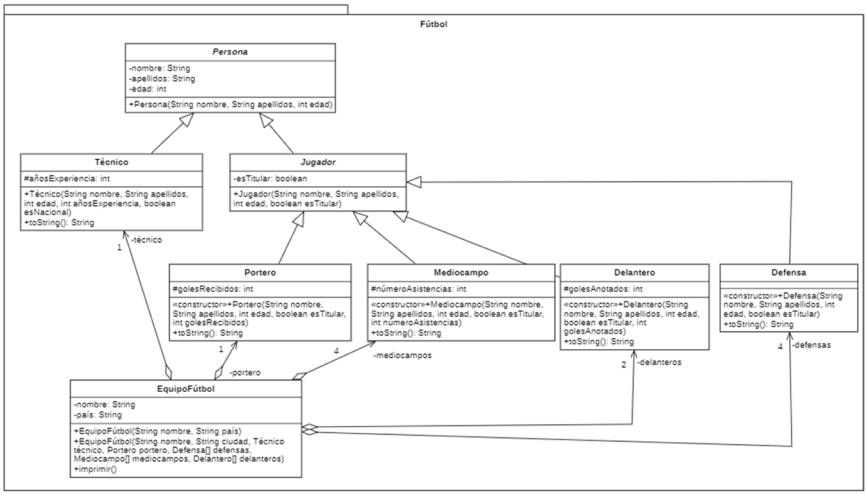


Figura 5.16. Diagrama de clases del ejercicio 5.6.

Explicación del diagrama de clases

Se ha definido un paquete denominado "Fútbol" que incluye una jerarquía de clases cuya clase raíz es Persona. La clase *Persona* tiene dos subclases: *Jugador* y Técnico. A su vez, los jugadores se clasifican en: Portero, Defensa, Mediocampo y Delantero. Las clases *Persona* y *Jugador* son abstractas debido a que no se van a tener objetos específicos de este tipo. Como ya se ha mencionado en capítulos anteriores, las clases abstractas se identifican en UML porque tienen su nombre en *cursiva*. En el programa se tendrán instancias de jugadores con un tipo concreto.

Las relaciones de herencia entre clases se identifican por una línea continua con un extremo que tiene un triángulo adyacente a la clase que identifica a la clase padre. La clase en el otro extremo de la línea es la hija.

El diagrama también muestra una clase denominada EquipoFútbol, la cual está relacionada con los diferentes tipos de jugadores por medio de una relación de agregación. Una relación de agregación se expresa por medio de una línea continua con un extremo que presenta un rombo claro adyacente a una clase que representa un todo. Al otro extremo de la línea, la otra clase representa la parte que está integrada en el todo. De acuerdo con esta terminología, un equipo de fútbol está conformado por un por-

tero, cuatro defensas, cuatro mediocampos, dos delanteros y un técnico (de acuerdo con la multiplicidad presentada al extremo de cada relación de agregación).

La relación de agregación expresa una relación semántica débil entre las clases. Por lo tanto, un equipo de fútbol contendrá diferentes tipos de jugadores, pero ellos existen independientemente del equipo. Si la clase EquipoFútbol se elimina, sus jugadores aún siguen existiendo. Así, cada clase presenta su colección de atributos, un constructor y métodos correspondientes.

Diagrama de objetos

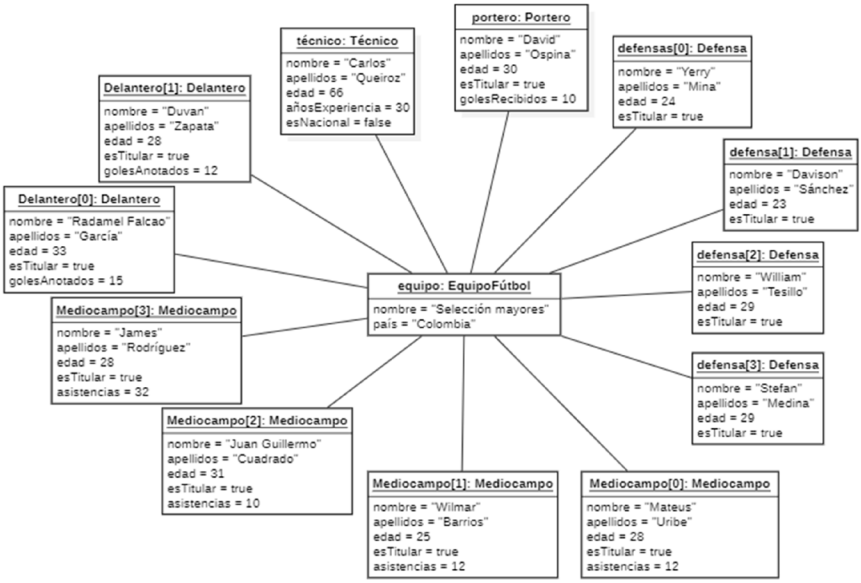


Figura 5.17. Diagrama de objetos del ejercicio 5.6.

Ejecución del programa

```
Nombre del equipo = Selección mayores
País = Colombia
Portero [Nombre = David, Apellidos = Ospina, Edad = 30, Goles recibidos = 10]
Defensas
Nombre = Yerry, Apellidos = Mina, Edad = 24
Nombre = Davison, Apellidos = Sánchez, Edad = 23
Nombre = William, Apellidos = Tesillo, Edad = 29
Nombre = Stefan, Apellidos = Medina, Edad = 29
Medicampo
Nombre = Mateus, Apellidos = Uribe, Edad = 28, Asistencias = 12
Nombre = Wilmar, Apellidos = Barrios, Edad = 25, Asistencias = 12
Nombre = Juan Guillermo, Apellidos = Cuadrado, Edad = 31, Asistencias = 10
Nombre = James, Apellidos = Rodríguez, Edad = 28, Asistencias = 32
Delanteros
Nombre = Radamel Falcao, Apellidos = García, Edad = 33, Goles anotados = 15
Nombre = Duvan, Apellidos = Zapata, Edad = 28, Goles anotados = 12
```

Figura 5.18. Ejecución del programa del ejercicio 5.6.

Ejercicios propuestos

- Modificar el programa anterior para que soporte la administración de partidos de un campeonato de fútbol. El campeonato estará conformado por diez equipos. Cada equipo puede jugar contra otro y se debe registrar el marcador del partido. Además, el programa debe generar una tabla de posiciones con la cantidad de puntos obtenidos por cada equipo, la cantidad de partidos ganados, empatados y perdidos, los goles a favor y en contra y, finalmente, la diferencia de goles. La tabla debe estar ordenada de acuerdo con la cantidad de puntos obtenidos por cada equipo.

Ejercicio 5.7. Significado de la relación de agregación

La relación de agregación representa una relación TIENE_UN (*has_a*). Esta facilita la reutilización del código. La herencia se debe usar solo si la relación se lee como ES_UN (*is_a*) y se mantiene durante toda la vida útil de los objetos involucrados; de lo contrario, la agregación es la mejor opción.

La agregación y composición son dos formas de asociación como se observa en la figura 5.19.

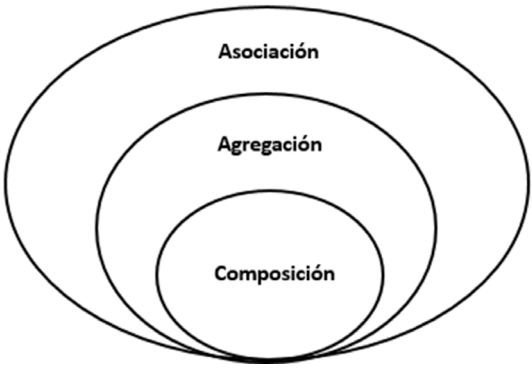


Figura 5.19. Relaciones de asociación, agregación y composición

Objetivo de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para modelar e implementar correctamente la relación de agregación entre clases.

Enunciado: clase Departamento y Municipio

La gobernación de un departamento desea gestionar la información sobre sus municipios. Las gobernaciones tienen un nombre que representa el departamento al que pertenecen. Así, se requiere desarrollar un programa que implemente las siguientes funcionalidades:

- Constructor de la clase Departamento.
- Métodos *get* y *set* para el atributo nombre del departamento.
- Agregar un municipio a lista de municipios de un departamento. Los datos de un municipio son: nombre del municipio, población total del municipio, temperatura media del municipio y altitud sobre el nivel del mar.
- Eliminar un municipio de la lista de municipios de la gobernación del departamento.
- Buscar un municipio en la lista de municipios del departamento. Si lo encuentra muestra en pantalla los datos del municipio. De lo contrario, se muestra el mensaje correspondiente.
- Consultar si dada una población específica, existen municipios con una población mayor o igual a un valor dado: si existen municipios que cumplen dicha condición, se muestra en pantalla el

nombre de municipio. Si no encuentra municipios que cumplan el criterio de búsqueda se mostrará el mensaje correspondiente.

- Cálculo del censo del departamento: calcular y mostrar en pantalla el total de población del departamento sumando las poblaciones de cada municipio.

Solución

Clase: Municipio

```
package Municipios;

/**
 * Esta clase denominada Municipio modela un municipio colombiano.
 * Tiene los siguientes atributos: un nombre de municipio, la población
 * del municipio, su temperatura media anual y su altitud sobre el nivel
 * del mar en metros.
 * @version 1.2/2020
 */
public class Municipio {
    // Atributo que define el nombre del municipio
    private String nombre;
    // Atributo que define la población del municipio
    private int población;
    /* Atributo que define la temperatura media anual del municipio en
       grados centígrados */
    private double temperaturaMedia;
    /* Atributo que define la altitud a la que se encuentra el municipio
       en metros */
    private double altitud;

    /**
    * Constructor de la clase Municipio
    * @param nombre Parámetro que define el nombre del municipio
    * @param población Parámetro que define la cantidad de habitantes
    * del municipio
    * @param temperaturaMedia Parámetro que define la temperatura
    * media anual del municipio
    * @param altitud Parámetro que define la altitud sobrel el nivel del
    * mar del municipio
    */
```

```
    public Municipio(String nombre, int población, double
        temperaturaMedia, double altitud) {
        this.nombre = nombre;
        this.población = población;
        this.temperaturaMedia = temperaturaMedia;
        this.altitud = altitud;
    }

    /**
    * Método que devuelve el nombre del municipio
    * @return El nombre del municipio
    */
    public String getNombre() {
        return nombre;
    }

    /**
    * Método que establece el nombre del municipio
    * @param nombre Parámetro que define el nombre del municipio
    */
    private void setNombre(String nombre) {
        this.nombre = nombre;
    }

    /**
    * Método que devuelve la población del municipio
    * @return La población del municipio
    */
    public int getPoblación() {
        return población;
    }

    /**
    * Método que establece la población del municipio
    * @param población Parámetro que define la población del municipio
    */
    private void setPoblación(int población) {
        this.población = población;
    }

    /**
    * Método que devuelve la temperatura media del municipio
    * @return La temperatura media del municipio
```

```
        */
        public double getTemperaturaMedia() {
            return temperaturaMedia;
        }

        /**
        * Método que establece la temperatura media del municipio
        * @param temperaturaMedia Parámetro que define la temperatura
        * media del municipio
        */
        private void setTemperaturaMedia(double temperaturaMedia) {
            this.temperaturaMedia = temperaturaMedia;
        }

        /**
        * Método que devuelve la altitud del municipio
        * @return La altitud del municipio
        */
        public double getAltitud() {
            return altitud;
        }

        /**
        * Método que establece la altitud del municipio
        * @param altitud Parámetro que define la altitud media del municipio
        */
        private void setAltitud(double altitud) {
            this.altitud = altitud;
        }

        /**
        * Método que muestra en pantalla los datos del municipio
        */
        public void imprimir() {
            System.out.println("Nombre = " + getNombre());
            System.out.println("Población = " + getPoblación());
            System.out.println("Temperatura promedio (C) = " +
                getTemperaturaMedia());
            System.out.println("Altitud (metros) = " + getAltitud());
        }
    }
```

Clase: Departamento

```
package Municipios;
import java.util.*;

/**
 * Esta clase denominada Departamento modela un departamento
 * colombiano con un nombre y un vector de municipios.
 * @version 1.2/2020
 */
public class Departamento {
    String nombre; // Atributo que identifica el nombre de un departamento
    /* Atributo que identifica los municipios que pertenecen a un
       departamento */
    private Vector municipios;

    /**
    * Constructor de la clase Departamento
    * @param nombre Parámetro que define el nombre del departamento
    * @param municipios Parámetro que define los municipios que
    * hacen parte del departamento
    */
    public Departamento(String nombre, Vector municipios) {
        this.nombre = nombre;
        this.municipios = municipios;
    }

    /**
    * Método que devuelve el nombre del departamento
    * @return El nombre del departamento
    */
    public String getNombre() {
        return nombre;
    }

    /**
    * Método que establece el nombre del departamento
    * @param nombre Parámetro que define el nombre del departamento
    */
    private void setNombre(String nombre) {
        this.nombre = nombre;
    }
```

```
/**
* Método que agrega un municipio al vector de municipios del
* departamento
* @param municipio Parámetro que define el municipio a agregar al
* vector de municipios del departamento
*/
public void agregarMunicipio(Municipio municipio) {
    municipios.add(municipio);
}

/**
* Método que elimina un municipio del vector de municipios del
* departamento
* @param nombre Parámetro que define el nombre del municipio a
* eliminar
*/
public void eliminarMunicipio(String nombre) {
    Municipio municipio;
    for (int i = 0; i < municipios.size(); i++) { /* Se debe recorrer el
        vector de municipios */
    // Se obtiene un objeto municipio aplicando el proceso de casting
        municipio = (Municipio) municipios.elementAt(i);
        // Si el nombre del municipio obtenido es igual al buscado
        if (nombre.equals(municipio.getNombre())) {
            municipios.remove(municipio); /* Elimina el municipio
                del vector */
            break; // No es necesario seguir buscando el municipio
        }
    }
}

/**
* Método que busca un municipio a partir de un nombre
* @param nombre Parámetro que define el nombre del municipio a
* buscar
*/
public void buscarMunicipio(String nombre) {
    Municipio municipio;
    for (int i = 0; i < municipios.size(); i++) { /* Se debe recorrer el
        vector de municipios */
        // Se obtiene un objeto municipio aplicando el proceso de casting
```

```
            municipio = (Municipio) municipios.elementAt(i);
            // Si el nombre del municipio obtenido es igual al buscado
            if (nombre.equals(municipio.getNombre())) {
                municipio.imprimir(); /* Se muestra en pantalla los datos
                    del municipio encontrado */
                break; // No es necesario seguir explorando el vector de
                    municipios
            }
            if (!nombre.equals(municipio.getNombre())) { /* Si el
                municipio no se encuentra */
                System.out.println("Municipio no encontrado.");
            }
        }
    }

    /**
    * Método que busca municipios con una población mayor o igual a
    * un valor dado como parámetro
    * @param población Parámetro que define una determinada
    * cantidad de población
    */
    public void buscarMunicipioConPoblaciónMayor(int población) {
        Municipio municipio;
        boolean encontróMunicipios = false;
        for (int i = 0; i < municipios.size(); i++) { /* Se debe recorrer el
            vector de municipios */
        // Se obtiene un objeto municipio aplicando el proceso de casting
            municipio = (Municipio) municipios.elementAt(i);
            /* Si la población del municipio obtenido es mayor o igual que
                la población ingresada */
            if (municipio.getPoblación() >= población) {
                System.out.println(municipio.getNombre()); /* Se
                    imprime el nombre del municipio */
                encontróMunicipios = true;
            }
        }
        /* En caso de no encontrar municipios que no tengan una
            población igual o mayor a la ingresada */
        if (!encontróMunicipios) {
```

```
                System.out.println("No existen municipios con esta población");
            }
        }

        /**
        * Método que calcula el censo poblacional del departamento
        * sumando la población de todos los municipios del departamento
        * @return La población total del departamento
        */
        public int calcularCensoPoblaciónDepartamento() {
            Municipio municipio;
            int totalCenso = 0; // El total a calcular se inicializa en cero
            for (int i = 0; i < municipios.size(); i++) { /* Se recorre el vector
                de municipios */
                // Se obtiene un municipio utilizando el proceso de casting
                municipio = (Municipio) municipios.elementAt(i);
                totalCenso += municipio.getPoblación(); /* Se acumula el
                    total de población del municipio */
            }
            return totalCenso; // Devuelve el total calculado
        }
    }
```

Clase: Prueba

```
package Municipios;
import java.util.*;

/**
 * Esta clase prueba las clases Departamento y Municipio.
 * @version 1.2/2020
 */
public class Prueba {

    /**
    * Metodo main que crea un departamento. Luego, crea varios
    * municipios que se agregan al departamento y luego calcula el
    * censo de población del departamento, muestra los municipios
    * con una población mayor a un valor dado y busca un municipio
    * determinado
    */
```

```
        public static void main(String[] args) {
            Municipio municipio1 = new Municipio("Manizales", 400000,
                17, 2200);
            Municipio municipio2 = new Municipio("La Dorada", 98000, 28,
                1000);
            Municipio municipio3 = new Municipio("Neira", 30000, 20,
                1969);
            Vector municipios = new Vector();
            municipios.add(municipio1);
            municipios.add(municipio2);
            municipios.add(municipio3);
            Departamento departamento = new Departamento("Caldas",
                municipios);
            System.out.println("El censo del Departamento de " +
                departamento.getNombre() + " es = " + departamento.
                calcularCensoPoblaciónDepartamento());
            System.out.println("Municipios con población mayor a 50.000");
            departamento.buscarMunicipioConPoblaciónMayor(50000);
            System.out.println("Búsqueda del municipio de Manizales");
            departamento.buscarMunicipio("Manizales");
        }
    }
```

Diagrama de clases

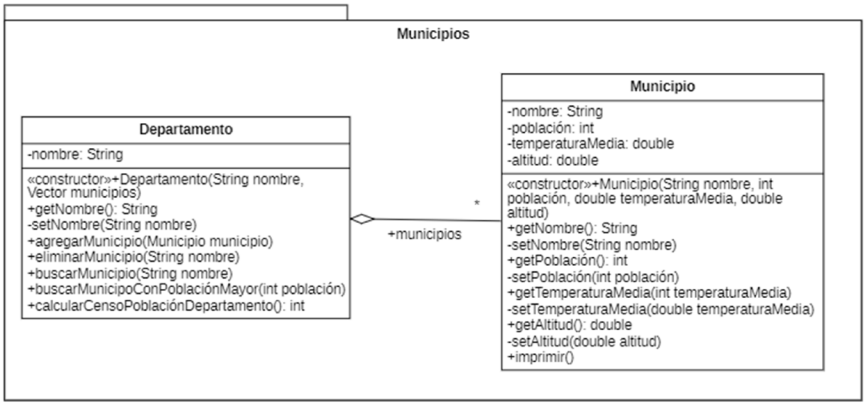


Figura 5.20. Diagrama de clases del ejercicio 5.7.

Se ha definido un paquete denominado "Municipios" con dos clases: Departamento y Municipio. La clase Departamento contiene un vector de municipios a través de una relación de agregación. Esta se expresa por medio de una línea continua con un extremo con un rombo claro. La clase adyacente al rombo es la clase que representa el todo y la clase que se encuentra en el extremo contrario representa la parte contenida en el todo. Por lo tanto, un municipio está contenido dentro de un Departamento y un Departamento tiene muchos municipios de acuerdo con la multiplicidad de la relación de agregación. Así, de acuerdo con la relación de agregación, la relación semántica entre Departamento y Municipio es débil, es decir, que un municipio puede existir independientemente de un departamento.

El sentido de la flecha representa la navegabilidad de la relación en sentido de Departamento a Municipio, lo que significa que un departamento conoce los municipios que contiene, pero un municipio no sabe a qué departamento pertenece.

En primer lugar, la clase Departamento tiene un único atributo privado: el nombre del departamento. La clase cuenta con un constructor, métodos *get* y *set* para cada atributo y los métodos que permiten manipular el vector de municipios, a saber: agregar un municipio, buscar un municipio y eliminar un municipio. Además, tiene métodos para buscar un municipio con una población mayor a un valor dado y para calcular el censo total de la población del departamento.

En segundo lugar, la clase Municipio cuenta con los atributos: nombre del municipio, población del municipio, temperatura media del municipio y altitud sobre el nivel del mar del municipio. Así mismo, la clase cuenta con un constructor, métodos *get* y *set* para cada atributo y un método *imprimir* para mostrar los datos del municipio en pantalla.

Diagrama de objetos

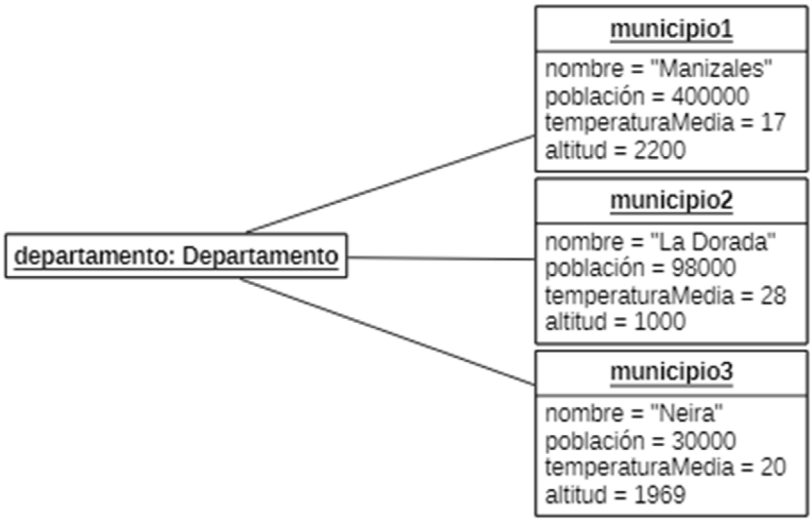


Figura 5.21. Diagrama de objetos del ejercicio 5.7.

Ejecución del programa

```
El censo del Departamento de Caldas es = 528000
Municipios con población mayor a 50.000
Manizales
La Dorada
Búsqueda del municipio de Manizales
Nombre = Manizales
Población = 400000
Temperatura promedio (C) = 17.0
Altitud (metros) = 2200.0
```

Figura 5.22. Ejecución del programa del ejercicio 5.7.

Ejercicios propuestos

- Una persona aficionada a las series de televisión requiere un programa que permita almacenar y consultar sus series. Las series tienen los siguientes atributos: título, género, país de origen y año de

realización. El programa debe permitir agregar y eliminar series. Además, debe permitir la consulta de series por su título, género y país de origen.

- Un evaluador de artículos científicos debe evaluar varios artículos que llegan a su correo electrónico. Un artículo científico está conformado por varias secciones y a su vez las secciones están conformadas por subsecciones y las subsecciones por párrafos. El evaluador puede agregar comentarios o no a cada sección, subsección o párrafo. Al finalizar la evaluación, el evaluador emite un concepto sobre el artículo el cual puede ser: aceptado sin modificaciones, aceptado con modificaciones o rechazado.

Capítulo 6
Genericidad, excepciones y lectura/escritura de archivos

El propósito general del sexto capítulo es presentar un conjunto de aspectos adicionales que son importantes para desarrollar programas con calidad, gestionando correctamente posibles errores que se generan durante su ejecución y que permitan el almacenamiento persistente de los datos de los objetos generados. En este capítulo se presentan once ejercicios relacionados con la genericidad de clases, tratamiento de excepciones, y lectura y escritura de archivos en Java.

- **Ejercicio 6.1. Clases genéricas**
  La genericidad es un concepto alterno a la herencia que permite a los programadores especificar un conjunto de tipos relacionados. El primer ejercicio aborda la definición y uso de clases genéricas.

- **Ejercicio 6.2. Métodos genéricos**
  Los métodos genéricos permiten a los programadores especificar con una sola declaración de un método un conjunto de métodos relacionados a través de diferentes tipos relacionados. El segundo ejercicio plantea el uso de métodos genéricos.

- **Ejercicio 6.3. *Array* de elementos genéricos**
  El concepto de genericidad también se puede aplicar a estructuras como los *arrays*. Por lo tanto, se pueden definir

arrays cuyos elementos no son de un tipo específico, sino que sus elementos son genéricos. El tercer ejercicio está orientado a la definición y aplicación de *arrays* genéricos.

- **Ejercicio 6.4. Excepciones**
  Generalmente, los programas generan errores en tiempo de ejecución debido a múltiples causas. Un programa que esté diseñado correctamente debe gestionar adecuadamente la aparición de dichos errores. La gestión de errores se realiza en Java por medio de objetos denominados excepciones. El cuarto ejercicio propone la definición de excepciones para el tratamiento de errores en un programa teniendo en cuenta sus bloques *try*, *catch* y *finally*.

- **Ejercicio 6.5. Lanzamiento de excepciones**
  Las excepciones en Java se pueden lanzar en forma explícita por medio de la sentencia *throw*. El quinto ejercicio presenta un problema, se debe aplicar esta sentencia para lanzar excepciones específicas.

- **Ejercicio 6.6. *Catchs* múltiples**
  Las excepciones que ocurren durante la ejecución de cierto bloque de un programa pueden ser de diversos tipos. Para capturar un tipo específico de excepción se utiliza la sentencia *catch*. El sexto ejercicio muestra la definición y aplicación de *catch* múltiples.

- **Ejercicio 6.7. Validación de campos**
  Una de las numerosas aplicaciones de las excepciones es que permiten que el programador pueda validar los campos ingresados por teclado durante la ejecución del programa. El séptimo ejercicio permite validar los campos de ingreso de datos de un cierto programa para que cumpla ciertos requerimientos.

- **Ejercicio 6.8. Lectura de archivos**
  Los archivos son el mecanismo más común para guardar datos en un sistema informático. Una situación común es que un programa lea un archivo de texto y presente su información al usuario. La lectura de estos archivos requiere de objetos especializados. El octavo ejercicio presenta un problema de lectura de archivos de texto.

- **Ejercicio 6.9. Escritura de archivos**
  Los archivos de texto se pueden también crear y guardar en el sistema de archivos de un computador. Para crear y guardar un archivo de texto se requiere trabajar con objetos especializados. El noveno ejercicio pretende aplicar estos objetos para crear y guardar un archivo con una información específica.

- **Ejercicio 6.10. Lectura de directorios**
  Los archivos están almacenados en estructuras o carpetas denominados directorios. El décimo ejercicio permite leer un directorio específico extrayendo información acerca de sus contenidos.

- **Ejercicio 6.11. Lectura/escritura de objetos**
  Los objetos creados durante la ejecución de los programas deben tener la capacidad de ser persistentes, es decir, que, si el programa termina y el terminal se apaga, su información puede seguir existiendo ya que queda almacenada. El último ejercicio de este capítulo ayuda a entender y aplicar el concepto de serialización de objetos que permite que la información de los objetos se guarde en un sistema de archivos.

Los diagramas UML utilizados en este capítulo son los diagramas de clase, de máquinas de estados y de objetos. Los paquetes permiten agrupar elementos UML y en los ejercicios se organizarán las clases en un único paquete debido a la pequeña cantidad de clases incluidas en las soluciones presentadas. Los diagramas de clase modelan la estructura estática de los programas. Así, los diagramas de clase de los ejercicios presentados, generalmente, estarán conformados por una o varias clases relacionadas e incorporan notaciones para identificar tipos de atributos y métodos. Los diagramas de máquinas de estado modelan el comportamiento presentado con la generación de excepciones en los programas y los diagramas de objetos identifican los objetos creados en el método *main* del programa.

Ejercicio 6.1. Clases genéricas

Una clase genérica o plantilla es una clase que se parametriza sobre tipos. Esencialmente, los tipos genéricos permiten escribir una clase o método genérico general que funciona con diferentes tipos, lo que permite la reutilización del código (Deitel y Deitel, 2017).

En lugar de especificar que un objeto sea de un tipo *int*, *String*, o de cualquier otro tipo, se define la clase como genérica con un parámetro genérico de tipo <T>. Las clases genéricas permiten especificar, con una sola declaración de clase, un conjunto de tipos relacionados (Schildt, 2018). El formato para la definición de una clase genérica es el siguiente:

```
class ClaseGenérica<T> {
}
```

T es el parámetro genérico.

Por convención, los nombres de los parámetros de tipo son letras mayúsculas y únicas. Los nombres de parámetros de tipo más utilizados son:

- E: elemento
- K: clave
- N: número
- T: tipo
- V: valor
- S, U, V, etc. 2°, 3° 4° tipo.

Se debe tener en cuenta que los parámetros de tipo solo pueden representar *wrappers*, no tipos primitivos de datos (como *int*, *double, char,* etc.).

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Crear clases genéricas que poseen diferentes elementos genéricos de diversos tipos pasados como parámetros.
- Instanciar objetos que poseen elementos de diferentes tipos parametrizados.

Enunciado: clase Tripleta

Se requiere desarrollar una clase genérica Tripleta que tiene 3 elementos genéricos. La clase debe poseer:

- Un constructor que inicialice una tripleta con sus tres valores genéricos.

- Un método *toString()* que muestre en pantalla los valores de sus tres atributos.
- Un método *main* que cree tres tripletas con los siguientes valores de la tabla 6.1

Tabla 6.1. Tripletas para crear en el método *main*

Tripleta	Par. 1	Valor	Par. 2	Valor	Par. 3	Valor
1	*Integer*	1	*String*	"Cálculo"	*String*	"Cómputo, cuenta o investigación que se hace de algo por medio de operaciones matemáticas".
2	*String*	"Real Madrid"	*String*	"Barcelona"	*String*	"3-3"
3	*String*	"Temperatura"	*String*	"Grados centígrados"	*Integer*	18

Solución

Clase: Tripleta

```
package Genericidad;
/**
 * Esta clase denominada Tripleta modela una clase genérica que
 * representa una tripleta (una colección con tres elementos). Los
 * atributos de la tripeta son tres elementos genéricos denominados i, p, s.
 * @version 1.2/2020
 */
class Tripleta<I, P, S>  {
    private final I i; /* Atributo que identifica el primer elemento de la
        tripleta */
    private final P p; /* Atributo que identifica el segundo elemento de
        la tripleta */
    private final S s; /* Atributo que identifica el tercer elemento de la
        tripleta */

    /**
     * Constructor de la clase Tripleta
```

```
* @param valor1 Parámetro que define el primer elemento de la
* tripleta
* @param valor2 Parámetro que define el segundo elemento de la
* tripleta
* @param valor3 Parámetro que define el tercer elemento de la
* tripleta
*/
public Tripleta(I valor1, P valor2, S valor3) {
    i = valor1;
    p = valor2;
    s = valor3;
}

/**
* Método que convierte a String los datos de una tripleta
* @return Un String que concatena los datos de la tripleta
*/
public String toString() {
    return String.format(i + " - " + p + " - " + s);
}

/**
* Método main que crea dos tripletas y muestra su información en
* pantalla
*/
public static void main(String[] args) {
    // Crea una tripleta conformada por un Integer y dos String
    Tripleta<Integer, String, String> tripleta1 = new Tripleta<Integer,
        String, String>(1, "Cálculo", "Cómputo, cuenta o
        investigación que se hace de algo por medio de operaciones
        matemáticas.");
    System.out.println(tripleta1);

    // Crea una tripleta conformada por tres String
    Tripleta<String, String, String> tripleta2 = new Tripleta<String,
        String, String> ("Real Madrid","Barcelona","3-3");
    System.out.println(tripleta2);

    // Crea una tripleta conformada por dos String y un Integer
    Tripleta<String, String, Integer> tripleta3 = new Tripleta<String,
        String, Integer> ("Temperatura","Grados centígrados",18);
```

```
            System.out.println(tripleta3);
        }
    }
```

Diagrama de clases

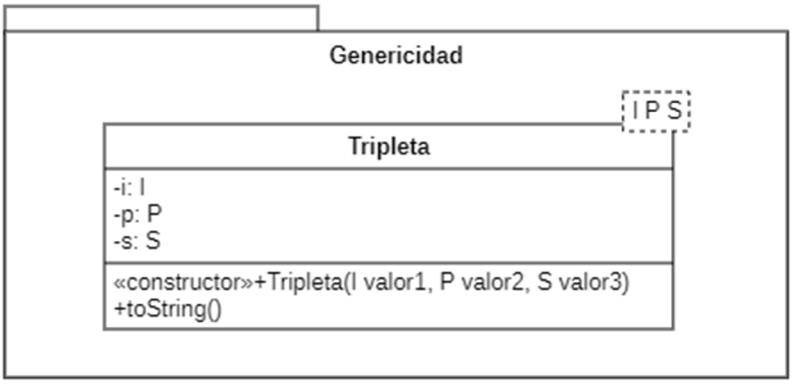


Figura 6.1. Diagrama de clases del ejercicio 6.1.

Explicación del diagrama de clases

Se ha definido un paquete denominado "Genericidad" con una única clase genérica: Tripleta. Las clases genéricas se representan en UML con la notación de clase a la que se le ha añadido una caja rectangular en la esquina superior derecha. La clase Tripleta contiene tres parámetros identificados con las letras I, P y S. También tiene tres atributos denominados i, p, s, que son genéricos, ya que no tienen asignado un tipo primitivo de dato y no son objetos especificados en forma explícita referenciando la clase a la que pertenecen.

La clase Tripleta cuenta con un constructor y un método denominado *toString* para obtener los tres elementos de la tripleta como un *String* concatenado.

Diagrama de objetos

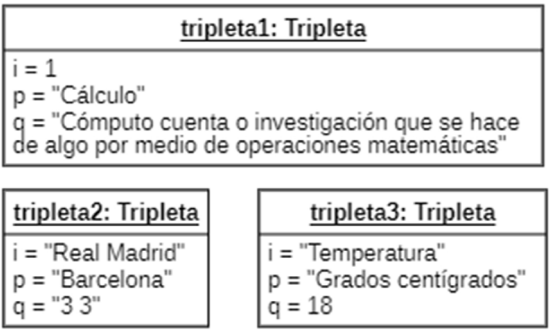


Figura 6.2. Diagrama de objetos del ejercicio 6.1.

Ejecuciones del programa

```
1 - Cálculo - Cómputo, cuenta o investigación que se hace de algo por medio de operaciones matemáticas.
Real Madrid - Barcelona - 3-3
Temperatura - Grados centígrados - 18
```

Figura 6.3. Ejecución del programa del ejercicio 6.1.

Ejercicios propuestos

- Modificar la clase genérica Tripleta para que sus tres valores sean del mismo tipo.
- Realizar una clase genérica que modele un Par de elementos con dos valores genéricos.

Ejercicio 6.2. Métodos genéricos

Un método genérico es un método que se puede invocar con argumentos de diferentes tipos (Arroyo-Díaz, 2019c). En función de los tipos de argumentos pasados al método genérico, el compilador maneja cada llamada de método de manera apropiada. Los métodos genéricos tienen el siguiente formato:

modificadoresAcceso <T> tipoRetorno nombreMétodo(argumentos) {
.....
}

Todas las declaraciones de métodos genéricos tienen una sección de parámetros de tipo delimitada por corchetes angulares (< >) que preceden al tipo de retorno del método.

Cada sección de parámetros contiene uno o más parámetros de tipo separados por comas. Un parámetro de tipo, también conocido como variable de tipo, es un identificador que especifica un nombre de tipo genérico.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Definir métodos genéricos que reciban tanto parámetros genéricos como valores de retorno genéricos.
- Utilizar la interfaz Comparable para comparar elementos genéricos.

Enunciado: clase MáximosMínimos

Se requiere desarrollar un programa que tenga dos métodos genéricos, uno para determinar el máximo de dos valores y otro, el mínimo de dos valores.

En un método *main* se deben pasar los siguientes valores para determinar su máximo y mínimo:

- Valores enteros: 10 y 20
- Valores reales: 12.56 y 28.71
- Valores *String*: "Tableta" y "Computador"

La clase debe utilizar por medio de herencia la interface predefinida *Comparable*.

Instrucciones Java del ejercicio

Tabla 6.2. Instrucciones Java del ejercicio 6.2.

Instrucción	Descripción	Formato
Comparable<T>	Esta interfaz impone un orden total en los objetos de cada clase que la implementa. El método *compareTo* se conoce como su método de comparación natural.	objeto1.compareTo(objeto2);

Solución

Clase: MáximoMínimo

```
package Genericidad;

/**
 * Esta clase denominada MáximoMínimo modela una clase que se
 * caracteriza por tener un método genérico que permite realizar
 * comparaciones de valores sin importar su tipo.
 * @version 1.2/2020
 */
public class MáximoMínimo {

    /**
    * Método estático que extiende la interfaz Comparable que permite
    * obtener el mayor de dos valores sin importar su tipo
    * @return Un valor genérico con el resultado del máximo de la
    * comparación
    */
    public static <E extends Comparable<E>> E máximo(E x, E y) {
        // Compara x con y
        if (y.compareTo(x) > 0) { /* Si el resultado es mayor que cero, y es
            el elemento mayor */
            return y;
        } else { /* Si el resultado es menor o igual que cero, x es el elemento
            mayor */
            return x;
        }
    }

    /**
    * Método estático que extiende la interfaz Comparable que permite
    * obtener el menor de dos valores sin importar su tipo
    * @return Un valor genérico con el resultado del mínimo de la
    * comparación
    */
    public static <E extends Comparable<E>> E mínimo(E x, E y) {
        // Compara x con y
```

```
            if (y.compareTo(x) < 0) { /* Si el resultado es menor que cero, y es
                el elemento menor */
                return y;
            } else { /* Si el resultado es menor o igual que cero, x es el elemento
                menor */
                return x;
            }
        }

        /**
        * Método main para probar los métodos genéricos para obtener el
        * máximo y mínimo de dos valores.
        */
        public static void main(String args[]) {
            // Obtiene el mayor de dos enteros
            System.out.printf("Máximo entre %d y %d es %d\n", 10, 20,
                máximo( 10, 20 ));
            // Obtiene el mayor de dos float
            System.out.printf("Máximo entre %.2f y %.2f es %.2f\n",12.56,
                28.71,máximo( 12.56, 28.71 ));
            // Obtiene el mayor de dos String
            System.out.printf("Máximo entre %s y %s es %s\n\n", "Tableta",
                "Computador",
                    máximo("Tableta", "Computador"));

            // Obtiene el menor de dos enteros
            System.out.printf("Mínimo entre %d y %d es %d\n", 10, 20,
                mínimo( 10, 20 ));
            // Obtiene el menor de dos float
            System.out.printf("Mínimo entre %.2f y %.2f es %.2f\n", 12.56,
                28.71, mínimo( 12.56, 28.71 ));
            // Obtiene el menor de dos String
            System.out.printf("Mínimo entre %s y %s es %s\n", "Tableta",
                "Computador", mínimo("Tableta", "Computador"));
        }
    }
```

Diagrama de clases

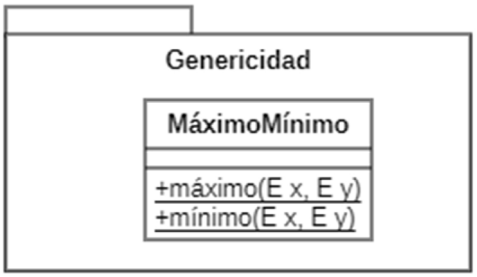


Figura 6.4. Diagrama de clases del ejercicio 6.2.

Explicación del diagrama de clases

Se ha definido un paquete denominado "Genericidad" con una única clase: MáximoMínimo. Esta tiene dos métodos públicos estáticos: máximo y mínimo. Los métodos estáticos se indican en UML con su texto en subrayado. Ambos métodos tienen dos parámetros genéricos que se identifican por la letra mayúscula E. Al ser métodos genéricos pueden recibir parámetros de diferente tipo.

Diagrama de objetos

No incluye diagrama de objetos ya que se invocan directamente los métodos estáticos, los cuales son compartidos por todos los objetos de la clase MáximoMínimo.

Ejecución de programas

```
Máximo entre 10 y 20 es 20
Máximo entre 12,56 y 28,71 es 28,71
Máximo entre Tableta y Computador es Tableta

Mínimo entre 10 y 20 es 10
Mínimo entre 12,56 y 28,71 es 12,56
Mínimo entre Tableta y Computador es Computador
```

Figura 6.5. Ejecución del programa del ejercicio 6.2.

Ejercicios propuestos

- Modificar el ejercicio de la Tripleta para que soporte un método permutar que realice todas las permutaciones posibles de sus tres elementos.

Ejercicio 6.3. *Array* de elementos genéricos

En Java también se permite crear *arrays* cuyos elementos son genéricos (Deitel y Deitel, 2017). El formato es el siguiente:

E[ ] array= new E[tamaño];

Entre sus restricciones se encuentran que no se puede crear una instancia de un *array* cuyos tipos de elemento sean genéricos y, además, no se puede crear un *array* de referencias genéricas específicas del tipo.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Definir *arrays* de elementos genéricos.
- Definir *arrays* con tipos primitivos de datos que serán enviados como parámetros a un método genérico.

Enunciado: clase ArrayElementosGenéricos

Se requiere un programa que defina una clase ArrayElementosGenéricos, esta tiene un método *imprimirArray*, que recibe como parámetro un *array* de elementos genéricos e *imprime* en pantalla los elementos del *array*.

En un método *main* se deben generar los siguientes *arrays* y se debe invocar al método *imprimirArray()* para mostrar sus elementos en pantalla.

- *Array*, tipo Integer, { 20, 25, 31 }
- *Array*, tipo Double, { 3.1415, 2.7182, 1.618 }
- *Array*, tipo Boolean, { true, false}
- *Array*, tipo Character, { 'I', 'n', 'f', 'o', 'r', 'm', 'á', 't', 'i', 'c', 'a' }
- *Array*, tipo *String*, {"Administración", "Sistemas", "Informáticos"}

Instrucciones Java del ejercicio

Tabla 6.3. Instrucciones Java del ejercicio 6.3.

Instrucción	Descripción	Formato

printf	Imprime un mensaje en pantalla utilizando una "cadena de formato" que incluye marcas para mezclar múltiples cadenas en la cadena final a mostrar. Las marcas pueden ser:
	• %d, %i: conversión decimal con signo de un entero.
	• %e, %E: conversión a coma flotante con signo en notación científica.
	• %f, %F: conversión a coma flotante con signo usando punto decimal.
	• %g, %G: conversión a coma flotante.
	• %s: cadena de caracteres (terminada en '\0').
		printf("%d", var);
		
		Ejemplo:
		int num = 28;
		int num2 = 12345;
		printf(„%d\n", num2);
		printf("%5d\n", num);
		
		Se imprime:
		12345
		28

Solución

Clase: ArrayElementosGenéricos

```
package Genericidad;

/**
 * Esta clase denominada ArrayElementosGenéricos modela una clase
 * que incluye un método genérico que imprime en pantalla los
 * elementos de un array genérico.
 * @version 1.2/2020
 */
public class ArrayElementosGenéricos {

    /**
    * Método estático que imprime en pantalla los elementos de un
    *array genérico
    */
    public static <E> void imprimirArray(E[] array) {
        /* Se recorre cada elemento del array donde sus elementos se
            indican con el parámetro genérico E */
        for(E ítem : array) {
            System.out.printf("%s ", ítem);
        }
        System.out.println();
    }

    /**
```

```
        * Método main que prueba el array genérico imprimiendo en
        * pantalla los valores de diferentes tipos del array
        */
        public static void main(String args[]) {
            // Imprime un array de datos enteros
            Integer[] intArray = { 20, 25, 31 };
            System.out.println("Array de enteros:");
            imprimirArray(intArray);

            // Imprime un array de datos double
            Double[] doubleArray = { 3.1415, 2.7182,  1.618};
            System.out.println("\nArray de doubles:");
            imprimirArray(doubleArray);

            // Imprime un array de datos boolean
            Boolean[] booleanArray = { true, false};
            System.out.println("\nArray de booleans:");
            imprimirArray(booleanArray);

            // Imprime un array de datos Character
            Character[] charArray = { 'I', 'n', 'f', 'o', 'r', 'm', 'á', 't', 'i', 'c', 'a' };
            System.out.println("\nArray de caracteres:");
            imprimirArray(charArray);

            // Imprime un array de datos String
            String[] stringArray = { "Administración","Sistemas",
                "Informáticos"};
            System.out.println("\nArray de strings:");
            imprimirArray(StringArray);
        }
    }
```

Diagrama de clases

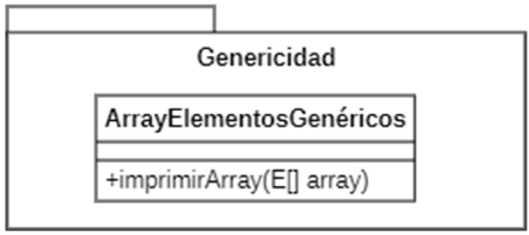


Figura 6.6. Diagrama de clases del ejercicio 6.3.

Explicación del diagrama de clases

Se ha definido un paquete denominado "Genericidad" con una única clase: ArrayElementosGenéricos. Esta clase tiene un único método público: imprimirArray, el cual se encargará de imprimir en pantalla los datos de un *array* de elementos genéricos. Los elementos genéricos del *array* se identifican con el parámetro genérico E.

Diagrama de objetos

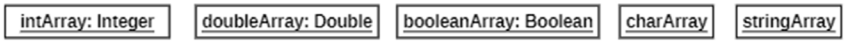


Figura 6.7. Diagrama de objetos del ejercicio 6.3.

Ejecución del programa

```
Array de enteros:
20 25 31

Array de doubles:
3.1415 2.7182 1.618

Array de booleans:
true false

Array de caracteres:
I n f o r m á t i c a

Array de strings:
Administración Sistemas Informáticos
```

Figura 6.8. Ejecución del programa del ejercicio 6.3.

Ejercicios propuestos

- Implementar una pila de elementos genéricos y métodos para apilar, desapilar, buscar un elemento y para determinar si la pila está vacía o llena.

Ejercicio 6.4. Excepciones

Las excepciones son un mecanismo especial para gestionar errores y permiten separar el tratamiento de errores del código normal de un programa (Kölling y Barnes, 2013).

El formato para escribir un bloque en el que se gestionan excepciones es:

```
try {
        instrucciones
} catch {
        instrucciones
} finally {
        instrucciones
}
```

Dentro del bloque *try* se coloca el código que podría generar una excepción. Los bloques *catch* capturan y tratan una excepción cuando esta ocurre. Pueden existir varios bloques *catch*. Estos se definen directamente después del bloque *try*. Ningún código puede estar entre el final del bloque *try* y el comienzo del primer bloque *catch*. Los *catch* se evalúan por orden, si un *catch* atrapa la excepción que ha ocurrido, se ejecuta y los demás no. Por último, el bloque *finallly* es opcional e incluye código que se ejecuta siempre, independientemente si se ha producido una excepción o no (Altadill-Izurra y Pérez-Martínez, 2017).

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Identificar bloques de código donde se pueden generar excepciones.
- Identificar los bloques *catch* que capturan una excepción específica.
- Reconocer y diferenciar los propósitos de los bloques *try*, *catch* y *finally* para la gestión de excepciones.

Instrucciones Java del ejercicio

Tabla 6.4. Instrucciones Java del ejercicio 6.4.

Instrucción	Descripción	Formato

try	Bloque de código en el que se intentará capturar una excepción si se produce y captura, se establece qué hacer con ella.	*try* {
		bloque donde puede ocurrir una excepción
		}

catch	Define un conjunto de instrucciones para tratar la excepción generada en el bloque *try* anterior.	*catch* (excepción e) {
		bloque donde se trata el problema
		}

finally	Bloque donde se pueden definir instrucciones necesarias tanto si hay o no excepciones (se ejecuta siempre).	*finally* {
		bloque que se ejecuta siempre
		}

Exception	Indica condiciones que una aplicación podría querer capturar y gestionar.	*Exception* e;
ArithmeticException	Esta excepción se lanza cuando ha ocurrido una condición aritmética excepcional. Por ejemplo, una división por cero.	*ArithmeticException* e;

Enunciado: clase PruebaExcepciones

¿Cuál es el resultado de la ejecución del método *main* del siguiente programa? Determinar qué se imprime en pantalla.

Solución

Clase: PruebaExcepciones

```
package Excepciones;
/**
 * Esta clase denominada PruebaExcepciones lanza diferentes
 * excepciones en situaciones específicas del programa. Los mensajes
 * que se muestran en pantalla ayudan a identificar la porción de código
 * que se ejecutó o no.
 * @version 1.2/2020
 */
public class PruebaExcepciones {
    /**
```

```
    * Método main con dos bloques try que generan excepciones que
    * deben ser gestionadas
    * @throws ArithmeticException Excepción aritmética de división
    * por cero
    * @throws Exception Excepción general
    */
    public static void main(String args[]) {
        // Primer bloque try
        try {
            System.out.println("Ingresando al primer try");
            double cociente = 10000/0; // Se lanza una excepción
            System.out.println("Después de la división"); /* Esta
                instrucción nunca será ejecutada */
        } catch (ArithmeticException e) { // Se captura la excepción
            System.out.println("División por cero"); /* Se imprime en
                pantalla este mensaje */
        } finally {
            /* La sentencia finally siempre se ejecuta, ocurra o no una
                excepción */
            System.out.println("Ingresando al primer finally");
        }

        // Segundo bloque try
        try {
            System.out.println("Ingresando al segundo try");
            Object objeto = null;
            objeto.toString(); // Se lanza una excepción
            /* Esta instrucción nunca será ejecutada porque se lanzó una
                excepción */
            System.out.println("Imprimiendo objeto");
        } catch (ArithmeticException e) { /* La excepción lanzada no es
            de este tipo */
            System.out.println("División por cero");
        } catch (Exception e) { // Se captura la excepción
            System.out.println("Ocurrió una excepción"); /* Se imprime
                en pantalla este mensaje */
        } finally {
            /* La sentencia finally siempre se ejecuta, ocurra o no una
                excepción */
            System.out.println("Ingresando al segundo finally");
        }
    }
}
```

Diagrama de clases

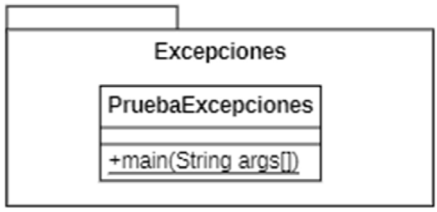


Figura 6.9. Diagrama de clases del ejercicio 6.4.

Explicación del diagrama de clases

Se ha definido un paquete denominado "Excepciones" con una única clase: PruebaExcepciones. Esta no tiene atributos, solamente contiene un método *main* con sus argumentos. El código del método *main* establecerá dos bloques *try* para la captura y tratamiento de posibles excepciones, que se pueden generar durante la ejecución del programa.

Diagrama de máquinas de estado

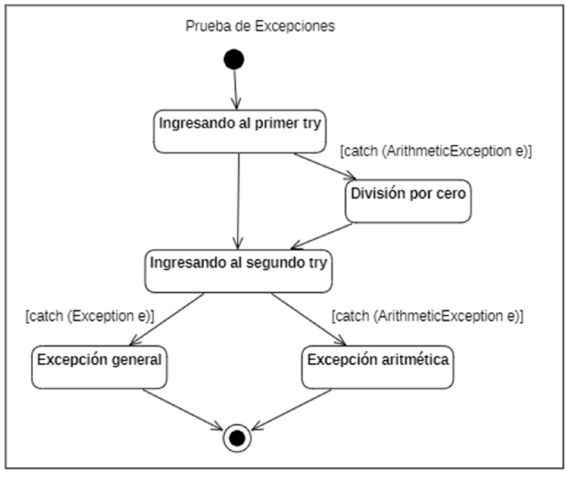


Figura 6.10. Diagrama de estados del ejercicio 6.4.

Explicación del diagrama de máquina de estado

Se ha agregado este tipo de diagrama UML, pero el diagrama de clases no expresa la lógica interna del método *main* donde se lanzan y gestionan posibles excepciones que ocurren durante la ejecución del programa. Un diagrama de máquina de estados permite entender los diferentes estados por los que pasa un programa a partir de los bloques *try*, que se están ejecutando y las posibles excepciones que pueden ocurrir.

El programa inicia pasando al primer estado denominado "Ingresando al primer *try*". Si se lanza y captura una excepción aritmética, el programa pasa al estado "División por cero". Así, si en el primer estado no ocurre una excepción o ya pasó por el estado de división por cero, el programa pasa inmediatamente al estado "Ingresando al segundo *try*". De igual manera que en el primer estado, si se lanza y captura una excepción (en este caso, excepción aritmética), el programa pasa al estado "Excepción aritmética". Si la excepción es general, el programa pasa al estado "Excepción general".

Ejecución del programa

```
Ingresando al primer try
División por cero
Ingresando al primer finally
Ingresando al segundo try
Ocurrió una excepción
Ingresando al segundo finally
```

Figura 6.11. Ejecución del programa del ejercicio 6.4.

Ejercicios propuestos

- ¿Cuál es el resultado de la ejecución del siguiente código?

```
class ExcepciónFueraLímite {
    public static void main(String args[]) {
        try {
            String texto = "Programación";
            char caracter = texto.charAt(14);
            System.out.println(caracter);
        }
```

```
            catch(StringIndexOutOfBoundsException e) {
                System.out.println("Indice de string por fuera del límite");
            }
        }
    }
```

- ¿Cuál es el resultado de la ejecución del siguiente código?

```
class ExcepciónFormatoNúmero {
    public static void main(String args[]) {
        try {
            int número = Integer.parseInt("Número");
            System.out.println(número);
        } catch(NumberFormatException e) {
            System.out.println("Excepción de formato de número");
        } finally {
            System.out.println("Ingresando al finally");
        }
    }
}
```

Ejercicio 6.5. Lanzamiento de excepciones

En Java, la sentencia *throw* se utiliza para lanzar una excepción de forma explícita desde el código. Al lanzar una excepción se interrumpirá el flujo de ejecución del programa y se buscará un código que la gestione apropiadamente en un bloque *catch* (Cha, 2016).

El formato de lanzamiento de excepciones es:

throw new Excepción(mensaje);

Cuando se lanza una excepción, nunca se ejecuta el código que está después de esta. Se inspecciona si el código incorpora una cláusula *catch* cuyo tipo coincide con el tipo de la excepción lanzada. Si se encuentra, el control se transfiere a dicha sentencia. Si no, se inspecciona el siguiente bloque *try* que la engloba y así sucesivamente (Schildt y Skrien, 2013).

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Lanzar excepciones específicas en métodos de una clase.
- Conocer y aplicar la sentencia *throw* para el lanzamiento de excepciones.

Enunciado: clase Vendedor

Se requiere implementar una clase vendedor que posee los siguientes atributos: nombre (tipo *String*), apellidos (tipo *String*) y edad (tipo *int*).

La clase contiene un constructor que inicialice los atributos de la clase. Además, la clase posee los siguientes métodos:

- Imprimir: muestra por pantalla los valores de sus atributos.
- Verificar edad: este método recibe como parámetro un valor entero que representa la edad del vendedor. Para que un vendedor pueda desempeñar sus labores se requiere que sea mayor de edad (mayor de 18 años). Si esta condición no se cumple, se lanza una excepción de tipo *IllegalArgumentException* con el mensaje "El vendedor debe ser mayor de 18 años". Además, se evalúa si la edad se encuentra en el rango de 0 a 120, si no se cumple, se genera una excepción de tipo *IllegalArgumentException* con el mensaje "La edad no puede ser negativa ni mayor a 120". Si la edad cumple estos requerimientos se pueden instanciar el objeto vendedor.

Además, se requiere que los datos del vendedor se ingresen por teclado.

Instrucciones Java del ejercicio

Tabla 6.5. Instrucciones Java del ejercicio 6.5.

Instrucción	Descripción	Formato
IllegalArgumentException	Se lanza esta excepción para indicar que un método ha pasado un argumento ilegal o inapropiado.	IllegalArgumentException e;

Solución

Clase: Vendedor

```
package Excepciones;
import java.util.*;

/**
 * Esta clase denominada Vendedor modela un vendedor que tiene
 * como atributos un nombre, apellidos y una edad.
 * @version 1.2/2020
 */
public class Vendedor {
    String nombre; // Atributo que identifica el nombre de un vendedor
    String apellidos; // Atributo que identifica los apellidos de un vendedor
    int edad; // Atributo que identifica la edad de un vendedor

    /**
    * Constructor de la clase Vendedor
    * @param nombre Parámetro que define el nombre de un vendedor
    * @param nombre Parámetro que define la edad de un vendedor
    */
    Vendedor(String nombre, String apellidos) {
        this.nombre = nombre;
        this.apellidos = apellidos;
    }

    /**
    * Método que muestra en pantalla los datos de un vendedor
    */
    void imprimir() {
        System.out.println("Nombre del vendedor = " + nombre);
        System.out.println("Apellidos del vendedor = " + apellidos);
        System.out.println("Edad del vendedor = " + edad);
    }

    /**
    * Método que verifica que la edad de un vendedor es apropiada; si
    * no lo es, se generan las excepciones correspondientes
    * @throws IllegalArgumentException Excepción de argumento ilegal
    * cuyo valor debe ser mayor que 18, ni negativo, ni mayor a 120
    */
```

```
    void verificarEdad(int edad) {
        if (edad < 18) { /* El vendedor debe tener una edad mayor de 18
            años */
            // Se genera una excepción de argumento ilegal
            throw new IllegalArgumentException("El vendedor debe ser
                mayor de 18 años.");
        } if (edad >= 0 && edad < 120) { /* El vendedor debe tener una
            edad entre 0 y 120 */
            this.edad = edad;
        } else throw new IllegalArgumentException("La edad no puede
            ser negativa ni mayor a 120.");
        /* En cualquier otro caso se genera una excepción de argumento
            ilegal */
    }

    /**
    * Método que solicita por teclado el nombre, apellidos y edad de un
    * vendedor. Luego, se verifica la edad. Si la edad no es correcta, se
    * generan las excepciones correspondientes.
    */
    public static void main(String[] args) {
        Scanner teclado = new Scanner(System.in);
        System.out.println("Nombre del vendedor = ");
        String n = teclado.next();
        System.out.println("Apellidos del vendedor = ");
        String a = teclado.next();
        Vendedor vendedor = new Vendedor(n, a);
        System.out.println("Edad del vendedor = ");
        int e = teclado.nextInt();
        vendedor.verificarEdad(e);
        vendedor.imprimir();
    }
}
```

Diagrama de clases

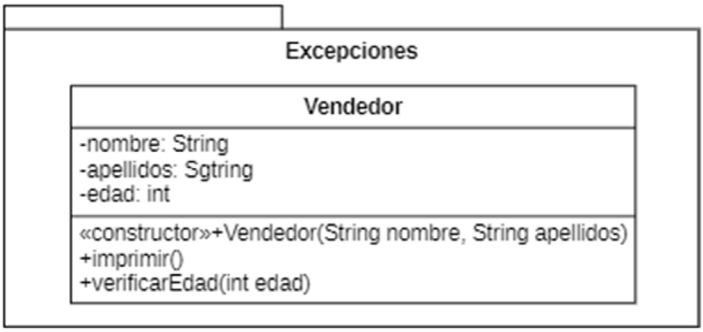


Figura 6.12. Diagrama de clases del ejercicio 6.5.

Explicación del diagrama de clases

Se ha definido un paquete denominado "Excepciones" con una única clase: Vendedor. Esta clase posee tres atributos privados: *nombre*, *apellidos* y *edad*. La clase tiene un constructor y dos métodos públicos: uno para *imprimir* los atributos de un vendedor y otro para verificar si la edad del vendedor cumple las restricciones del programa, de lo contrario, el método genera las excepciones adecuadas.

Diagrama de máquinas de estado

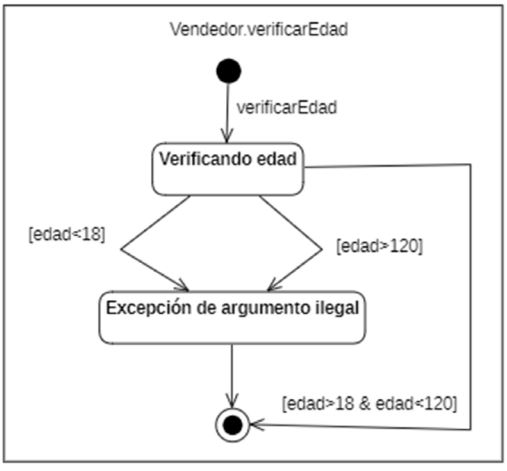


Figura 6.13. Diagrama de estados del ejercicio 6.5.

Explicación del diagrama de máquinas de estado

Se ha agregado este tipo de diagrama UML. El diagrama de clases no expresa la lógica interna del método verificarEdad de la clase Vendedor donde se pueden lanzar y gestionar posibles excepciones que ocurren durante la ejecución del programa.

El método comienza a ejecutarse y pasa al estado "Verificando edad". Si la edad es menor a 18 o mayor a 120, pasa al estado "Excepción de argumento ilegal". Si la edad cumple el criterio de ser mayor a 18 y menor a 120, el programa continua su ejecución y termina.

Ejecución del programa

```
Nombre del vendedor =
Pedro
Apellidos del vendedor =
Perez
Edad del vendedor =
17

Can only enter input while your programming is running
java.lang.IllegalArgumentException: El vendedor debe ser mayor de 18 años.
        at Excepciones.Vendedor.verificarEdad(Vendedor.java:42)
        at Excepciones.Vendedor.main(Vendedor.java:62)
```

Figura 6.14. Ejecución del programa del ejercicio 6.5.

En este ejemplo se han ingresado los datos: "Pedro" como nombre del vendedor, "Pérez" como apellidos del vendedor y el valor numérico 17 como edad del vendedor. Como la edad del vendedor no cumple los criterios estipulados del programa (ser mayor a 18), entonces se genera una excepción que se muestra en la parte inferior del programa. La excepción generada es de tipo *IlegalArgumentException* y el texto descriptivo es el indicado en el código del programa: "El vendedor debe ser mayor de 18 años".

Ejercicios propuestos

- Escribir un programa con un constructor que lanza una excepción a un controlador de excepciones. El programa debe intentar crear un objeto y detectar la excepción que se genera desde el constructor.
- Implementar una clase con la tabla ASCII, cada símbolo tiene asociado un valor numérico. La clase tiene dos métodos:
  - *int get*(*String* símbolo): dado un símbolo recupera el número asociado.
  - *void set*(*String* símbolo, *int* número): asocia el número con el símbolo.

Se deben definir excepciones apropiadas, de tal manera que los métodos funcionen correctamente. Por ejemplo, cuando se requiere recuperar un símbolo inexistente o que el parámetro pasado sea un valor nulo.

Ejercicio 6.6. *Catchs* múltiples

En los bloques *try/catch/finally* se permite incluir múltiples *catch*, cada uno para gestionar un tipo específico de excepción. El orden de las sentencias *catch* es relevante. Se deben colocar primero los bloques *catch* que tratan excepciones específicas y luego las excepciones más generales (Flanagan y Evans, 2019). El formato es el siguiente:

```
try {
        instrucciones
} catch (Exception) {
        instrucciones
} catch (Exception) {
        instrucciones
} catch (Exception) {
        instrucciones
} finally {
        instrucciones
}
```

Para eliminar código duplicado se utiliza el símbolo |.

```
try {
        instrucciones
} catch (Exception | Exception) {
        instrucciones
} finally {
        instrucciones
}
```

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Definir múltiples *catch* para el tratamiento de excepciones.
- Definir gestores para el tratamiento de excepciones aritméticas.

Enunciado: clase CálculosNuméricos

Se requiere definir una clase denominada CálculosNúmericos que realice las siguientes operaciones:

- Calcular el logaritmo neperiano recibiendo un valor *double* como parámetro. Este método debe ser estático. Si el valor no es positivo se genera una excepción aritmética.
- Calcular la raíz cuadrada recibiendo un valor *double* como parámetro. Este método debe ser estático. Si el valor no es positivo se genera una excepción aritmética.

Se debe crear un método *main* que utilice dichos métodos ingresando un valor por teclado.

Instrucciones Java del ejercicio

Tabla 6.6. Instrucciones Java del ejercicio 6.6.

Instrucción	Descripción	Formato
Math.log	Retorna el logaritmo de un valor pasado como parámetro.	*Math.log(double a);*
ArithmeticException	Esta excepción se lanza cuando ha ocurrido una condición aritmética excepcional. Por ejemplo, división por cero.	*ArithmeticException e;*
InputMismatchException	Excepción lanzada por un *Scanner* para indicar que el valor recuperado no coincide con el tipo esperado o que el valor está fuera del rango.	*InputMismatchInput e;*

Solución

Clase: CálculosNuméricos

```
package Excepciones;
import java.util.*;

/**
 * Esta clase denominada CálculosNuméricos permite realizar dos
 * cálculos matemáticos: calcular el logaritmo neperiano de un valor y
 * calcular la raíz cuadrada de un valor.
 * @version 1.2/2020
 */
public class CálculosNuméricos {

    /**
    * Método que calcula el logaritmo neperiano de un valor numérico.
    * Si el valor pasado como parámetro no es correcto se genera la
    * excepción correspondiente
    * @param valor Parámetro que define el valor al cual se le calculará
    * el logaritmo neperiano
    * @throws ArithmeticException Excepción aritmética que indica
    * que el número debe ser positivo
    * @throws InputMismatchException Excepción que indica que el
    * valor ingresado debe ser numérico
    */
```

```
static void calcularLogaritmoNeperiano(double valor) {
    try {
        if (valor < 0) {
            // Si el valor es negativo, se genera una excepción aritmética
            throw new ArithmeticException("El valor debe ser un
                número positivo");
        }
        // Si el valor es positivo, se calcula el algoritmo neperiano
        double resultado = Math.log(valor);
        System.out.println("Resultado = " + resultado);
    } catch (ArithmeticException e) {
        /* Si ocurre una excepción aritmética, se muestra un mensaje
            de error */
        System.out.println("El valor debe ser un número positivo para
            calcular el logaritmo");
    } catch (InputMismatchException e) {
        /* Si el valor a calcular no es numérico, se muestra un mensaje
            de error */
        System.out.println("El valor debe ser numérico para calcular
            el logaritmo");
    }
}

/**
* Método que calcula la raíz cuadrada de un valor numérico. Si el
* valor pasado como parámetro no es correcto se genera la
* excepción correspondiente
* @param valor Parámetro que define el valor al cual se le calculará
* la raíz cuadrada
* @throws ArithmeticException Excepción aritmética que indica
* que el número debe ser positivo
* @throws InputMismatchException Excepción que indica que el
* valor ingresado debe ser numérico
*/
static void calcularRaízCuadrada(double valor) {
    try {
        if (valor < 0) {
            // Si el valor es negativo, se genera una excepción aritmética
            throw new ArithmeticException("El valor debe ser un
                número positivo");
        }
        double resultado = Math.sqrt(valor); /* Si el valor es positivo,
            se calcula la raíz cuadrada */
```

```
                System.out.println("Resultado = " + resultado);
            } catch (ArithmeticException e) {
                /* Si ocurre una excepción aritmética, se muestra un mensaje
                   de error */
                System.out.println("El valor debe ser un número positivo para
                   calcular la raíz cuadrada");
            } catch (InputMismatchException e) {
                /* Si el valor a calcular no es numérico, se muestra un mensaje
                   de error */
                System.out.println("El valor debe ser numérico para calcular
                   la raíz cuadrada");
            }
        }

        /**
        * Método main que solicita un valor numérico por teclado. Para
        * dicho valor se calcula el logaritmo neperiano y la raíz cuadrada. Si
        * el valor ingresado no es correcto, se generan las excepciones
        * correspondientes.
        */
        public static void main(String args[]) {
            System.out.print("Valor numérico = ");
            Scanner teclado = new Scanner(System.in);
            double valor = teclado.nextDouble();
            calcularLogaritmoNeperiano(valor);
            calcularRaízCuadrada(valor);
        }
    }
```

Diagrama de clases

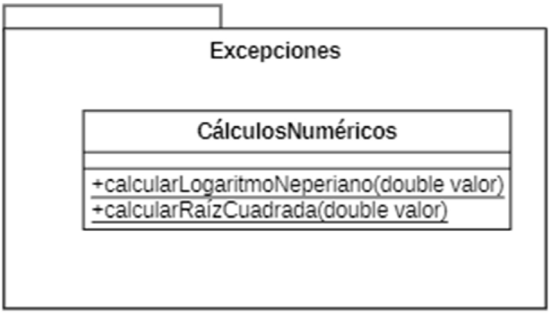


Figura 6.15. Diagrama de clases del ejercicio 6.6.

Explicación del diagrama de clases

Se ha definido un paquete denominado "Excepciones" con una única clase: CálculosNuméricos. Esta clase no posee atributos. La clase contiene dos métodos públicos estáticos para calcular el logaritmo neperiano y la raíz cuadrada de un valor *double* pasado como parámetro.

El código de estos dos métodos puede generar excepciones si el valor pasado como parámetro no es correcto.

Diagrama de máquinas de estado

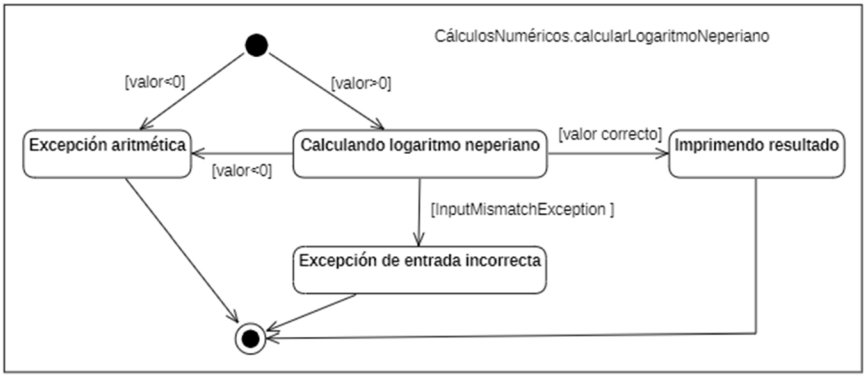


Figura 6.16. Diagrama de máquinas de estado del ejercicio 6.6. (logaritmo neperiano)

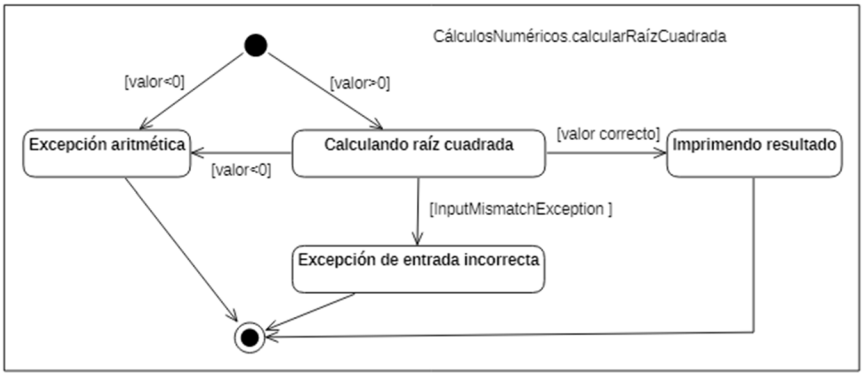


Figura 6.17. Diagrama de máquinas de estado del ejercicio 6.6. (raíz cuadrada)

Explicación del diagrama de máquinas de estado

Se ha agregado este tipo de diagrama UML porque el diagrama de clases no expresa la lógica interna de los métodos: calcular logaritmo neperiano y calcular raíz cuadrada, donde se pueden lanzar y gestionar posibles excepciones que ocurren durante la ejecución del programa.

En el primer diagrama de máquinas de estado se evalúa el valor pasado como parámetro, si el valor es menor que cero se pasa al estado "Excepción aritmética" y el programa finaliza. Si el valor pasado como parámetro es mayor que cero se pasa al estado "Calculando algoritmo neperiano". Si el valor calculado es correcto se pasa al estado "Imprimiendo resultado". Si al calcular el algoritmo neperiano se detecta un valor no numérico se pasa al estado "Excepción de entrada incorrecta" y se termina el programa.

El segundo diagrama de estados tiene un comportamiento muy similar al primero. Se evalúa el valor pasado como parámetro, si el valor es menor que cero se pasa al estado "Excepción aritmética" y el programa finaliza. Si el valor pasado como parámetro es mayor que cero se pasa al estado "Calculando raíz cuadrada". Si el valor calculado es correcto se pasa al estado "Imprimiendo resultado". Si al calcular la raíz cuadrada se detecta un valor no numérico se pasa al estado "Excepción de entrada incorrecta" y se termina el programa.

Ejecución del programa

```
Valor numérico = -200
El valor debe ser un número positivo para calcular el logaritmo
El valor debe ser un número positivo para calcular la raíz cuadrada
```

Figura 6.18. Ejecución del programa del ejercicio 6.6.

Ejercicios propuestos

- Agregar al ejercicio anterior los métodos que realicen las siguientes operaciones matemáticas:
  - Calcular la pendiente de una recta.
  - Calcular el punto medio de una recta.
  - Calcular las raíces de una ecuación cuadrática.
  - Convertir un número en base 10 a un número en base *b*.

Y agregar los manejadores de excepciones correspondientes.

Ejercicio 6.7. Validación de campos

Las excepciones se pueden utilizar para validar el formato de campos de entrada. Por ejemplo:

- Restringir que un campo de entrada de datos sea numérico, alfabético, alfanumérico, etc.
- Restringir la cantidad de caracteres que debe tener un campo de entrada de datos.
- Impedir que se ingresen fechas anteriores a la fecha actual.
- Definir campos obligatorios en un formulario de tal manera que no reciba valores vacíos.
- Definir restricciones especiales de los campos de entrada. Por ejemplo, si es un correo electrónico, que tenga el carácter @.

Para estos casos, si se presentan situaciones que no cumplen dichas condiciones se debe generar la excepción correspondiente.

Objetivo de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para definir métodos que realicen validación de campos y generen las excepciones apropiadas en caso de que no se cumplan las condiciones estipuladas en los requisitos de un programa.

Enunciado: clase EquipoMaratónProgramación

Un equipo de programadores desea participar en una maratón de programación. El equipo tiene los siguientes atributos:

- Nombre del equipo (tipo *String*).
- Universidad que está representando el equipo (tipo *String*).
- Lenguaje de programación que va a utilizar el equipo en la competencia (tipo *String*).
- Tamaño del equipo (tipo *int*).

Se requiere un constructor que inicialice los atributos del equipo. El equipo está conformado por varios programadores, mínimo dos y máximo

tres. Cada programador posee nombre y apellidos (de tipo *String*). Se requieren además los siguientes métodos:

- Un método para determinar si el equipo está completo.
- Un método para añadir programadores al equipo. Si el equipo está lleno se debe imprimir la excepción correspondiente.
- Un método para validar los atributos nombre y apellidos de un programador para que reciban datos que sean solo texto. Si se reciben datos numéricos se debe generar la excepción correspondiente. Además, no se permiten que los campos *String* tengan una longitud igual o superior a 20 caracteres.
- En un método *main* se debe crear un equipo solicitando sus datos por teclado y se validan los nombres y apellidos de los programadores.

Instrucciones Java del ejercicio

Tabla 6.7. Instrucciones Java del ejercicio 6.7.

Clase	Método	Descripción
String	*char* charAt(index i)	Retorna un valor *char* en el índice especificado.
Character	*boolean* isDigit(*char*)	Determina si el carácter especificado es un dígito.

Instrucción	Descripción	Formato
throw	Palabra clave utilizada para lanzar una excepción dentro de un método.	*throw new Exception*()

Solución

Clase: EquipoMaratónProgramación

```
package Excepciones;
import java.util.*;

/**
 * Esta clase denominada EquipoMaratónProgramación modela un
 * equipo de programadores que participará en una maratón de
 * programación. Un equipo cuenta con un nombre, la universidad
 * a la que pertenece, el lenguaje de programación que utilizará en la
 * competición, el tamaño del equipo y un array de programadores.
 * @version 1.2/2020
 */
```

```
public class EquipoMaratónProgramación {
    /* Atributo que define el nombre el equipo de la maratón de
       programación */
    String nombreEquipo;
    /* Atributo que define el nombre de la universidad a la que
       pertenece el equipo de la maratón de programación */
    String universidad;
    /* Atributo que define el lenguaje de programación utilizado por el
       equipo de la maratón de programación */
    String lenguajeProgramación;
    /* Atributo que define los programadores que conforman el equipo
       de la maratón de programación */
    Programador[] programadores;
    /* Atributo que define el tamaño del equipo de la maratón de
       programación */
    int tamañoEquipo;

    /**
    * Constructor de la clase EquipoMaratónProgramación
    * @param nombreEquipo Parámetro que define el nombre del
    * equipo de la maratón de programación
    * @param universidad Parámetro que define la universidad a la que
    * pertenece el equipo de la maratón de programación
    * @param lenguajeProgramación Parámetro que define el lenguaje
    * de programación que utilizará el equipo de la maratón de
    * programación
    */
    EquipoMaratónProgramación(String nombreEquipo, String
       universidad, String lenguajeProgramación) {
        this.nombreEquipo = nombreEquipo;
        this.universidad = this.universidad;
        this.lenguajeProgramación = this.lenguajeProgramación;
        this.programadores = this.programadores;
        tamañoEquipo = 0; // El tamaño del equipo inicialmente es cero
        programadores = new Programador[3]; /* Crea un array con tres
            programadores */
    }

    /**
    * Método que determina si el array de programadores del equipo
    * está lleno o no
    * @return Valor boolean que determina si el array de programadores
    * está lleno o no
    */
```

```
boolean estáLleno() {
return tamañoEquipo == programadores.length;
}

/**
* Método que permite añadir un programador al array de
* programadores
* @param programador Parámetro que define el programador a
* agregar al array de programadores
* @throws Exception Excepción que indica que el equipo de
* programación está completo
*/
void añadir(Programador programador) throws Exception {
    if (estáLleno()) { /* Si el array está lleno, se genera la excepción
        correspondiente */
        throw new Exception ("El equipo está completo. No se pudo
            agregar programador.");
    }
    // Se asigna el programador al array de programadores
    programadores[tamañoEquipo] = programador;
    tamañoEquipo++; // Se incrementa el tamaño del equipo
}

/**
* Método que permite validar un campo evaluando si el campo no
* tiene dígitos y su longitud no debe ser superior a 20 caracteres. Si
* no cumple estos criterios, se generan las excepciones
* correspondientes
* @param campo Parámetro que define el campo a validar
* @throws Exception Excepción que indica que el nombre ingresado
* no puede tener dígitos o que la longitud no debe ser superior a 20
* caracteres
*/
static void validarCampo(String campo) throws Exception {
    for (int j = 0; j < campo.length(); j++) {
        char c = campo.charAt(j);
        if (Character.isDigit(c)) { /* Si el caracter es un dígito se genera
            la excepción correspondiente */
            throw new Exception("El nombre no puede tener dígitos.");
        }
    }
```

```
        /* Si la longitud del campo es mayor que 20 caracateres, se
           genera una excepción */
        if (campo.length() > 20) {
            throw new Exception("La longitud no debe ser superior a 20
                caracteres.");
        }
    }

    /**
    * Método main que solicita ingresar por teclado el nombre del
    * equipo, la universidad, el lenguaje de programación, crea un
    * equipo de maratón de programación y luego solicita ingresar por
    * teclado tres integrantes del equipo, realizando las validaciones de
    * datos.
    */
    public static void main(String args[]) throws Exception {
        Scanner teclado = new Scanner(System.in);
        System.out.println("Nombre del equipo = ");
        String nombre = teclado.next();
        System.out.println("Universidad = ");
        String universidad = teclado.next();
        System.out.println("Lenguaje de programación = ");
        String lenguaje = teclado.next();
        EquipoMaratónProgramación equipo = new
            EquipoMaratónProgramación(nombre, universidad,
            lenguaje);
        System.out.println("Datos de los integrantes del equipo");
        String nombreProgramador;
        String apellidosProgramador;
        for (int i= 0; i < 3; i++) {
            System.out.println("Nombre del integrante = ");
            nombreProgramador = teclado.next();
            validarCampo(nombreProgramador); /* Valida que el nombre
                del integrante sea correcto */
            System.out.println("Apellidos del integrante = ");
            apellidosProgramador = teclado.next();
            validarCampo(apellidosProgramador); /* Valida que el
                apellido sea correcto */
            Programador programador = new
                Programador(nombreProgramador,
```

```
                    apellidosProgramador); // Crea un programador
                equipo.añadir(programador); /* Añade el programador al
                    array de programadores */
            }
        }
    }
```

Clase: EquipoMaratónProgramación

```
package Excepciones;

/**
 * Esta clase denominada Programador modela un integrante de un
 * equipo de programadores que participará en una maratón de
 * programación. Un programador cuenta con un nombre y apellidos.
 * @version 1.2/2020
 */
public class Programador {
    String nombre; /* Atributo que identifica el nombre de un
        programador */
    String apellidos; /* Atributo que identifica los apellidos de un
        programador */

    /**
    * Constructor de la clase Programador
    * @param nombre Parámetro que define el nombre del programador
    * @param apellidos Parámetro que define los apellidos del
    * programador
    */
    Programador(String nombre, String apellidos) {
        this.nombre = nombre;
        this.apellidos = apellidos;
    }
}
```

Diagrama de clases

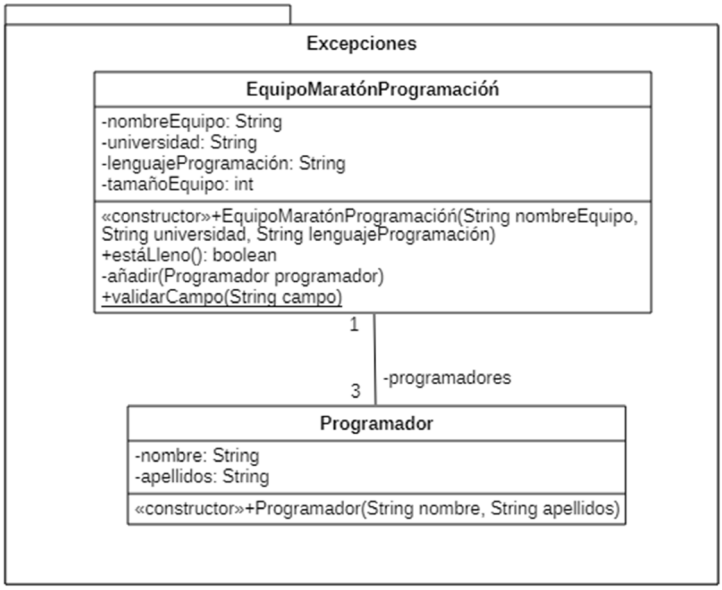


Figura 6.19. Diagrama de clases del ejercicio 6.7.

Explicación del diagrama de clases

Se ha definido un paquete denominado "Excepciones" con dos clases: EquipoMaratónProgramación y Programador.

La clase EquipoMaratónProgramación posee los atributos privados: nombre del equipo, universidad, lenguaje de programación y tamaño del equipo. Cuenta con un constructor y métodos para determinar si el equipo está lleno (máximo tres integrantes), añadir un programador al equipo y un método estático para validar los campos de ingreso de datos (los métodos estáticos se identifican porque tienen su texto subrayado).

La clase Programador tiene dos atributos privados: el nombre y los apellidos del programador y cuenta con su respectivo constructor.

Un equipo está conformado por programadores, esta relación se muestra en el diagrama con una línea continua entre las dos clases, donde un equipo se relaciona con muchos programadores y un programador con un solo equipo, de acuerdo con la multiplicidad mostrada.

Diagrama de objetos

Figura 6.20. Diagrama de objetos del ejercicio 6.7.

Ejecución del programa

```
Nombre del equipo =
EquipoUNAL
Universidad =
Nacional
Lenguaje de programación =
Java
Datos de los integrantes del equipo
Nombre del integrante =
Pedro
Apellidos del integrante =
Perez
Nombre del integrante =
Pablo3

Can only enter input while your programming is running

java.lang.Exception: El nombre no puede tener dígitos.
        at Excepciones.EquipoMaratónProgramación.validarCampo(EquipoMaratónProgramación.java:80)
        at Excepciones.EquipoMaratónProgramación.main(EquipoMaratónProgramación.java:109)
```

Figura 6.21. Ejecución del programa del ejercicio 6.7. con excepciones

Al ejecutar el programa se han ingresado los siguientes datos: "EquipoUNAL" como nombre del equipo, "Nacional" como nombre de la universidad del equipo, "Java" como lenguaje de programación. Luego, se ingresan los datos de los integrantes del equipo. El primer integrante tiene como nombre "Pedro", apellido: "Pérez", pero como nombre se ha ingresado un error: "Pablo3". Por ello, se ha generado una excepción con el texto descriptivo: "El nombre no puede tener dígitos".

En la siguiente ejecución del programa se ingresan datos que sí cumplen los requisitos del programa.

```
Nombre del equipo =
EquipoUNAL
Universidad =
Nacional
Lenguaje de programación =
Java
Datos de los integrantes del equipo
Nombre del integrante =
Jorge
Apellidos del integrante =
Ramírez
Nombre del integrante =
Pablo
Apellidos del integrante =
Perez
Nombre del integrante =
Jairo
Apellidos del integrante =
Méndez
```

Figura 6.22. Ejecución del programa del ejercicio 6.7. sin excepciones.

Ejercicios propuestos

- Se requiere desarrollar una contraseña válida. Los requisitos de la contraseña son los siguientes:
  - Mínimo 8 caracteres.
  - No debe tener espacios en blanco.
  - Debe tener por lo menos un carácter, un carácter en mayúscula, un número y un carácter especial.
  - La contraseña se debe ingresar dos veces para su confirmación.

Se lanzarán excepciones si no se cumplen dichos requerimientos y si las dos contraseñas no son iguales.

Ejercicio 6.8. Lectura de archivos

Los archivos en Java se pueden manejar como archivos de texto o archivos de flujos de *bytes*. La clase *FileInputStrem* obtiene *bytes* de entrada de un archivo en un sistema de archivos. Mientras que la clase *FileInputStream* puede ser utilizada para leer secuencias de caracteres (Samoylov, 2019).

Esta clase se utiliza junto la clase *InputStreamReader* que toma los *bytes* y los convierte a caracteres Unicode. Luego, estos se pasan a una clase *BufferedReader*, que almacena en un búfer los caracteres, para proporcionar una lectura eficiente de los caracteres, *arrays* y líneas. Luego, mediante el método *readLine()* de *BufferedReader* se realiza la lectura línea a línea del archivo (Joy *et al.*, 2013). El formato de las instrucciones combinadas es el siguiente:

```
InputStreamReader entrada = new InputStreamReader(System.in);
BufferedReader teclado = new BufferedReader (entrada);
String cadena = teclado.readLine();
```

Para utilizar las clases *FileInputStream*, *InputStreamReader* y *BufferedReader* se debe importar el paquete: *java.io*.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Crear un flujo de *bytes* para leer archivos de texto.
- Conocer y aplicar las clases *InputStreamReader* y *BufferedReader* para la creación del flujo de *bytes* que facilita la lectura de archivos.

Enunciado: clase LeerArchivo

Se tiene un archivo de texto denominado prueba.txt en una cierta localización en un sistema de archivos. Se requiere desarrollar un programa que lea dicho archivo de texto utilizando un flujo de *bytes* que muestre los contenidos del archivo en pantalla.

Instrucciones Java del ejercicio

Tabla 6.8. Instrucciones Java del ejercicio 6.8.

Clase	Método	Descripción
BufferedReader	*String readLine()*	Lee una línea de texto de un archivo.
	void close()	Cierra un flujo de datos abierto previamente.
IOException		Indica que se ha producido una excepción de E/S de algún tipo.

Solución

Clase: LeerArchivo

```
package Archivos;
import java.io.*;

/**
 * Esta clase denominada LeerArchivo tiene como objetivo leer los datos
 * presentes en un archivo de texto con extensión .txt
 * @version 1.2/2020
 */
public class LeerArchivo {

    /**
    * Método main que lee una archivo de texto y muestra su contenido
    * en pantalla
    * @throws IOException Excepción que indica que no se pudo leer
    * el archivo
    */
    public static void main(String[] args) throws Exception {
        String nombreArchivo = "C:/prueba.txt"; /* Definición del
            archivo de texto a leer */
        FileInputStream archivo; // Definición de flujo de datos
        InputStreamReader conversor; // Definición del flujo de lectura
        BufferedReader filtro; // Definición del buffer
        String línea;
        try {
            /* Crea los objetos FileInputStream, BufferedReader y
                BufferedReader */
            archivo = new FileInputStream(nombreArchivo);
            conversor = new InputStreamReader(archivo);
            filtro = new BufferedReader(conversor);
            línea = filtro.readLine();
            while (línea != null) {
                System.out.println(línea); /* Imprime en pantalla una
                    línea del archivo */
                línea = filtro.readLine(); // Lee la siguiente línea
            }
            filtro.close(); // Cierra el archivo
        } catch (IOException e) { // En caso que se genere una excepción
```

```
                System.out.println("No se pudo leer el archivo.");
            }
        }
    }
```

Diagrama de clases

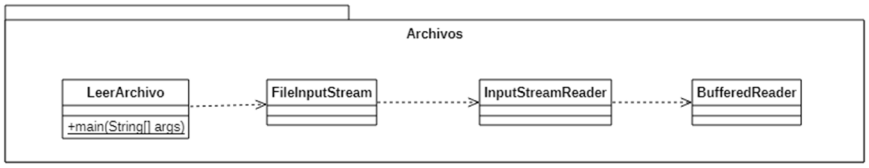


Figura 6.23. Diagrama de clases del ejercicio 6.8.

Explicación del diagrama de clases

Se ha definido un paquete denominado "Archivos" con clases que permiten la lectura de un archivo de texto. La clase principal y la única que se define se denomina LeerArchivo. En el método *main* de dicha clase se define y utiliza un objeto *FileInputStream*, el cual debe utilizar un objeto *InputStreamReader*, que, a su vez, usa un objeto *BufferedReader* para hacer la lectura final del archivo de texto.

Las relaciones entre estas clases se especifican por medio de relaciones de dependencia, las cuales se modelan en UML por medio de una línea entrecortada con punta en flecha. La relación se lee como "una clase A depende de la clase B", el sentido de la flecha va de A hacia B. Una relación de dependencia indica que una clase depende de otra, es decir, si una clase cambia, la otra también cambia. Por lo tanto, en el diagrama de clases se presenta una cadena de dependencias para hacer la lectura de un archivo de texto.

Diagrama de objetos

Figura 6.24. Diagrama de objetos del ejercicio 6.8.

Ejecución del programa

```
Linea 1
Linea 2
Linea 3
Linea 4
Linea 5
Linea 6
Linea 7
Linea 8
Linea 9
Linea 10
```

Figura 6.25. Ejecución del programa del ejercicio 6.8.

Ejercicios propuestos

- Escribir un programa que lea por teclado el nombre de un archivo de texto y muestre su contenido en pantalla.
- Desarrollar un método que lea el contenido del archivo y lo muestre en pantalla con todos los caracteres en minúsculas convertidos a mayúsculas.

Ejercicio 6.9. Escritura de archivos

Para escribir archivos de texto en Java se recomienda trabajar con la clase *FileWriter*, que permite crear archivos de texto. Luego, se utiliza el método *write()* de *FileWriter* para escribir el texto en el archivo. Después, el archivo se puede cerrar con el método *close()* (Horstmann, 2018).

Por otro lado, para escribir texto formateado a un archivo se recomienda utilizar la clase *PrintWriter*. Esta clase implementa todos los métodos de impresión que se encuentran en *PrintStream*, por lo que puede utilizar todos los formatos de las declaraciones *System.out.println()*. El formato combinando *FileWriter* y *PrintWriter* es el siguiente:

```
try {
        FileWriter archivo = new FileWriter(String rutaArchivo);
        PrintWriter impresor = new PrintWriter(archivo);
        impresor.println(String contenido);
}
```

Para crear objetos de estas clases, estos deben estar dentro de un *try-catch*. Y para utilizar la clase *FileWriter* y *PrintWriter* se debe importar el paquete: *java.io*.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Crear flujos de *bytes* para escribir en archivos de texto.
- Utilizar las clases *FileWriter* y *PrintWriter* junto con sus métodos para escribir archivos de texto.

Enunciado: clase EscribirArchivo

Se requiere crear un archivo de texto denominado "prueba.txt" localizado en un sistema de archivos. Además, se requiere escribir en cada línea de dicho archivo, las cadenas "Línea *X*", donde *X* representa la línea del archivo.

Instrucciones Java del ejercicio

Tabla 6.9. Instrucciones Java del ejercicio 6.9.

Clase	Método	Descripción
Exception	*void printStackTrace()*	Imprime información sobre la excepción y una traza hasta la secuencia de error estándar.
FileWriter	void close()	Cierra un flujo de datos abierto.

Solución

Clase: EscribirArchivo

```
package Archivos;
import java.io.*;
/**
 * Esta clase denominada EscribirArchivo tiene como objetivo escribir
 * datos en un archivo de texto.
 * @version 1.2/2020
 */
public class EscribirArchivo {
```

```
    /**
    * Método main que escribe datos en un archivo de texto
    * @throws Exception Excepción que indica que no se pudo
    * imprimir datos en el archivo
    */
    public static void main(String[] args) {
        FileWriter archivo = null;
        PrintWriter impresor = null;
        try {
            archivo = new FileWriter("f:/prueba.txt"); /* Definición del
                archivo a crear */
            impresor = new PrintWriter(archivo); /* Definición del objeto
                que imprime */
            for (int i = 0; i < 10; i++) // Ciclo para imprimir 10 líneas
                impresor.println("Linea " + i);
        } catch (Exception e) {
            /* Captura una excepción en caso de no poder imprimir datos
                en el archivo de texto */
            e.printStackTrace(); /* Muestra un texto descriptivo acerca de
                la excepción generada */
        } finally {
            try {
                if (archivo != null)
                    archivo.close(); // Cierra el archivo
            } catch (Exception e2) { /* Captura una excepción en caso de
                que ocurra */
                e2.printStackTrace(); /* Muestra un texto descriptivo
                acerca de la excepción */
            }
        }
    }
}
```

Diagrama de clases

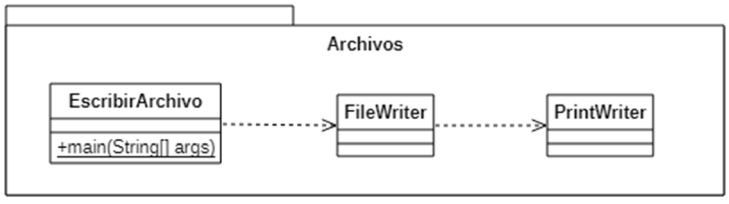


Figura 6.26. Diagrama de clases del ejercicio 6.9.

Explicación del diagrama de clases

Se ha definido un paquete denominado "Archivos" con clases que permiten la escritura de un archivo de texto. La clase principal y la única que se define se denomina EscribirArchivo. El método *main* de esta clase define y utiliza un objeto *FileWriter*, que debe utilizar un objeto *PrinterWriter* para hacer la escritura final en el archivo de texto.

Las relaciones entre estas clases se especifican por medio de relaciones de dependencia, las cuales se modelan en UML por medio de una línea entrecortada con punta en flecha. La relación se lee como "una clase A depende de la clase B", el sentido de la flecha va de A hacia B. Una relación de dependencia indica que una clase depende de otra, es decir, si una clase cambia la otra también cambia. Por lo tanto, en el diagrama de clases se presenta una cadena de dependencias para hacer la escritura en un archivo de texto.

Diagrama de objetos

Figura 6.27. Diagrama de objetos del ejercicio 6.9.

Ejecución del programa

```
Linea 0
Linea 1
Linea 2
Linea 3
Linea 4
Linea 5
Linea 6
Linea 7
Linea 8
Linea 9
```

Figura 6.28. Ejecución del programa del ejercicio 6.9.

Ejercicios propuestos

- Desarrollar una clase que imprima una factura con artículos comprados. Los atributos de la clase son *arrrays* de: precios, unidades y nombres de productos.
  Un ejemplo de impresión podría ser:

Precio	Unidades	Producto
30000	12	Camiseta
5000	10	Cuaderno
10000	12	Lapicero

Ejercicio 6.10. Lectura de directorios

La clase *File* proporciona información acerca de archivos, sus atributos, directorios, etc. (API Java, 2020). Con el constructor se identifica la ruta donde se encuentra un directorio o archivo:

File archivo = new File(String path);

Los métodos más importantes que describen esta clase son (API Java, 2020):

- ***getName()***: obtiene el nombre del archivo o directorio.
- ***getPath()***: obtiene la ruta del archivo o directorio.
- ***exists()***: obtiene un valor booleano determinando si el archivo o directorio existe.
- ***canWrite()***: obtiene un valor booleano determinando si el archivo o directorio se puede escribir.
- ***canRead()***: obtiene un valor booleano determinando si el archivo o directorio se puede leer.
- ***isFile()***: obtiene un valor booleano determinando si es un archivo.
- ***isDirectory()***: obtiene un valor booleano determinando si es un directorio.
- ***list()***: lista el directorio actual.

Para utilizar la clase *File* se debe importar el paquete: *java.io*.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Leer los contenidos de un directorio determinado desde el sistema de archivos.
- Conocer y aplicar diferentes métodos para consultar información sobre directorios y archivos.

Enunciado: clase LeerDirectorio

Se requiere desarrollar un programa que liste el contenido de un determinado directorio de un sistema de archivos.

Solución

Clase: LeerDirectorio

```
package Archivos;
import java.io.File;
import java.util.Arrays;

/**
 * Esta clase denominada LeerDirectorio tiene como objetivo leer y
 * mostrar en pantalla los contenidos de un directorio específico.
 * @version 1.2/2020
 */
public class LeerDirectorio {

    /**
     * Método main donde se define un directorio y se define un array de
     * Strings donde se guardarán los contenidos del directorio
     */
    public static void main(String args[]) {
        File directorio = new File("f://"); // Definición del directorio a leer
        /* Definición del array de Strings que almacena los contenidos del
           directorio */
        String[] archivos = directorio.list();
        for (int i = 0; i < archivos.length; i++) { /* Ciclo que muestra en
           pantalla contenidos del array */
```

```
                System.out.println( "Archivo: " + archivos[i] );
            }
        }
    }
```

Diagrama de clases

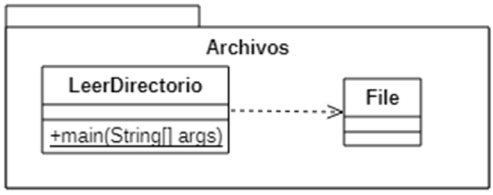


Figura 6.29. Diagrama de clases del ejercicio 6.10.

Explicación del diagrama de clases

Se ha definido un paquete denominado "Archivos" con clases que permiten la lectura del contenido de un determinado directorio en el sistema de archivos de un computador. La clase principal y la única que se define en el código del programa se denomina LeerDirectorio. En el método *main* de esta clase se define y utiliza un objeto *File*, que se encarga de leer el contenido del directorio especificado.

La relación entre estas dos clases especifica una relación de dependencia, la cual se modela en UML por medio de una línea entrecortada con punta en flecha. La relación se lee como "la clase LeerDirectorio depende de la clase *File*", el sentido de la flecha indica el sentido de la dependencia. Una relación de dependencia indica que una clase depende de otra, es decir, si una clase cambia la otra también.

Diagrama de objetos

Figura 6.30. Diagrama de objetos del ejercicio 6.10.

Ejecución del programa

```
Archivo: System Volume Information
Archivo: Proyectos
Archivo: Libro1
Archivo: Libro2
Archivo: Introducción
Archivo: Escritorio
Archivo: Lecturas
Archivo: Otros
```

Figura 6.31. Ejecución del programa del ejercicio 6.10.

Ejercicios propuestos

- Escribir un programa para obtener los nombres de los archivos específicos desde un directorio determinado, previamente, se debe dar la extensión del archivo.
- Escribir un programa para verificar si un archivo o directorio existe dando previamente una ruta especificada.
- Escribir un programa para encontrar la palabra más larga en un archivo de texto.

Ejercicio 6.11. Lectura/escritura de objetos

Java proporciona la serialización de objetos, un mecanismo en el que un objeto se representa como una secuencia de *bytes* que incluye el tipo del objeto, los tipos de datos del objeto y los datos almacenados en el objeto (Schildt, 2018).

Un objeto serializado permite que sus datos se escriban en un archivo y que sus datos se puedan leer desde el archivo, además, permite que la información de tipo y los *bytes* que representan el objeto y sus datos se puedan usar para recrear el objeto en la memoria.

Para serializar y deserializar objetos se requiere, en primer lugar, objetos *FileInputStream* y *FileOutputStream* para referenciar los archivos,

respectivamente. Luego, las clases *ObjectInputStream* y *ObjectOutputStream* contienen los métodos para serializar y deserializar objetos (Clark, 2017).

- *Clase ObjectOutputStream*: un *ObjectOutputStream* escribe tipos de datos primitivos y de objetos en un *OutputStream*.
  - El método *writeObject* se utiliza para escribir un objeto en el flujo. Cualquier objeto, incluidas *String* y *arrays*, se escribe con *writeObject*. Se pueden escribir múltiples objetos o tipos primitivos de datos en el flujo. Los objetos deben leerse desde el *ObjectInputstream* correspondiente con los mismos tipos y en el mismo orden en que fueron escritos.

- *Clase ObjectInputStream*: un *ObjectInputStream* deserializa tipos de datos primitivos y objetos escritos previamente utilizando un *ObjectOutputStream*.
  - El método *readObject* se utiliza para leer un objeto de la secuencia. Se debe utilizar el *casting* para obtener el tipo deseado. En Java, los *Strings* y *arrays* son objetos y se tratan como objetos durante la serialización. Cuando se lean, se deben convertir al tipo esperado. El mecanismo de deserialización predeterminado para objetos restaura el contenido de cada campo al valor y tipo que tenían cuando se escribió.

Para utilizar la clase *File* se debe importar el paquete: *java.io*.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Serializar objetos, es decir, crear archivos que contengan información almacenada sobre objetos.
- Deserializar objetos, es decir, extraer información sobre los objetos almacenados en archivos.

Enunciado: clase Asignatura

Se requiere un programa que almacene y recupere información almacenada en un archivo sobre una asignatura universitaria.

La información de la asignatura comprende:

- Código de la asignatura (tipo *int*).

- Nombre de la asignatura (tipo *String*).
- Cantidad de créditos de la asignatura (tipo *int*).

La clase asignatura debe tener un constructor que inicialice sus atributos y los siguientes métodos:

- imprimir: muestra en pantalla los valores de los atributos de una asignatura.
- leerAsignatura: lee los contenidos de un archivo que tiene almacenada la información de una asignatura y los muestra en pantalla.
- escribirAsignatura: escribe los contenidos de un objeto asignatura en un archivo.

Se debe desarrollar una clase de prueba que cree una asignatura, la almacene en un archivo y, luego, la recupere y muestre su información en pantalla.

Instrucciones Java del ejercicio

Tabla 6.10. Instrucciones Java del ejercicio 6.10.

Clase	Descripción/Método	Formato
Serializable	La serialización de una clase está habilitada por la clase que implementa la interfaz *java.io.Serializable*.	*Class Clase implements Serializable*
FileNotFoundException	Indica que ha fallado un intento de abrir el archivo indicado por un nombre de ruta especificado.	*catch (FileNotFoundException e)*
IOException	Indica que se ha producido una excepción de E/S de algún tipo.	*catch (IOException e) {…}*
ObjectInputStream	*void close()*: cierra un flujo de entrada creado previamente.	*ObjectInputStream.close();*
ObjectOutputStream	*void close()*: cierra un flujo de salida creado previamente.	*ObjectOutputStream.close();*

Solución

Clase: Asignatura

```
package Archivos;
import java.io.*;

/**
 * Esta clase denominada Asignatura modela una determinada
 * asignatura que tiene un nombre, código y cantidad de créditos. La
 * clase implementa la interfaz Serializable que permite guardar objetos
 * en un archivo de datos.
 * @version 1.2/2020
 */
public class Asignatura implements Serializable {
    int código; // Atributo que identifica el código de la asignatura
    String nombre; // Atributo que identifica el nombre de la asignatura
    int créditos; /* Atributo que identifica la cantidad de créditos que
        tiene la asignatura */

    /**
    * Constructor de la clase Asignatura
    * @param código Parámetro que define el código de la asignatura
    * @param nombre Parámetro que define el nombre de la asignatura
    * @param créditos Parámetro que define la cantidad de créditos de
    * la asignatura
    */
    Asignatura(int código, String nombre, int créditos) {
        this.código = código;
        this.nombre = nombre;
        this.créditos = créditos;
    }

    /**
    * Método que muestra en pantalla los datos de una asignatura
    */
    void imprimir() {
        System.out.println("Código de la asignatura = " + código);
        System.out.println("Nombre de la asignatura = " + nombre);
        System.out.println("Cantidad de créditos = " + créditos);
    }
```

```
/**
* Método que crea un archivo y guarda los datos de un objeto
* Asignatura
* @throws IOException Excepción que indica que no se pudo
* escribir en el archivo
*/
void escribirAsignatura() {
    try {
        // Define el archivo donde se guardarán los datos del objeto
        FileOutputStream archivo = new
            FileOutputStream("Asignatura.dat");
        // Crea el objeto de flujo de salida para la escritura del objeto
        ObjectOutputStream salida = new
            ObjectOutputStream(archivo);
        salida.writeObject(this); // Escribe el objeto en el flujo de salida
        salida.close(); // Cierra el flujo de salida de datos
    } catch (IOException e) {
        /* Captura una excepción en caso de ocurrir un error en la
            escritura de datos del archivo */
        System.out.println("No se puedo escribir en el archivo");
    }
}

/**
* Método que leer un archivo de datos, los asigna a los atributos de
* un objeto Asignatura y muestra su contenido en pantalla
* @throws FileNotFoundException Excepción que indica que no se
* pudo leer el archivo
* @throws IOException Excepción que indica que se presentó un
* error de entrada/salida
* @throws Exception Excepción general
*/
void leerAsignatura() {
    try {
        FileInputStream archivo = new FileInputStream("Asignatura.
            dat"); // Define el archivo a leer
        // Crea el objeto de flujo de entrada para la lectura del objeto
        ObjectInputStream entrada = new
            ObjectInputStream(archivo);
        // Lee el objeto en el flujo de entrada
        Asignatura asignatura = (Asignatura) entrada.readObject();
        asignatura.imprimir(); /* Muestra en pantalla los datos del
            objeto leído */
```

```
            entrada.close(); // Cierra el flujo de entrada de datos
        } catch (FileNotFoundException e) {
            // Captura una excepción en caso de no encontrar el archivo
            System.out.println("No se pudo leer el archivo");
        } catch (IOException e) {
            /* Captura una excepción en caso de ocurrir un error de
               entrada/salida */
            System.out.println("Error de entrada/salida");
        } catch (Exception e) { // Captura una excepción general
            System.out.println("Error al leer el archivo");
        }
    }
}
```

Clase: Prueba

```
package Archivos;

/**
 * Esta clase prueba la creación de un objeto Asignatura y lo guarda en
 * un archivo de datos. Luego lo lee y muestra su contenido en pantalla.
 * @version 1.2/2020
 */
public class Prueba {

    /**
     * Método main que prueba la creación de una asignatura, guarda el
     * objeto creado en un archivo de datos y luego, lee el archivo y
     * muestra su contenido en pantalla
     */
    public static void main(String args[]) {
        Asignatura asignatura = new Asignatura(4100547,"Programación
            Orientada a Objetos", 4);
        asignatura.escribirAsignatura();
        asignatura.leerAsignatura();
    }
}
```

Diagrama de clases

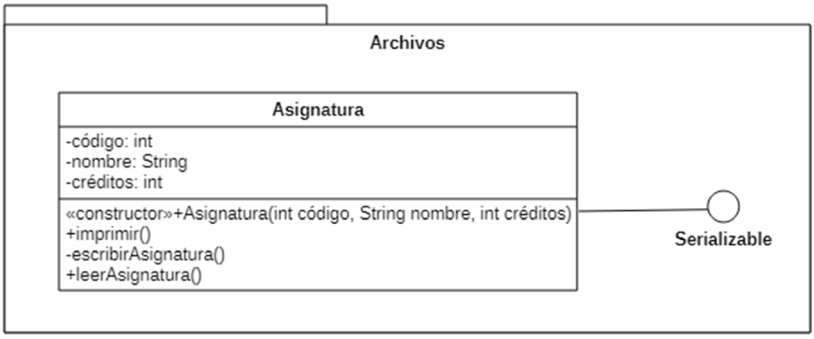


Figura 6.32. Diagrama de clases del ejercicio 6.11.

Explicación del diagrama de clases

Se ha definido un paquete denominado "Archivos" con una única clase que modela una Asignatura con los atributos privados: código, nombre y créditos. La clase posee un constructor y métodos para imprimir datos en pantalla, escribir un objeto asignatura en un archivo de datos y leer un objeto asignatura de un archivo de datos.

Para que un objeto de tipo asignatura puede escribirse y leerse de un archivo de datos, se debe implementar la interfaz Serializable, la cual se indica en UML con la notación gráfica de un círculo que se conecta mediante una línea continua con la clase que lo implementa.

Diagrama de objetos

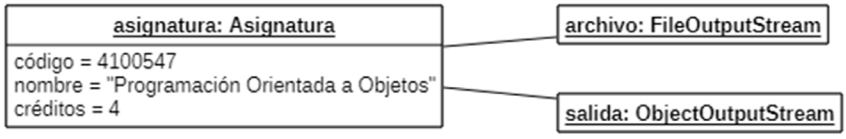


Figura 6.33. Diagrama de objetos del ejercicio 6.11.

Ejecución del programa

```
Código de la asignatura = 4100547
Nombre de la asignatura = Programación Orientada a Objetos
Cantidad de créditos = 4
```

Figura 6.34. Ejecución del programa del ejercicio 6.11.

Ejercicios propuestos

- Una inmobiliaria posee un listado de inmuebles para vender y arrendar. Cada inmueble tiene como atributos: dirección, ciudad, tipo (de arriendo o venta). Se requiere que un programa puede agregar, editar y eliminar inmuebles. Además, el listado de inmuebles junto con sus datos se debe guardar en un archivo binario.
- Una persona que tiene una cuenta de ahorros bancaria puede realizar consignaciones y retiros de su cuenta. Se requiere un programa que genere un extracto bancario de las consignaciones, retiros y saldos obtenidos en cada transacción y se almacene en un archivo binario.
- Se desea guardar en un archivo binario las 10 notas finales de las asignaturas cursadas por un estudiante. Para ello, se requiere desarrollar un programa que capture por teclado el nombre de las asignaturas y sus correspondientes notas finales, las cuales se guardarán en un archivo. Luego, el programa debe leer el archivo y calcular el promedio final del estudiante.

Capítulo 7
Clases útiles

El propósito general del séptimo capítulo es presentar un conjunto adicional de clases que pueden ser útiles en la implementación de programas. Varios sistemas de *software* trabajan con fechas, realizan procesamiento de datos tipo *String*, dan formato a la información de acuerdo con su tipo y trabajan con elementos aleatorios. En este capítulo se presentan cuatro ejercicios sobre algunas clases útiles para hacer programas utilizando Java.

- **Ejercicio 7.1. Clase *LocalDate***

  Durante el desarrollo de programas es muy común el uso, manejo y procesamiento de fechas. Java posee la clase *LocalDate* para gestionar acciones que requieren trabajar con este tipo de información. De esta manera, se cuenta con una clase que posee numerosos métodos para manipular información sobre fechas. El primer ejercicio propone un problema que utiliza esta clase.

- **Ejercicio 7.2. Clase *StringTokenizer***

  La definición y procesamiento de datos de tipo *String* son muy numerosos durante el desarrollo de programas. Generalmente, se requiere extraer porciones específicas de información de las cadenas de caracteres que conforman un *String*. Estas porciones de texto se denominan *tokens* y Java cuenta con la clase especializada denominada *StringTokenizer* para implementar dicho concepto.

El segundo ejercicio permite utilizar e implementar esta clase para extraer *tokens* específicos.

- **Ejercicio 7.3. Clase *NumberFormat***
  Los datos de un objeto poseen un tipo de dato predefinido por Java, pero cuando se muestran al usuario final, estos tienen asociado un formato de presentación específico dependiendo del contexto. Por ejemplo, si es dinero debe ir en formato moneda con el símbolo $, separadores de miles y cifras decimales. Java posee una clase especializada denominada *NumberFormat* para dar formato a posibles valores numéricos. El tercer ejercicio propone un problema que será abordado utilizando esta clase.

- **Ejercicio 7.4. Generación de números aleatorios**
  Algunos programas, durante su ejecución, requieren la generación de números aleatorios que modelen sucesos o eventos obtenidos al azar, es decir, que los números tengan la misma probabilidad de ser elegidos y que la elección de uno no dependa de la elección de otro generado anteriormente. El último ejercicio de este capítulo presenta un problema que incluye la generación de números aleatorios en su solución.

Los diagramas UML utilizados en este capítulo son los diagramas de clase y de objetos. Los paquetes permiten agrupar elementos UML y en los ejercicios se organizarán las clases en un único paquete debido a la pequeña cantidad de clases incluidas en las soluciones presentadas. En primer lugar, los diagramas de clase modelan la estructura estática de los programas. Así, los diagramas de clase de los ejercicios presentados, generalmente, estarán conformados por una o varias clases relacionadas e incorporan notaciones para identificar tipos de atributos y métodos. En segundo lugar, los diagramas de objeto identifican los objetos creados en el método *main* del programa.

Ejercicio 7.1. Clase *LocalDate*

LocalDate es un objeto inmutable que representa una fecha, a menudo vista en el formato año-mes-día. También se puede acceder a otros campos de fecha como día del año, día de la semana y semana del año (Joy *et al.*, 2013).

La clase *LocalDate* presenta principalmente los siguientes métodos (API Java, 2020):

- ***compareTo(ChronoLocalDate otraFecha)*:** compara la fecha actual con otra.
- ***getDayOfMonth()*:** obtiene el día del mes.
- ***getDayOfWeek()*:** obtiene el día de la semana, el cual es un dato enumerado *DayOfWeek*.
- ***getDayOfYear()*:** obtiene el día del año.
- ***getMonth()*:** obtiene el mes del año utilizando el dato enumerado *Month*.
- ***getMonthValue()*:** obtiene el mes del año del 1 al 12.
- ***getYear()*:**obtiene el año.
- ***now()*:** obtiene la fecha actual desde el reloj del sistema y la zona horaria por defecto.

Para utilizar la clase *LocalDate* se debe importar el paquete: *java.time*.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Crear objetos de tipo *LocalDate* para el manejo de fechas.
- Utilizar diferentes métodos de la clase *LocalDate*.

Enunciado: clase Persona

Una persona posee un nombre (tipo *String*) y una fecha de nacimiento (objeto tipo *LocalDate*). Se requiere un programa que solicite por teclado el nombre, el día, el mes y el año de nacimiento. El programa debe determinar la edad en años de la persona y mostrar su resultado en pantalla.

Instrucciones Java del ejercicio

Tabla 7.1. Instrucciones Java del ejercicio 7.1.

Clase	Método	Descripción
Period	*between(LocalDate fechaInicio, LocalDate fechaFin)*	Obtiene un periodo que consiste en la cantidad de años, meses y días entre dos fechas.
LocalDate	*LocalDate of(int año, int mes, int día)*	Obtiene una instancia de *LocalDate* a partir de un año, mes y día.
	LocalDate now()	Obtiene la fecha actual del reloj del sistema en la zona horaria predeterminada.

Solución

Clase: Persona

```
package ClasesUtiles;
import java.time.*;
import java.util.*;

/**
 * Esta clase denominada Persona modela una persona que posee un
 * nombre y un año de nacimiento y tiene un método para calcular la edad.
 * @version 1.2/2020
 */
public class Persona {
    String nombre; // Atributo para identificar el nombre de una persona
    LocalDate añoNacimiento; /* Atributo para identificar el año de
        nacimiento de una persona */

    /**
    * Constructor de la clase Persona
    * @param nombre Parámetro que define el nombre de una persona
    * @param añoNacimiento Parámetro que define el año de
    * nacimiento de una persona
    */
    Persona(String nombre, LocalDate añoNacimiento) {
        this.nombre = nombre;
        this.añoNacimiento = añoNacimiento;
    }
```

```
/**
* Método estático que permite calcular la edad a partir de la fecha
* actual y la fecha de nacimiento de una persona
* @param fechaNacimiento Parámetro que define la fecha de
* nacimiento de una persona
* @param fechaActual Parámetro que define la fecha actual del sistema
*/
public static int calcularEdad(LocalDate fechaNacimiento, LocalDate
    fechaActual) {
    // Se calcula la edad si las fechas de nacimiento y actual no son nulas
    if ((fechaNacimiento != null) && (fechaActual != null)) {
        /* Calcula un espacio de tiempo entre dos fechas y lo
            convierte a años */
        return Period.between(fechaNacimiento, fechaActual).getYears();
    } else {
        return 0; // Devuelve cero si alguna de las fechas es nula
    }
}

/**
* Método main que prueba la clase Persona solicitando ingresar por
* teclado el nombre y la fecha de nacimiento de una persona. Luego,
* calcula la edad de la persona en años, mostrando el resultado en
* pantalla
*/
public static void main(String args[]) {
    Scanner teclado = new Scanner(System.in);
    System.out.print("Nombre = ");
    String nombre = teclado.next();
    System.out.println("Fecha de nacimiento");
    System.out.print("Día = ");
    int día = teclado.nextInt();
    System.out.print("Mes (1-12) = ");
    int mes = teclado.nextInt();
    System.out.print("Año = ");
    int año = teclado.nextInt();
    // Convierte los datos de la fecha ingresados en un objeto LocalDate
    LocalDate fechaNacimiento = LocalDate.of(año, mes, día);
    LocalDate hoy = LocalDate.now(); /* Calcula la fecha actual del
        sistema */
    int años = calcularEdad(fechaNacimiento, hoy); /* Llama al
        método calcularEdad */
```

```
            System.out.println("Fecha de nacimiento = " + fechaNacimiento);
        System.out.println("Años = " + años);
        }
    }
```

Diagrama de clases

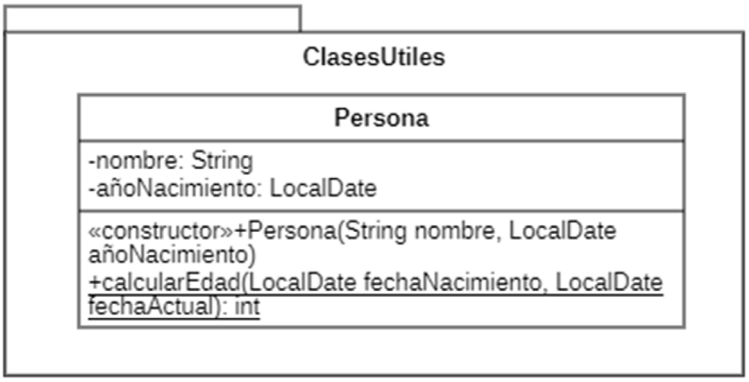


Figura 7.1. Diagrama de clases del ejercicio 7.1.

Explicación del diagrama de clases

Se ha definido un paquete denominado "ClasesUtiles" con una única clase que modela una Persona con los atributos privados: nombre (tipo *String*) y año de nacimiento (tipo *LocalDate*). La clase contiene un constructor que inicializa dichos atributos y un método estático para calcular la edad que tiene como parámetros la fecha de nacimiento y la fecha actual del sistema. Los métodos estáticos se identifican en UML porque su texto está subrayado.

Diagrama de objetos

Figura 7.2. Diagrama de objetos del ejercicio 7.1.

Ejecución del programa

```
Nombre = Pedro
Fecha de nacimiento
Día = 10
Mes (1-12) = 3
Año = 2001
Fecha de nacimiento = 2001-03-10
Años = 20
```

Figura 7.3. Ejecución del programa del ejercicio 7.1.

Ejercicios propuestos

- Realizar un programa que reciba dos fechas ingresadas por teclado y determinar la cantidad de años, meses y días entre las dos fechas. La primera fecha debe ser menor que la segunda.
- Un determinado producto tiene un nombre y una fecha de caducidad. Construir un programa que determine si un producto está caducado (la fecha actual es mayor que la fecha de caducidad).
- Desarrollar un programa que reciba por teclado los siguientes datos: una fecha y una cantidad de años, meses y días. El programa debe calcular la nueva fecha agregando la cantidad de años, meses y días a la fecha.

Ejercicio 7.2. Clase *StringTokenizer*

Cuando se ingresan o se generan datos en un programa es muy común insertar espacios en blanco entre palabras, pero, posteriormente, se deben extraer y separar individualmente. Los ejemplos no se limitan solo a espacios en blanco, numerosos archivos son generados en programas que utilizan un carácter o varios caracteres como delimitadores de los datos. Por lo tanto, se deben generar programas que extraigan los datos concretos, ya sea eliminando los espacios en blanco o los caracteres delimitadores. La clase *StringTokenizer* ayuda a realizar esta tarea.

La clase *StringTokenizer* permite dividir un *String* en *subStrings* o *tokens*, utilizando un *subString* como delimitador. El conjunto de delimitadores (los

caracteres que separan los *tokens*) se puede especificar en el momento de la creación (Joy *et al.*, 2013).

La creación de un objeto *StringTokenizer* utiliza el siguiente constructor:

StringTokenizer(String unaCadena, String delimitador)

La clase *StringTokenizer* presenta principalmente los siguientes métodos (API Java, 2020):

- ***countTokens()*:** calcula el número de veces que el método *nextToken()* puede invocarse antes de generar una excepción.
- ***hasMoreTokens()***: prueba si hay más *tokens* desde el *String* inicial.
- ***nextToken()*:** retorna el siguiente *token* desde el *String* inicial.

Para utilizar la clase *StringTokenizer* se debe importar el paquete: *java.util*.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Crear objetos de tipo *StringTokenizer* para extraer *tokens* de *Strings*.
- Aplicar diferentes métodos de la clase *StringTokenizer* para el manejo de *tokens*.

Enunciado: clase EjemploStringTokenizer

Se requiere desarrollar un programa que reciba un *String* para representar una colección de apellidos separados por espacios en blanco, por ejemplo: "PÉREZ GONZÁLEZ GUTIÉRREZ RODRÍGUEZ" y muestre en pantalla cada apellido en líneas separadas.

Además, el programa también recibe un *String* de apellidos con identificadores iniciales, por ejemplo, "#1-ALVÁREZ #1-FERNÁNDEZ #1-GÓMEZ #1-HERNÁNDEZ" y se desea mostrar en pantalla los apellidos en líneas separadas, pero sin los identificadores.

Instrucciones Java del ejercicio

Tabla 7.2. Instrucciones Java del ejercicio 7.2.

Clase	Método	Descripción
StringTokenizer	*boolean hasMoreTokens()*	Retorna si hay más caracteres separadores disponibles en el *String*.
	String nextToken()	Retorna los siguientes caracteres separadores del *String*.

Solución

Clase: EjemploStringTokenizer

```
package ClasesUtiles;
import java.util.*;
/**
 * Esta clase denominada EjemploStringTokenizer permite trabajar con
 * la clase StringTokenizer que manipula Strings.
 * @version 1.2/2020
 */
public class EjemploStringTokenizer {
    /**
     * Método main que crea un StringTokenizer y realiza varias acciones
     * sobre el mismo
     */
    public static void main(String args[]) {
        // Crea un objeto StringTokenizer
        StringTokenizer cadena1 = new StringTokenizer("PÉREZ
            GONZÁLEZ GUTIÉRREZ RODRÍGUEZ");
        /* Divide el String en palabras separadas, eliminando los espacios
            en blanco */
        while (cadena1.hasMoreTokens()) {
        System.out.println(cadena1.nextToken());
    }
    System.out.println();
    /* Crea un objeto StringTokenizer y los caracteres separadores son
        "#-1" */
    StringTokenizer cadena2 = new StringTokenizer("#1-ALVÁREZ
        #1-FERNÁNDEZ #1-GÓMEZ #1-HERNÁNDEZ","#1-");
```

```
            /* Divide el String en palabras separadas, eliminando los
                caracateres separadores */
            while (cadena2.hasMoreTokens()) {
                System.out.println(cadena2.nextToken());
            }
        }
    }
```

Diagrama de clases

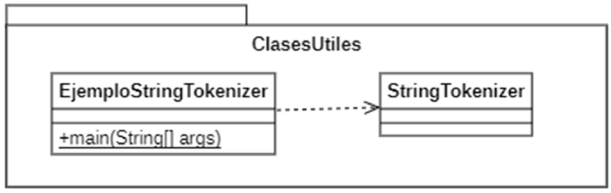


Figura 7.4. Diagrama de clases del ejercicio 7.2.

Explicación del diagrama de clases

Se ha definido un paquete denominado "ClasesUtiles" con una única clase que modela un ejemplo de uso de la clase *StringTokenizer*. Como la clase EjemploStringTokenizer utiliza a *StringTokenizer*, se ha modelado esta interacción a través de una relación de dependencia entre ellas.

La relación entre estas clases se especifica por medio de una relación de dependencia, la cual se modela en UML con una línea entrecortada con punta en flecha en un extremo. La relación se lee como "EjemploStringTokenizer depende de la clase *StringTokenizer*". Una relación de dependencia indica que una clase depende de otra, es decir, si una clase cambia la otra también cambia.

Diagrama de objetos

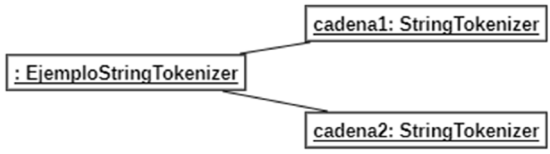


Figura 7.5. Diagrama de objetos del ejercicio 7.2.

Ejecución del programa

```
PEREZ
GONZALEZ
GUTIERREZ
RODRIGUEZ

ALVAREZ
FERNANDEZ
GOMEZ
HERNANDEZ
```

Figura 7.6. Ejecución del programa del ejercicio 7.2.

Ejercicios propuestos

- Un palíndromo es una cadena de caracteres que se lee igual tanto hacia adelante como hacia atrás. Ejemplos: "radar" y "31413". Escriba un método con valor de retorno *boolean* que tenga como un único parámetro la cadena a evaluar. El método debería devolver verdadero si y solo si su parámetro es un palíndromo.
- Modificar el programa anterior para evaluar si una frase es palíndroma. Por ejemplo: "Se van sus naves", "Anita lava la tina". El programa no debe tener en cuenta el texto en mayúsculas o signos de puntuación.
- Se requiere desarrollar un programa que lea un archivo de texto, el cual tiene los siguientes datos:

  Nombre del estudiante1; nota1; nota2; […] ; nota10

  Nombre del estudiante2; nota1; nota2; […] ; nota10

  El archivo consiste en un listado de estudiantes, cada línea corresponde a los datos de un estudiante: su nombre y diez notas. Los datos están separados por punto y comas.

  El programa debe generar un nuevo archivo de texto con el nombre de cada estudiante, promedio y desviación estándar de sus notas.

Ejercicio 7.3. Clase *NumberFormat*

La clase *NumberFormat* proporciona la interfaz para dar formato a elementos numéricos. Entre las posibles opciones de formateo numérico se incluyen: formatos para puntos decimales, separadores de miles o incluso los dígitos decimales particulares utilizados (Joy *et al.*, 2013).

La clase *NumberFormat* presenta los siguientes métodos (API Java, 2020):

- ***getInstance()*:** obtiene el formato del idioma actual.
- ***getCurrencyInstance()*:** obtiene el formato del idioma actual con formato de moneda.
- ***getMaximumFractionDigits()*:** obtiene el número máximo de dígitos permitidos en la porción decimal del número.
- ***getMaximumIntegerDigits()*:** obtiene el número mínimo de dígitos permitidos en la porción entera del número.
- ***getPercentInstance()*:** obtiene un formato porcentual para el objeto *Local* por defecto.

Para utilizar la clase *NumberFormat* se debe importar el paquete: *java.text*.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Crear objetos de tipo *NumberFormat* para darle formato a valores numéricos.
- Dar formato de porcentaje, moneda y valores decimales a valores numéricos.

Enunciado: clase Estudiante

Se requiere realizar un programa que modele un estudiante. Los datos de un estudiante son: nombre (tipo *String*), porcentaje de avance de la carrera (tipo *double*) y costo de la matrícula (tipo *double*). La clase debe tener un constructor que inicialice dichos atributos. Además, se requiere implementar un método imprimir que muestre en pantalla los valores de los atributos teniendo en cuenta que:

- El porcentaje de avance de la carrera debe estar en formato de porcentaje con dos dígitos decimales.
- El costo de la matrícula debe estar en formato moneda.

Instrucciones Java del ejercicio

Tabla 7.3. Instrucciones Java del ejercicio 7.3.

Clase	Método	Descripción
NumberFormat	*String format(double* número)	Aplica un formato definido a una variable.
	void setMinimumFraction Digits (*int* valor)	Establece el número mínimo de dígitos permitido en la porción de fracción de un número.
	NumberFormat getCurrencyInstance()	Devuelve un formato de moneda para la configuración regional predeterminada actual.

Solución

Clase: Estudiante

```
package ClasesUtiles;
import java.text.*;
import java.util.*;

/**
 * Esta clase denominada Estudiante modela un estudiante universitario
 * que tiene como atributos un nombre, un porcentaje de avance en la
 * carrera y un costo de su matrícula.
 * @version 1.2/2020
 */
public class Estudiante {
    String nombre; // Atributo que identifica el nombre de un estudiante
    double porcentajeAvance; /* Atributo que identifica el porcentaje de
        avance de un estudiante */
    double costoMatrícula; /* Atributo que identifica el costo de la
        matrícula de un estudiante */
    /**
     * Constructor de la clase Estudiante
     * @param nombre Parámetro que define el nombre de un estudiante
```

```
    * @param porcentajeAvance Parámetro que define el porcentaje de
    *avance del estudiante para finalizar su carrera
    * @param costoMatrícula Parámetro que define el costo de la
    * matrícula para un estudiante
    */
    Estudiante(String nombre, double porcentajeAvance, double
        costoMatrícula) {
        this.nombre = nombre;
        this.porcentajeAvance = porcentajeAvance;
        this.costoMatrícula = costoMatrícula;
    }

    /**
    * Método que muestra en pantalla los datos de un estudiante. El
    * método muestra los datos con un cierto formato
    */
    void imprimir() {
        System.out.println("Nombre del estudiante = " + nombre);
        // Define un formato en porcentaje
        NumberFormat formatoPorcentaje = NumberFormat.
            getPercentInstance();
        /* Define un formato de porcentaje con mínimo dos digitos de
            fracción */
        formatoPorcentaje.setMinimumFractionDigits(2);
        // Aplica el formato de porcentaje al atributo porcentajeAvance
        System.out.println("Porcentaje de avance = " +
formatoPorcentaje.format(porcentajeAvance));
        // Define un formato como moneda
        NumberFormat formatoImporte = NumberFormat.
            getCurrencyInstance();
        // Aplica el formato moneda al atributo costoMatrícula
        System.out.println("Costo de la matrícula = " + formatoImporte.
            format(costoMatrícula));
    }

    /**
    * Método main que solicita ingresar por teclado el nombre del
    * estudiante, el porcentaje de avance de la carrera y el costo de la
    * matrícula. Luego, crea un estudiante con estos datos y muestra los
    * datos en pantalla dándoles un cierto formato.
    */
```

```
        public static void main(String args[]) {
            Scanner sc = new Scanner(System.in);
            System.out.println("Nombre del estudiante = ");
            String nombre = sc.next();
            System.out.println("Porcentaje de avance (0,XXXX) = ");
            double porcentaje = sc.nextDouble(); /* Captura un dato double
                por teclado */
            System.out.println("Costo de la matrícula = $");
            double matrícula = sc.nextDouble(); /* Captura un dato double
                por teclado */
            // Crea un objeto Estudiante
            Estudiante estudiante = new Estudiante(nombre, porcentaje,
                matrícula);
            estudiante.imprimir();
        }
    }
```

Diagrama de clases

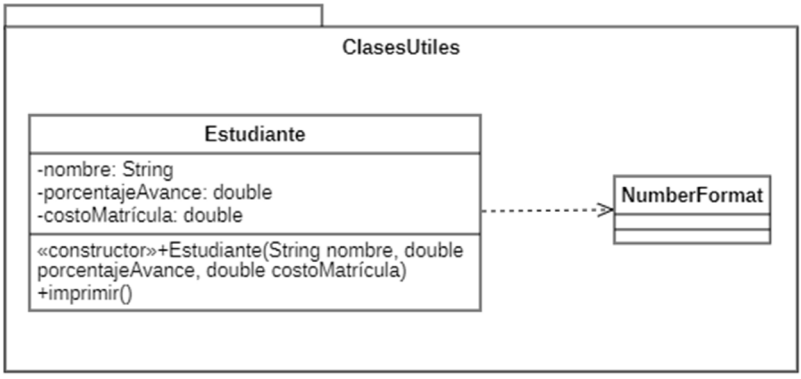


Figura 7.7. Diagrama de clases del ejercicio 7.3.

Explicación del diagrama de clases

Se ha definido un paquete denominado "ClasesUtiles" con una única clase que modela un estudiante, los atributos privados son: su nombre, el porcentaje de avance y el costo de la matrícula. La clase Estudiante cuenta con un constructor que inicializa sus atributos y un método para imprimir los datos de un estudiante en pantalla.

La clase Estudiante tiene una relación de dependencia con la clase predefinida *NumberFormat*. La relación se expresa en UML como una línea entrecortada con punto de flecha en un extremo, esta se conecta con la clase de la cual depende. En este ejemplo, la clase Estudiante depende de *NumberFormat*. Por lo tanto, si hay un cambio en la clase *NumberFormat* debe haber un cambio en Estudiante, la clase dependiente.

Diagrama de objetos

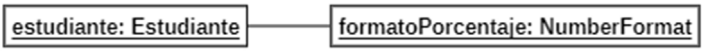


Figura 7.8. Diagrama de objetos del ejercicio 7.3.

Ejecución del programa

```
Nombre del estudiante =
Pablo
Porcentaje de avance (0,XXXX) =
0,8
Costo de la matrícula = $
1500000
Nombre del estudiante = Pablo
Porcentaje de avance = 80,00 %
Costo de la matrícula = $ 1.500.000,00
```

Figura 7.9. Ejecución del programa del ejercicio 7.3.

Ejercicios propuestos

- Un archivo de texto contiene el valor de los salarios mensuales de un grupo de trabajadores en el formato:
  Empleado1: salario1
  Empleado2: salario2
  Por ejemplo:
  Pepito Pérez: 1 000 000
  Paquita Gómez: 2 000 000
  Se requiere generar un nuevo archivo de texto, pero con los valores monetarios formateados:
  Pepito Pérez: $ 1 000 000
  Paquita Gómez: $ 2 000 000

Ejercicio 7.4. Generación de números aleatorios

La clase *Random* se utiliza para generar números aleatorios. Sin embargo, varias aplicaciones utilizan el método *Math.random()*, ya que es más simple de utilizar (Joy *et al.*, 2013). El formato del método es:

public static double random()

El método retorna un valor *double* con un signo positivo, mayor o igual que 0.0 y menor que 1.0.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para generar números aleatorios y los usará para solucionar diferentes problemas estadísticos.

Enunciado: clase LanzamientoMonedas

Se requiere hacer un programa que simule el lanzamiento de una moneda *n* veces y realice el conteo de cuántas veces cayó la cara o la cruz de la moneda. El programa debe mostrar en pantalla la cantidad de veces que ocurrió cada resultado y la probabilidad de obtener cada resultado.

Solución

Clase: LanzamientoMonedas

```
package ClasesUtiles;
import java.util.*;

/**
 * Esta clase denominada LanzamientoMonedas modela un proceso de
 * lanzar una moneda cierta cantidad de veces, calculando el número de
 * lados de la moneda obtenidos y los porcentajes correspondientes.
 * @version 1.2/2020
 */
public class LanzamientoMonedas {
```

```
    /**
    * Método main que solicita ingresar por teclado la cantidad de
    * lanzamientos de moneda a realizar, para luego simular el
    * lanzamiento aleatorio de la moneda y obtener el número de lados
    * de la moneda y los porcentajes correspondientes
    */
    public static void main(String args[]) {
        int númeroCaras = 0; // Contador del número de caras
        int númeroCruces = 0; // Contador del número de cruces
        Scanner sc = new Scanner(System.in);
        System.out.println("Número de lanzamientos = ");
        int intentos = sc.nextInt(); /* Captura el número de lanzamientos
            ingresado */
        for (int i = 0; i < intentos; i++) { /* Ciclo para realizar la cantidad
            de lanzamientos */
            double número = Math.random()*10; /* Genera un número
                aleatorio entre 0 y 10 */
            if (número < 5) { /* Si el número es menor que 5, se considera
                cara */
                númeroCaras++; // Se contabiliza el número de caras
            } else { // Si el número es mayor o igual a 5, se considera cruz
                númeroCruces++; // Se contabiliza el número de cruces
            }
        }
        System.out.println("Número de caras = " + númeroCaras);
        System.out.println("Número de cruces = " + númeroCruces);
        /* Se calculan porcentajes de cada lado de la moneda. Si el número
            de lanzamientos es alto, ambos porcentajes deben estar cerca
            a 0.5 */
        double porcentajeCaras = (double) númeroCaras/intentos;
        double porcentajeSellos = (double) númeroCruces/intentos;
        System.out.println("Porcentaje de caras = " + porcentajeCaras);
        System.out.println("Porcentaje de cruces = " + porcentajeSellos);
    }
}
```

Diagrama de clases

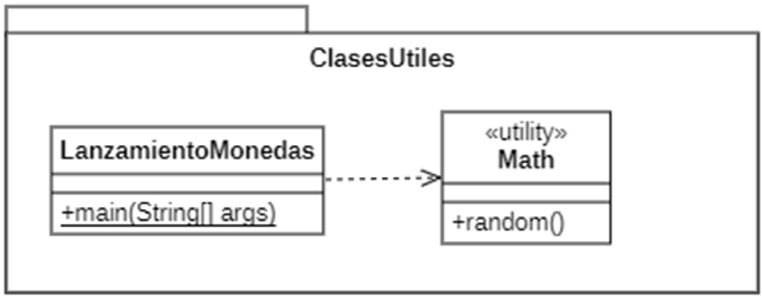


Figura 7.10. Diagrama de clases del ejercicio 7.4.

Explicación del diagrama de clases

Se ha definido un paquete denominado "ClasesUtiles" con una única clase que modela el lanzamiento de monedas. La cual utiliza la clase predefinida *Math* y en concreto, el método *random* que permite generar números aleatorios entre 0 y 1. La clase *Math* está identificada con el estereotipo *<<utility>>* para señalar que es una utilidad proporcionada por Java.

La clase LanzamientoMonedas tiene una relación de dependencia con la clase predefinida *Math*. De acuerdo con lo que ya se ha mencionado, la relación de dependencia se expresa en UML como una línea entrecortada con punto de flecha en un extremo, esta se conecta con la clase de la cual depende. En este ejemplo, la clase LanzamientoMoneda depende de *Math*. Por lo tanto, si hay un cambio en la clase *Math* debe haber un cambio en LanzamientoMoneda, la clase dependiente.

Diagrama de objetos

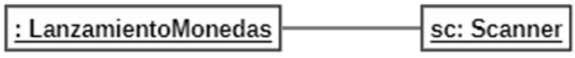


Figura 7.11. Diagrama de objetos del ejercicio 7.4.

Ejecución del programa

```
Número de lanzamientos =
1000
Número de caras = 488
Número de cruces = 512
Porcentaje de caras = 0.488
Porcentaje de cruces = 0.512
```

Figura 7.12. Ejecución del programa del ejercicio 7.4.

Ejercicios propuestos

- Desarrollar un programa que asignará una variable *char* de un valor aleatorio del conjunto: { 'A', 'B', 'C', 'D', 'E' }.
- Desarrollar un programa que simule una jornada de votación con 5 candidatos y 10 000 votantes. El programa debe mostrar los resultados en pantalla con el siguiente formato de la tabla 7.4.

Tabla 7.4. Formato de presentación de datos

	Candidato 1	Candidato 2	Candidato 3	Candidato 4	Candidato 5	En blanco
Votos						
Porcentaje						

Además, el programa debe indicar cuál fue el candidato que ganó la votación.

Capítulo 8
Interfaz gráfica de usuario

El propósito general del octavo capítulo es presentar y aplicar las clases más importantes que conforman el paquete *swing* de Java, para crear interfaces gráficas de usuario (GUI, del inglés, *Graphical User Interface*). Las GUI son programas informáticos que utilizan un conjunto de imágenes y objetos gráficos para representar la información y acciones disponibles en la interfaz proporcionando un entorno visual para permitir la comunicación con el programa. En este capítulo se presentan cinco ejercicios para aplicar los conceptos de construcción de interfaces gráficas de usuario utilizando componentes *swing* de Java.

- **Ejercicio 8.1. Paquete *swing***
  El paquete *swing* de Java incluye una serie de componentes gráficos clasificados en contenedores de alto nivel, intermedios y componentes específicos. El primer ejercicio propone el desarrollo de un programa que utilice estos componentes gráficos separando las clases orientadas a la construcción de la GUI de las clases orientadas al modelo de los conceptos de la solución del problema.

- **Ejercicio 8.2. Componentes *swing***
  Java poseen una gran cantidad de componentes gráficos para elaborar interfaces gráficas de usuario tanto simples

como sofisticadas. El segundo ejercicio presenta el desarrollo de un programa que aplica estos componentes gráficos.

- **Ejercicio 8.3. Gestión de eventos**
  La interacción del usuario con los componentes gráficos es gestionada por medio de objetos especializados denominados oyentes o *listeners*, que se encargan de gestionar los eventos generados a partir de estas interacciones. El tercer ejercicio propone el desarrollo de una interfaz gráfica de usuario con varios oyentes que se encargan de procesar los eventos generados.

- **Ejercicio 8.4. Cuadros de diálogo**
  Existen componentes gráficos específicos, incluidos en el paquete *swing*, que permiten generar cuadros de diálogo, es decir, un tipo de ventana que establece una comunicación simple entre el usuario y el programa. El cuarto ejercicio presenta un problema donde se deben generar algunos cuadros de diálogo y a su vez incorporar diversos componentes gráficos.

- **Ejercicio 8.5. Gestión de contenidos**
  Java posee clases especializadas que se encargan de la organización de los componentes gráficos en una ventana. Estos elementos se denominan administradores de diseño o *layouts*. El último ejercicio de este capítulo está orientado a generar una interfaz gráfica de usuario que incluye el uso de *layouts* en su diseño e implementación. A su vez, pone en práctica la generación de componentes gráficos, cuadros de diálogo y generación y procesamiento de eventos.

Los diagramas UML utilizados en este capítulo son los diagramas de clase y de objetos. Los paquetes permiten agrupar elementos UML y en los ejercicios se organizarán las clases en un único paquete debido a la pequeña cantidad de clases en las soluciones presentadas. En primer lugar, los diagramas de clase modelan la estructura estática de los programas. Generalmente, los diagramas de clase de los ejercicios presentados, estarán conformados por una o varias clases relacionadas e incorporan notaciones para identificar tipos de atributos y métodos. En segundo lugar, los diagramas de objeto identifican los objetos creados en el método *main* del programa.

Ejercicio 8.1. Paquete *swing*

El paquete de componentes *Java swing* GUI contiene clases e interfaces que permiten desarrollar la interfaz gráfica de usuario de los programas desarrollados (Yang, 2018). Anteriormente, los componentes estaban en un paquete denominado AWT. El cual es una interfaz Java para el código GUI del sistema nativo presente en el sistema operativo que ejecuta el programa (Murach, Boehm y Delameter, 2017). Mientras que los componentes *swing* son más portables y presentan la misma apariencia en todas las plataformas.

Sin embargo, los componentes *swing* utilizan la infraestructura de AWT, incluyendo el modelo de eventos AWT, el cual gestiona eventos como la pulsación del teclado, ratón, etc. Los programas *swing* deben importar los paquetes: *javax.swing.** y AWT: *java.awt.** y *java.awt.event.**.

La estructura básica de una interfaz gráfica utilizando *swing* tiene los siguientes elementos (Deitel y Deitel, 2017):

- **Contenedores de alto nivel**: representan las ventanas de la interfaz gráfica de usuario (clase *JFrame* y cuadros de diálogo *JDialog*). Estos tienen contenedores intermedios.
- **Contenedores intermedios**: agrupan componentes de acuerdo con algún criterio lógico (clase *JPanel* y clase *JScrollPane*). A su vez, los contenedores intermedios contienen componentes.
- **Componentes**: elementos de bajo nivel que representan diferentes componentes gráficos, estos interactúan con los usuarios a través de eventos.

Los pasos básicos para la construcción de una interfaz gráfica de usuario son (Yang, 2018):

1. Crear una nueva clase para una ventana (o directamente instanciar un objeto *JFrame*).
2. Crear los componentes de la interfaz gráfica de usuario.
3. Crear uno o varios contenedores intermedios.
4. Asociar los componentes al contenedor.
5. Asociar el contenedor a la ventana.
6. Hacer visible la ventana.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Desarrollar interfaces de gráficas de usuario básicas.
- Conocer y aplicar diferentes componentes básicos de Java.

Enunciado: Persona

Se requiere desarrollar un programa con una interfaz gráfica de usuario que genere una ventana para solicitar los datos de una persona: nombre, apellidos, dirección y teléfono, todos de tipo *String*. Una vez se reciban estos datos, la persona se debe agregar a una lista de personas.

La lista se puede consultar y si se selecciona una persona específica, es posible eliminar la persona seleccionada. También se permite borrar todas las personas de la lista.

Instrucciones Java del ejercicio

Tabla 8.1. Instrucciones Java del ejercicio 8.1.

Clase	Método	Descripción
JFrame	*JFrame()*	Constructor de la clase JFrame.
	void setTitle(String título)	Establece el título de la ventana con el *String* especificado.
	void setSize(int x, int y)	Cambia el tamaño del componente para que tenga una anchura *x* y una altura *y*.
	void setLocationRelativeTo (Component c)	Establece la ubicación de la ventana en relación con el componente especificado.
	void setDefaultCloseOperation (opciones)	Usado para especificar una de las siguientes opciones del botón de cierre: EXIT_ON_CLOSE, HIDE_ON_CLOSE, DISPOSE_ON_CLOSE o DO_NOTHING_ ON_CLOSE.
	void setResizable(boolean resizable)	Para evitar que se cambie el tamaño de la ventana.
	void setVisible(boolean b)	Muestra u oculta la ventana según el valor del parámetro b.

Clase	Método	Descripción
ActionListener	*ActionListener*	Interface que debe ser implementada para gestionar eventos.
	void actionPerformed(actionEvent e)	Se invoca cuando ocurre un evento.
Container	*Container getContentPane()*	Retorna un objeto *ContentPane* de la ventana.
	void setLayout(LayoutManager mgr)	Establece el *layout* de la ventana.
	Component add(Component comp)	Añade el componente especificado al final del contenedor.
Component	*void addActionListener(this)*	Añade un oyente de eventos al componente actual.
	void setBounds(int x, int y, int ancho, int alto)	Mueve y cambia el tamaño del componente.
JLabel	*JLabel()*	Constructor de la clase *JLabel*.
	void setText(String text)	Define una línea de texto que mostrará este componente.
JTextField	*JTextField()*	Constructor de la clase *JTextField*.
	String getText()	Retorna el texto contenido en el componente de texto.
JButton	*JButton()*	Constructor de la clase *JButton*.
	void setText(String text)	Define una línea de texto que mostrará este componente.
JList	*JList()*	Constructor de la clase JList.
	void setText(String text)	Define una línea de texto que mostrará este componente.
	void setSelectionMode(int modoSelección)	Establece el modo de selección de la lista.
	void setModel(ListModel<E> model)	Establece el modelo que representa el contenido de la lista.
	int getSelectedIndex()	Devuelve el índice seleccionado.
DefaultList Model	*DefaultListModel()*	Constructor de la clase *DefaultListModel*.
	void addElement(Element e)	Añade el componente especificado al final de la lista.
	void removeElementAt(int índice)	Elimina el componente en el índice especificado.
	void removeAllElements()	Elimina todos los componentes de la lista y coloca su tamaño en cero.
	void clear()	Elimina todos los elementos de la lista.

Clase	Método	Descripción
JScrollPane	*JScrollPane()*	Constructor de la clase *JScrollPane*.
	void setViewportView(Component view)	Crea una ventana gráfica si es necesario y luego establece su vista.
Event	*Object getSource()*	El objeto sobre el cual el evento inicialmente ha ocurrido.
JOptionPane	*void showMessageDialog (Component componentePadre, Object mensaje)*	Crea un cuadro de diálogo.

Solución

Clase: VentanaPrincipal

```
package Personas;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;

/**
 * Esta clase denominada VentanaPrincipal define una interfaz gráfica
 * que permitirá crear una persona y agregarla a un vector de personas.
 * Luego, se puede eliminar una persona seleccionada o borrar todas las
 * personas.
 * @version 1.2/2020
 */
public class VentanaPrincipal extends JFrame implements
ActionListener {
    private ListaPersonas lista; // El objeto ListaPersonas de la aplicación
    private Container contenedor; /* Un contenedor de elementos
        gráficos */
    // Etiquetas estáticas para los nombres de los atributos
    private JLabel nombre, apellidos, teléfono, dirección;
    // Campos de ingreso de texto
    private JTextField campoNombre, campoApellidos, campoTeléfono,
        campoDirección;

    private JButton añadir, eliminar, borrarLista; // Botones
    private JList listaNombres; // Lista de personas
    private DefaultListModel modelo; // Objeto que modela la lista
    private JScrollPane scrollLista; // Barra de desplazamiento vertical
```

```
/**
* Constructor de la clase VentanaPrincipal
*/
public VentanaPrincipal(){
    lista = new ListaPersonas(); // Crea la lista de personas
    inicio();
    setTitle("Personas"); // Establece el título de la ventana
    setSize(270,350); // Establece el tamaño de la ventana
    setLocationRelativeTo(null); /* La ventana se posiciona en el
        centro de la pantalla */
    // Establece que el botón de cerrar permitirá salir de la aplicación
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setResizable(false); /* Establece que el tamaño de la ventana no
        se puede cambiar */
}

/**
* Método que crea la ventana con sus diferentes componentes
* gráficos
*/
private void inicio() {
    contenedor = getContentPane(); /* Obtiene el panel de
        contenidos de la ventana */
    contenedor.setLayout(null); /* Establece que el panel no tiene
        asociado ningún layout */

    // Establece la etiqueta y el campo nombre
    nombre = new JLabel();
    nombre.setText("Nombre:");
    nombre.setBounds(20, 20, 135, 23); /* Establece la posición de
        la etiqueta nombre */
    campoNombre = new JTextField();
    // Establece la posición del campo de texto nombre
    campoNombre.setBounds(105, 20, 135, 23);

    // Establece la etiqueta y el campo apellidos
    apellidos = new JLabel();
    apellidos.setText("Apellidos:"); /* Establece la posición de la
        etiqueta apellidos */
    apellidos.setBounds(20, 50, 135, 23);
    campoApellidos = new JTextField();
    // Establece la posición del campo de texto apellidos
    campoApellidos.setBounds(105, 50, 135, 23);
```

```
// Establece la etiqueta y el campo teléfono
teléfono = new JLabel();
teléfono.setText("Teléfono:");
teléfono.setBounds(20, 80, 135, 23); /* Establece la posición de
    la etiqueta teléfono */
campoTeléfono = new JTextField();
// Establece la posición del campo de texto teléfono
campoTeléfono.setBounds(105, 80, 135, 23);

// Establece la etiqueta y el campo dirección
dirección = new JLabel();
dirección.setText("Dirección:");
dirección.setBounds(20, 110, 135, 23); /* Establece la posición
    de la etiqueta dirección */
campoDirección = new JTextField();
// Establece la posición del campo de texto dirección
campoDirección.setBounds(105, 110, 135, 23);

// Establece el botón Añadir persona
añadir = new JButton();
añadir.setText(«Añadir»);
añadir.setBounds(105, 150, 80, 23); /* Establece la posición del
    botón Añadir persona */
/* Agrega al botón un ActionListener para que gestione eventos
    del botón */
añadir.addActionListener(this);

// Establece el botón Eliminar persona
eliminar= new JButton();
eliminar.setText("Eliminar");
eliminar.setBounds(20, 280, 80, 23); /* Establece la posición del
    botón Eliminar persona */
/* Agrega al botón un ActionListener para que gestione eventos
    del botón */
eliminar.addActionListener(this);

// Establece el botón Borrar lista
borrarLista= new JButton();
borrarLista.setText("Borrar Lista");
borrarLista.setBounds(120, 280, 120, 23); /* Establece la
    posición del botón Borrar lista */
/* Agrega al botón un ActionListener para que gestione eventos
    del botón */
borrarLista.addActionListener(this);
```

```
        // Establece la lista gráfica de personas
        listaNombres = new JList();
        /* Establece que se pueda seleccionar solamente un elemento de
            la lista */
        listaNombres.setSelectionMode(ListSelectionModel.SINGLE_
            SELECTION);
        modelo = new DefaultListModel();

        // Establece una barra de desplazamiento vertical
        scrollLista = new JScrollPane();
        // Establece la posición de la barra de desplazamiento vertical
        scrollLista.setBounds(20, 190 ,220, 80);
        // Asocia la barra de desplazamiento vertical a la lista de personas
        scrollLista.setViewportView(listaNombres);

        // Se añade cada componente gráfico al contenedor de la ventana
        contenedor.add(nombre);
        contenedor.add(campoNombre);
        contenedor.add(apellidos);
        contenedor.add(campoApellidos);
        contenedor.add(teléfono);
        contenedor.add(campoTeléfono);
        contenedor.add(dirección);
        contenedor.add(campoDirección);
        contenedor.add(añadir);
        contenedor.add(eliminar);
        contenedor.add(borrarLista);
        contenedor.add(scrollLista);
    }

    /**
    * Método que gestiona los eventos generados en la ventana principal
    */
    @Override
    public void actionPerformed(ActionEvent evento) {
        if (evento.getSource() == añadir) { // Si se pulsa el botón añadir
            añadirPersona(); // Se invoca añadir persona
        }
        if (evento.getSource() == eliminar) { /* Si se pulsa el botón
            eliminar */
            /* Se invoca el método eliminarNombre que elimina el
                elemento seleccionado */
            eliminarNombre(listaNombres.getSelectedIndex() );
        }
```

```
        if (evento.getSource() == borrarLista) { /* Si se pulsa el botón
            borrar lista */
            borrarLista(); // Se invoca borrar lista
        }
    }

    /**
    * Método que agrega una persona al vector de personas y a la lista
    * gráfica de personas
    */
    private void añadirPersona() {
        /* Se obtienen los campos de texto ingresados y se crea una
            persona */
        Persona p = new Persona(campoNombre.getText(),
            campoApellidos.getText(),
                campoTeléfono.getText(), campoDirección.getText());
        lista.añadirPersona(p); /* Se añade una persona al vector de
            personas */
        String elemento = campoNombre.getText() + "-" +
            campoApellidos.getText() +
                "-" + campoTeléfono.getText() + "-" + campoDirección.
                    getText();
        modelo.addElement(elemento); /* Se agrega el texto con los
            datos de la persona al JList */
        listaNombres.setModel(modelo);
        // Se colocan todos los campos de texto nulos
        campoNombre.setText("");
        campoApellidos.setText("");
        campoTeléfono.setText("");
        campoDirección.setText("");
    }

    /**
    * Método que elimina una persona del vector de personas y de la
    * lista gráfica de personas en la ventana
    * @param indice Parámetro que define la posición de la persona a
    * eliminar
    */
    private void eliminarNombre(int indice) {
        if (indice >= 0) { // Si la posición existe
            modelo.removeElementAt(indice); /* Se elimina la persona
                seleccionada de la lista gráfica */
```

```
                lista.eliminarPersona(indice); /* Se elimina la persona
                    seleccionada del vector de personas */
            } else { /* Si no se seleccionó ninguna persona, se genera un
                mensaje de error */
                JOptionPane.showMessageDialog(null, "Debe seleccionar un
                    elemento","Error",
                    JOptionPane.ERROR_MESSAGE);
            }
        }

        /**
        * Método que elimina todas las personas del vector de personas
        */
        private void borrarLista() {
            lista.borrarLista(); // Se eliminan todas las personas del vector
            modelo.clear(); // Limpia el JList, la lista gráfica de personas
        }
}
```

Clase: ListaPersonas

```
package Personas;
import java.util.*;

/**
 * Esta clase denominada ListaPersonas define un vector de Personas.
 * @version 1.2/2020
 */
public class ListaPersonas {
    Vector listaPersonas; // Atributo que identifica un vector de personas

    /**
    * Constructor de la clase ListaPersonas
    */
    public ListaPersonas() {
        listaPersonas = new Vector(); // Crea el vector de personas
    }

    /**
    * Método que permite agregar una persona al vector de personas
    * @param p Parámetro que define la persona a agregar al vector de
    * personas
    */
```

```
    public void añadirPersona(Persona p) {
        listaPersonas.add(p);
    }

    /**
    * Método que permite eliminar una persona del vector de personas
    * @param i Parámetro que define la posición a eliminar en el vector
    * de personas
    */
    public void eliminarPersona(int i) {
        listaPersonas.removeElementAt(i);
    }

    /**
    * Método que permite eliminar todos los elementos del vector de
    * personas
    */
    public void borrarLista() {
        listaPersonas.removeAllElements();
    }
}
```

Clase: Persona

```
package Personas;

/**
 * Esta clase denominada Persona modela una persona que cuenta con
 * los atributos: nombre, apellidos, teléfono y dirección.
 * @version 1.2/2020
 */
public class Persona {
    String nombre; // Atributo que identifica el nombre de una persona
    String apellidos; /* Atributo que identifica los apellidos de una
        persona */
    String teléfono; // Atributo que identifica el teléfono de una persona
    String dirección; /* Atributo que identifica la dirección de una
        persona */

    /**
    * Constructor de la clase Persona
    * @param nombre Parámetro que define el nombre de una persona
    * @param apellidos Parámetro que define los apellidos de una
    * persona
```

```
     * @param teléfono Parámetro que define el teléfono de una persona
     * @param dirección Parámetro que define la dirección de una
     * persona
     */
    public Persona(String nombre, String apellidos, String teléfono,
        String dirección) {
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.teléfono = teléfono;
        this.dirección = dirección;
    }
}
```

Clase: Principal

```
package Personas;

/**
 * Esta clase define el punto de ingreso al programa de gestión de
 * personas.
 * Por lo tanto, cuenta con un método main de acceso al programa.
 * @version 1.2/2020
 */
public class Principal {

     /**
     * Método main que sirve de punto de entrada al programa
     */
    public static void main(String[] args) {
        VentanaPrincipal miVentanaPrincipal; /* Define la ventana
            principal */
        miVentanaPrincipal= new VentanaPrincipal(); /* Crea la ventana
            principal */
        miVentanaPrincipal.setVisible(true); /* Establece la ventana
            como visible */
    }
}
```

Diagrama de clases

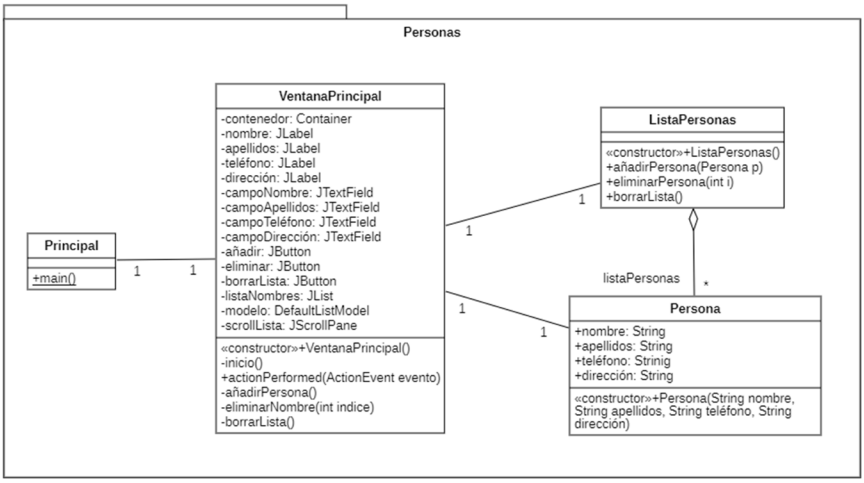


Figura 8.1. Diagrama de clases del ejercicio 8.1.

Explicación del diagrama de clases

Se ha definido un paquete denominado "Personas" que incluye un conjunto de clases. El punto de entrada al programa es la clase Principal que cuenta con el método *main*. La clase Principal está vinculada mediante una relación de asociación con la clase VentanaPrincipal que posee atributos privados para identificar los diferentes componentes gráficos de la ventana: un contenedor de componentes gráficos (*Container*), etiquetas (*JLabel*), campos de texto (*JTextField*), botones (*JButton*), una lista (*JList* y *DefaultListModel*) y una barra de desplazamiento vertical (*JScrollPane*). La clase VentanaPrincipal cuenta con un constructor y métodos para generar la ventana gráfica con sus componentes (inicio) y para gestionar los diferentes eventos surgidos al interactuar con esta ventana (*actionPerformed*).

La clase VentanaPrincipal se relaciona con las clases ListaPersonas y Persona a la par que se generan los diferentes eventos, cuando el usuario interactúa con los botones presentes en la ventana. Estas relaciones de asociación tienen una multiplicidad de 1 a 1, ya que la VentanaPrincipal tiene una única lista de personas y cuando crea una persona, solo es posible crear una a la vez.

La clase ListaPersonas tiene una relación de agregación con la clase Persona. La relación de agregación se expresa en UML por medio de una

línea continua que tiene un rombo claro adyacente a la clase que representa el todo. En este caso, la clase que representa el todo es ListaPersonas, la cual está constituida de objetos tipo Persona.

Las clases ListaPersona y Persona tienen sus atributos, constructor y métodos correspondientes. Los atributos de Persona son: nombre, apellidos, dirección y teléfono y tiene un constructor para inicializar estos atributos. La clase ListaPersona tiene un constructor y métodos para añadir, eliminar personas y borrar la lista de personas.

Diagrama de objetos

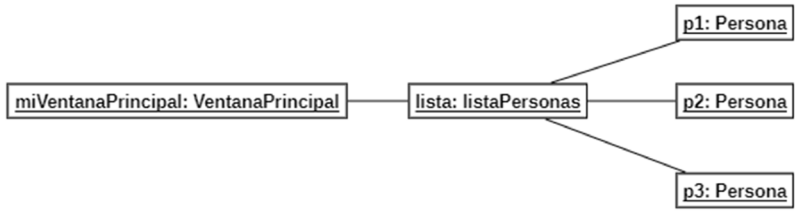
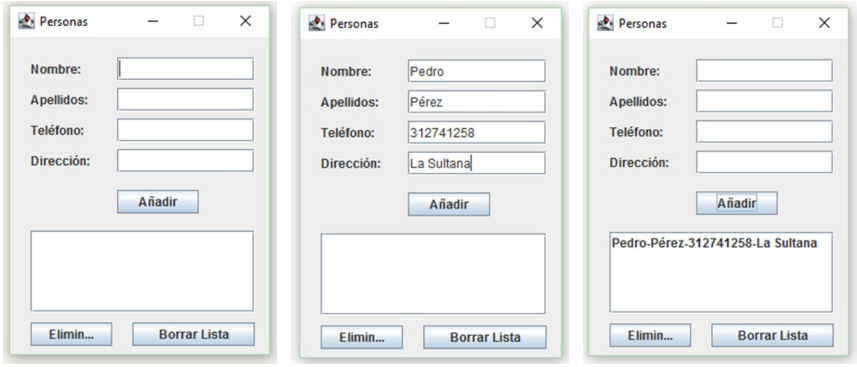


Figura 8.2. Diagrama de objetos del ejercicio 8.1.

Ejecución del programa



a) Ventana Principal b) Ingreso de persona c) Lista de personas

Figura 8.3. Ejecución del programa del ejercicio 8.1.

Ejercicios propuestos

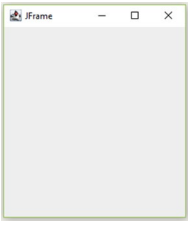
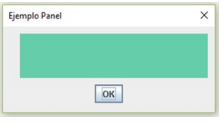
- Agregar la funcionalidad para editar los datos de una persona seleccionada de la lista de personas.
- Agregar la funcionalidad para generar mensajes de alerta cuando se ingrese una persona cuyos nombres y apellidos ya se encuentren en la lista.

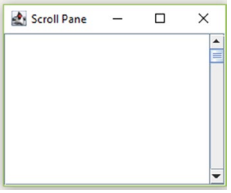
Ejercicio 8.2. Componentes *swing*

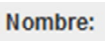
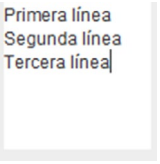
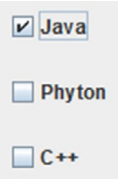
Un componente es el objeto fundamental de la interfaz de usuario en Java. Todo lo que ve en la interfaz en una aplicación Java es un componente. Para que este pueda ser utilizado debe colocarse en un contenedor. Todos los componentes *swing* se derivan de la clase abstracta *JComponent* (Schildt, 2018).

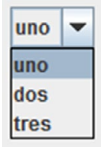
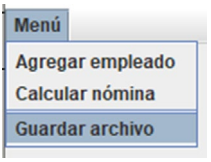
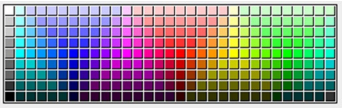
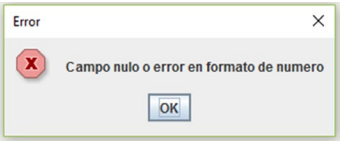
En la tabla 8.2 hay un listado abreviado de diferentes contenedores y componentes *swing* (Yang, 2018; API Java, 2020).

Tabla 8.2. Lista de contenedores y componentes *swing*

Contenedores de alto nivel	Descripción	Figura
JFrame	Clase para generar ventanas en las cuales se agregarán distintos objetos con los que podrá interactuar el usuario.	
JDialog	Clase para crear diálogos personalizados.	
Contenedores intermedios	**Descripción**	**Figura**
JPanel	Clase que representa un contenedor genérico.	

Contenedores de alto nivel	Descripción	Figura
Container	Contenedor genérico AWT.	
JScrollPane	Gestiona una ventana gráfica con barras de desplazamiento vertical y horizontal opcionales.	

Componentes	Descripción	Figura
JLabel	Un componente utilizado para mostrar una cadena corta de caracteres o un icono de imagen.	
JButton	Un componente para crear un botón etiquetado. La aplicación realiza algún tipo de acción cuando se pulsa el botón.	
JTextField	Un componente que implementa un texto editable de una solo línea.	
JTextArea	Un componente que implementa un texto editable multilínea.	
JCheckBox	Un componente que implementa una casilla de verificación: un elemento que se puede seleccionar o deseleccionar y que muestra su estado al usuario.	
JRadioButton	Un componente que implementa un botón de radio: un elemento que se puede seleccionar o deseleccionar y que muestra su estado al usuario.	

Contenedores de alto nivel	Descripción	Figura
JComboBox	Un componente que combina un campo editable y una lista desplegable. Se puede seleccionar un valor de la lista.	
JList	Un componente que muestra una lista de objetos y permite al usuario seleccionar uno o más elementos.	
JMenu	Implementación de un menú: una ventana emergente que contiene *JMenuItems*, estos se muestran cuando se selecciona un elemento en *JMenuBar*.	
JColorChooser	Proporciona un panel de control que permite a un usuario manipular y seleccionar un color.	
JOptionPane	Facilita la aparición de un cuadro de diálogo estándar que solicita a los usuarios un valor o informa algo.	
JToolBar	Proporciona un componente que es útil para mostrar controles de uso común.	
JSlider	Permite seleccionar un valor deslizando un indicador dentro de un rango.	
JSpinner	Un campo de entrada que permite seleccionar un número o un valor de una secuencia ordenada.	
JPasswordField	Un componente que permite editar una sola línea de texto se muestra que hay algo escrito, pero no muestra los caracteres originales.	
JTable	Un componente utilizado para mostrar y editar tablas de celdas bidimensionales regulares.	

Clases de soporte	Descripción	Figura
Graphics	Clase abstracta para los contextos gráficos que permiten que se dibuje imágenes en componentes fuera de la pantalla.	
Color	Clase para encapsular colores RGB o colores arbitrarios identificados por un *ColorSpace*.	
Font	Clase que representa las fuentes que se utilizan para representar el texto de forma visible.	

Cada componente posee una colección de atributos y métodos bastante extensa. Por cuestiones de espacio no se incluyen en este texto. Se recomienda referirse a la documentación API de Java en: https://docs.oracle.com/javase/7/docs/api/

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Conocer y aplicar diversos componentes *swing* para desarrollar interfaces gráficas de usuario.
- Realizar una gestión básica de los eventos generados mediante la interacción con los diferentes componentes *swing*.

Enunciado: Notas

Se requiere desarrollar un programa con interfaz gráfica de usuario que genere una ventana donde se solicite el ingreso de cinco notas obtenidas por un estudiante.

El programa debe calcular y mostrar en la parte inferior de la ventana los siguientes datos:

- El promedio de notas ingresadas.
- La desviación estándar de las notas ingresadas.
- La mayor nota obtenida.
- La menor nota obtenida.

Instrucciones Java del ejercicio

Tabla 8.3. Instrucciones Java del ejercicio 8.2.

Clase	Método	Descripción
Math	*double pow(double a, double b)*	Retorna el valor del primer argumento elevado a la potencia del segundo argumento.
	double sqrt(double a)	Retorna la raíz cuadrada positiva del valor a.
JFrame	*JFrame()*	Constructor de la clase *JFrame*.
	void setTitle(String título)	Establece el título de la ventana con el *String* especificado.
	void setSize(int x, int y)	Cambia el tamaño del componente para que tenga una anchura *x* y una altura *y*.
	void setLocationRelativeTo (Component c)	Establece la ubicación de la ventana en relación con el componente especificado.
	void setDefaultCloseOperation (opciones)	Usado para especificar una de las siguientes opciones del botón de cierre: EXIT_ON_CLOSE, HIDE_ON_CLOSE, DISPOSE_ON_CLOSE o DO_NOTHING_ ON_CLOSE.
	void setResizable(boolean resizable)	Para evitar que se cambie el tamaño de la ventana.
	void setVisible(boolean b)	Muestra u oculta la ventana según el valor del parámetro b.
ActionListener	*void actionPerformed(actionEvent e)*	Se invoca cuando ocurre un evento.
Container	*Container getContentPane()*	Retorna el objeto *ContentPane* de la ventana.
	void setLayout(LayoutManager mgr)	Establece el *layout* de la ventana.
	Component add(Component comp)	Añade el componente especificado al final del contenedor.
Component	*void addActionListener(this)*	Añade un oyente de eventos al componente actual.
	void setBounds(int x, int y, int ancho, int alto)	Mueve y cambia el tamaño del componente.
JLabel	*JLabel()*	Constructor de la clase *JLabel*.
	void setText(String text)	Define una línea de texto que mostrará este componente.
JTextField	*JTextField()*	Constructor de la clase *JTextField*.
	String getText()	Retorna el texto contenido en el componente de texto.

Clase	Método	Descripción
JButton	*JButton()*	Constructor de la clase *JButton*.
	void setText(String text)	Define una línea de texto que mostrará este componente.
Event	*Object getSource()*	El objeto sobre el cual el evento inicialmente ha ocurrido.

Solución

Clase: Notas

```
package Notas;
/**
 * Esta clase denominada Notas define un array de notas numéricas de
 * tipo double.
 * @version 1.2/2020
 */
public class Notas {
    double[] listaNotas; /* Atributo que identifica un array de notas de
        tipo double */

    /**
    * Constructor de la clase Notas, instancia un array con 5 notas de
    * tipo double
    */
    public Notas() {
        listaNotas = new double[5]; // Crea un array de 5 notas
    }

    /**
    * Método que calcula el promedio de notas
    * @return El promedio de notas calculado
    */
    double calcularPromedio() {
        double suma = 0;
        for(int i=1; i < listaNotas.length; i++) { // Se recorre el array
            suma = suma + listaNotas[i]; // Suma las notas del array
        }
        /* Obtiene el promedio como la división de la suma de notas
            sobre el total de notas */
        return (suma / listaNotas.length);
    }
```

```
/**
* Método que calcula la desviación estándar del array de notas
* @return La desviación estándar del array de notas
*/
double calcularDesviación() {
    double prom = calcularPromedio(); /* Invoca el método para
        calcular el promedio */
    double suma = 0;
    for(int i=0; i < listaNotas.length; i++) {
        // Aplica fórmula para la sumatoria de elementos
        suma += Math.pow(listaNotas[i] - prom, 2 );
    }
    return Math.sqrt (suma/listaNotas.length ); /* Retorna el cálculo
        final de la desviación */
}

/**
* Método que calcula el valor menor del array de notas
* @return El valor menor del array de notas
*/
double calcularMenor()  {
    double menor = listaNotas[0]; /* Define una variable como la
        nota menor */
    for(int i=0; i < listaNotas.length; i++) { // Se recorre el array
        if (listaNotas[i] < menor) {
            /* Si un elemento del array es menor que el menor actual,
                se actualiza su valor */
            menor = listaNotas[i];
        }
    }
    return menor;
}

/**
* Método que calcula el valor mayor del array de notas
* @return El valor mayor del array de notas
*/
double calcularMayor()  {
    double mayor = listaNotas[0]; /* Define una variable como la
        nota mayor */
    for(int i=0; i < listaNotas.length; i++) { // Se recorre el array
        if (listaNotas[i] > mayor) {
```

```
                    /* Si un elemento del array es mayor que el mayor actual,
                       se actualiza su valor */
                    mayor = listaNotas[i];
                }
            }
            return mayor;
        }
    }
```

Clase: VentanaPrincipal

```
package Notas;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;

/**
 * Esta clase denominada VentanaPrincipal define una interfaz gráfica
 * que permitirá crear un array de notas. Luego, se puede calcular el
 * promedio de notas, la desviación, la nota mayor y la nota menor del
 * array.
 * @version 1.2/2020
 */
public class VentanaPrincipal extends JFrame implements
ActionListener {
    // Un contenedor de elementos gráficos
    private Container contenedor;
    // Etiquetas estáticas de cada nota
    private JLabel nota1, nota2, nota3, nota4, nota5, promedio,
        desviación, mayor, menor;
    // Campos de ingreso de cada nota
    private JTextField campoNota1, campoNota2, campoNota3,
        campoNota4, campoNota5;
    // Botones para realizar cálculos y para borrar las notas
    private JButton calcular, limpiar;

    /**
    * Constructor de la clase VentanaPrincipal
    */
    public VentanaPrincipal(){
  inicio();
```

```
        setTitle("Notas"); // Establece el título de la ventana
        setSize(280,380); // Establece el tamaño de la ventana
        setLocationRelativeTo(null); /* La ventana se posiciona en el
            centro de la pantalla */
        // Establece que el botón de cerrar permitirá salir de la aplicación
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setResizable(false); /* Establece que el tamaño de la ventana no
            se puede cambiar */
    }

    /**
    * Método que crea la ventana con sus diferentes componentes
    * gráficos
    */
    private void inicio() {
        contenedor = getContentPane(); /* Obtiene el panel de
            contenidos de la ventana */
        contenedor.setLayout(null); /* Establece que el contenedor no
            tiene un layout */

        // Establece la etiqueta y el campo de texto de la nota 1
        nota1 = new JLabel();
        nota1.setText("Nota 1:");
        nota1.setBounds(20, 20, 135, 23); /* Establece la posición de la
            etiqueta nota 1 */
        campoNota1 = new JTextField();
        // Establece la posición del campo de texto de la nota 1
        campoNota1.setBounds(105, 20, 135, 23);

        // Establece la etiqueta y el campo de texto de la nota 2
        nota2 = new JLabel();
        nota2.setText("Nota 2:");
        nota2.setBounds(20, 50, 135, 23); /* Establece la posición de la
            etiqueta nota 2 */
        campoNota2 = new JTextField();
        // Establece la posición del campo de texto de la nota 2
        campoNota2.setBounds(105, 50, 135, 23);

        // Establece la etiqueta y el campo de texto de la nota 3
        nota3 = new JLabel();
        nota3.setText("Nota 3:");
        nota3.setBounds(20, 80, 135, 23); /* Establece la posición de la
            etiqueta nota 3*/
        campoNota3 = new JTextField();
```

```
// Establece la posición del campo de texto de la nota 3
campoNota3.setBounds(105, 80, 135, 23);

// Establece la etiqueta y el campo de texto de la nota 4
nota4 = new JLabel();
nota4.setText("Nota 4:");
nota4.setBounds(20, 110, 135, 23); /* Establece la posición de la
    etiqueta nota 4 */
campoNota4 = new JTextField();
// Establece la posición del campo de texto de la nota 4
campoNota4.setBounds(105, 110, 135, 23);

// Establece la etiqueta y el campo de texto de la nota 5
nota5 = new JLabel();
nota5.setText("Nota 5:");
nota5.setBounds(20, 140, 135, 23); /* Establece la posición de la
    etiqueta nota 5 */
campoNota5 = new JTextField();
// Establece la posición del campo de texto de la nota 5
campoNota5.setBounds(105, 140, 135, 23);

// Establece el botón Calcular
calcular = new JButton();
calcular.setText("Calcular");
calcular.setBounds(20, 170, 100, 23); /* Establece la posición
    del botón Calcular */
/* Agrega al botón un ActionListener para que gestione eventos
    del botón */
calcular.addActionListener(this);

// Establece el botón Limpiar
limpiar = new JButton();
limpiar.setText("Limpiar");
limpiar.setBounds(125, 170, 80, 23); /* Establece la posición del
    botón Limpiar */
/* Agrega al botón un ActionListener para que gestione eventos
    del botón */
limpiar.addActionListener(this);

// Establece la etiqueta del promedio con su valor numérico
promedio = new JLabel();
promedio.setText("Promedio = ");
promedio.setBounds(20, 210, 135, 23); /* Establece la posición
    de la etiqueta promedio */
```

```
        // Establece la etiqueta de la desviación con su valor numérico
        desviación = new JLabel();
        desviación.setText("Desviación = ");
        desviación.setBounds(20, 240, 200, 23); /* Establece la posición
            de la etiqueta desviación */

        // Establece la etiqueta de la nota mayor con su valor numérico
        mayor = new JLabel();
        mayor.setText("Nota mayor = ");
        mayor.setBounds(20, 270, 120, 23); /* Establece la posición de
            la etiqueta nota mayor */

        // Establece la etiqueta de la nota menor con su valor numérico
        menor = new JLabel();
        menor.setText("Nota menor = ");
        menor.setBounds(20, 300, 120, 23); /* Establece la posición de
            la etiquete nota menor */

        // Se añade cada componente gráfico al contenedor de la ventana
        contenedor.add(nota1);
        contenedor.add(campoNota1);
        contenedor.add(nota2);
        contenedor.add(campoNota2);
        contenedor.add(nota3);
        contenedor.add(campoNota3);
        contenedor.add(nota4);
        contenedor.add(campoNota4);
        contenedor.add(nota5);
        contenedor.add(campoNota5);
        contenedor.add(calcular);
        contenedor.add(limpiar);
        contenedor.add(promedio);
        contenedor.add(desviación);
        contenedor.add(mayor);
        contenedor.add(menor);
    }

    /**
    * Método que gestiona los eventos generados en la ventana principal
    */
    @Override
    public void actionPerformed(ActionEvent evento) {
```

```
        if (evento.getSource() == calcular) { /* Si se pulsa el botón
            Calcular */
            Notas notas = new Notas(); // Se crea un objeto Notas
            // Se obtiene y convierte el valor numérico de la nota 1
            notas.listaNotas[0] = Double.parseDouble(campoNota1.
                getText());
            // Se obtiene y convierte el valor numérico de la nota 2
            notas.listaNotas[1] = Double.parseDouble(campoNota2.
                getText());
            // Se obtiene y convierte el valor numérico de la nota 3
            notas.listaNotas[2] = Double.parseDouble(campoNota3.
                getText());
            // Se obtiene y convierte el valor numérico de la nota 4
            notas.listaNotas[3] = Double.parseDouble(campoNota4.
                getText());
            // Se obtiene y convierte el valor numérico de la nota 5
            notas.listaNotas[4] = Double.parseDouble(campoNota5.
                getText());
            notas.calcularPromedio(); // Se calcula el promedio
            notas.calcularDesviación(); // Se calcula la desviación
            // Se muestra el promedio formateado
            promedio.setText("Promedio = " + String.valueOf(String.
                format("%.2f",
                notas.calcularPromedio())));
            double desv = notas.calcularDesviación();
            // Se muestra la desviación formateada
            desviación.setText("Desviación estándar = " + String.
                format("%.2f", desv));
            // Se muestra el valor mayor formateado
            mayor.setText("Valor mayor = " + String.valueOf(notas.
                calcularMayor()));
            // Se muestra el valor menor formateado
            menor.setText("Valor menor = " + String.valueOf(notas.
                calcularMenor()));
        }
        /* Se se pulsa el botón Limpiar se dejan en blanco los campos de
            notas */
        if (evento.getSource() == limpiar) {
            campoNota1.setText("");
            campoNota2.setText("");
            campoNota3.setText("");
            campoNota4.setText("");
```

```
                campoNota5.setText("");
            }
        }
    }
```

Clase: Principal

```
package Notas;
/**
 * Esta clase define el punto de ingreso al programa de operaciones
 * sobre notas. Por lo tanto, cuenta con un método main de acceso al
 * programa.
 * @version 1.2/2020
 */
public class Principal {
    /**
     * Método main que sirve de punto de entrada al programa
     */
    public static void main(String[] args) {
        VentanaPrincipal miVentanaPrincipal; /* Define la ventana
            principal */
        miVentanaPrincipal= new VentanaPrincipal(); /* Crea la ventana
            principal */
        miVentanaPrincipal.setVisible(true); /* Establece la ventana
            como visible */
    }
}
```

Diagrama de clases

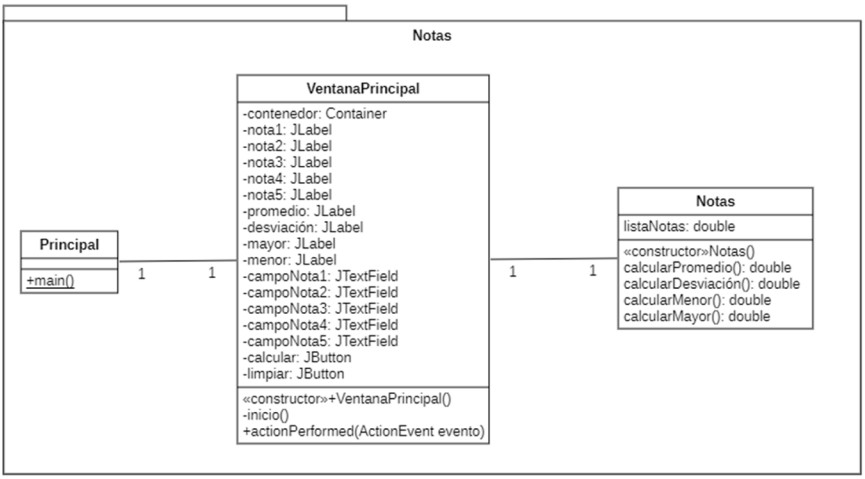


Figura 8.4. Diagrama de clases del ejercicio 8.2.

Explicación del diagrama de clases

Se ha definido un paquete denominado "Notas" que incluye un conjunto de clases. El punto de entrada al programa es la clase Principal que cuenta con el método *main*. La clase Principal está relacionada mediante una relación de asociación con la clase VentanaPrincipal que posee atributos privados para identificar los diferentes componentes gráficos de la ventana: un contenedor de componentes gráficos (*Container*), etiquetas (*JLabel*), campos de texto para ingreso de las notas (*JTextField*) y botones (*JButton*) para calcular diferentes métricas y borrar las notas. La clase VentanaPrincipal cuenta con un constructor y unos métodos para generar la ventana gráfica con sus componentes (inicio) y para gestionar los diferentes eventos surgidos al interactuar con esta ventana (*actionPerformed*).

La clase VentanaPrincipal se conecta por medio de una relación de asociación con la clase Notas. La relación de asociación tiene una multiplicidad de uno en cada extremo, lo cual indica que la VentanaPrincipal se relaciona con una única instancia de Notas y, a su vez, Notas se relaciona con una sola VentanaPrincipal.

La clase Notas tiene como un único atributo un *array* de notas de tipo *double*. Además, cuenta con un constructor y métodos para calcular la media, desviación estándar, la mayor nota y la menor nota.

Diagrama de objetos

Figura 8.5. Diagrama de objetos del ejercicio 8.2.

Ejecución del programa

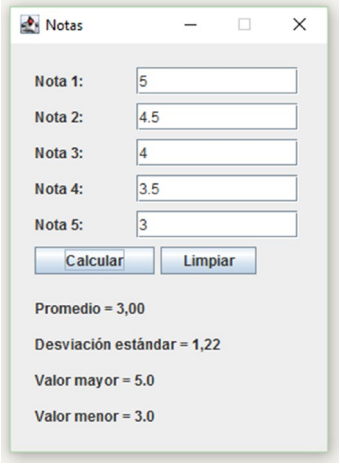


Figura 8.6. Ejecución del programa del ejercicio 8.2.

Ejercicios propuestos

- Modificar el programa del ejercicio para generar mensajes de alerta cuando se ingresen datos que no sean numéricos.
- Es obligatorio ingresar las cinco notas, se genera un mensaje de alerta cuando una nota no ha sido ingresada.

Ejercicio 8.3. Gestión de eventos

Los usuarios realizan acciones sobre componentes de la interfaz gráfica. La realización de estas acciones genera eventos que deben ser procesados por el programa. Los componentes de la interfaz gráfica "deben estar atentos" para gestionar los eventos generados. Para ello, tienen asociados unos objetos denominados oyentes o *listeners* (Morelli y Walde, 2012).

Los *listeners* son interfaces Java incluidos en el paquete que se deben importar:

java.awt.event.;*

Los componentes que procesan los eventos deben implementar la interfaz *ActionListener*, que incluye el método *actionPerformed(ActionEvent e)*, el cual debe ser implementado.

Los eventos pueden ser de bajo nivel o semánticos (Schildt, 2018):

- **Los eventos de bajo nivel:** representan entradas o interacciones de bajo nivel con la GUI. Por ejemplo, cambio de tamaño de un componente, de foco, etc.
- **Los eventos semánticos:** encapsulan la semántica del modelo del componente. Por ejemplo, cambiar el estado de un componente, seleccionar un ítem, etc.

Objetivo de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para definir oyentes o *listeners* para el tratamiento de eventos generados mediante la interacción de usuario en la interfaz gráfica.

Enunciado: Persona

Se requiere desarrollar un programa con interfaz gráfica de usuario que permita calcular el volumen y superficie de varias figuras geométricas. Las figuras geométricas son el cilindro, la esfera y la pirámide.

- Para el cilindro se solicitan su radio y altura (en centímetros).
- Para la esfera, su radio (en centímetros).
- Para la pirámide, su base, altura y apotema (en centímetros).

Una vez ingresados estos datos, el programa calcula el volumen y superficie de cada figura. Para desarrollar el programa se debe crear una jerarquía de clases para las diferentes figuras geométricas requeridas.

Instrucciones Java del ejercicio

Tabla 8.4. Instrucciones Java del ejercicio 8.3.

Clase	Método	Descripción
Math	*double PI*	Valor *double* que representa la constante Pi.
	double pow(double a, double b)	Retorna el valor del primer argumento elevado a la potencia del segundo argumento.
Double	*double parsedouble(Strings)*	Devuelve un *double* inicializado con el valor representado por el *String* especificado.
String	*String format(String formato, Object … args)*	Retorna un *String* formateado utilizando el formato y argumentos especificados.
JFrame	*JFrame()*	Constructor de la clase *JFrame*.
	void setTitle(String título)	Establece el título de la ventana con el *String* especificado.
	void setSize(int x, int y)	Cambia el tamaño del componente para que tenga una anchura *x* y una altura *y*.
	void setLocationRelativeTo (Component c)	Establece la ubicación de la ventana en relación con el componente especificado.
	void setDefaultCloseOperation (opciones)	Usado para especificar una de las siguientes opciones del botón de cierre: EXIT_ON_ CLOSE, HIDE_ON_CLOSE, DISPOSE_ON_ CLOSE o DO_NOTHING_ON_CLOSE.
	void setResizable(boolean resizable)	Para evitar que se cambie el tamaño de la ventana.
	void setVisible(boolean b)	Muestra u oculta la ventana según el valor del parámetro b.
ActionListener	*void actionPerformed(actionEvent e)*	Se invoca cuando ocurre un evento.
Container	*Container getContentPane()*	Retorna el objeto *ContentPane* de la ventana.
	void setLayout(LayoutManager mgr)	Establece el *layout* de la ventana.
	Component add(Component comp)	Añade el componente especificado al final del contenedor.
Component	*void addActionListener(this)*	Añade un oyente de eventos al componente actual.
	void setBounds(int x, int y, int ancho, int alto)	Mueve y cambia el tamaño del componente.
JLabel	*JLabel()*	Constructor de la clase *JLabel*.
	void setText(String text)	Define una línea de texto que mostrará este componente.

Clase	Método	Descripción
JTextField	*JTextField()*	Constructor de la clase *JTextField*.
	String getText()	Retorna el texto contenido en el componente de texto.
JButton	*JButton()*	Constructor de la clase *JButton*.
	void setText(String text)	Define una línea de texto que mostrará este componente.
	int getSelectedIndex()	Devuelve el índice seleccionado.
Event	*Object getSource()*	El objeto sobre el cual el evento inicialmente ha ocurrido.
JOptionPane	*void showMessageDialog (Component componentePadre, Object mensaje)*	Crea un cuadro de diálogo.

Solución

Clase: FiguraGeométrica

```
package Figuras;
/**
 * Esta clase denominada FiguraGeométrica modela un figura
 * geométrica que cuenta con un volumen y una superficie a ser
 * calculados de acuerdo al tipo de figura geométrica.
 * @version 1.2/2020
 */
public class FiguraGeométrica {
    private double volumen; /* Atributo que identifica el volumen de
        una figura geométrica */
    private double superficie; /* Atributo que identifica la superficie de
        una figura geométrica */

    /**
    * Método para establecer el volumen de una figura geométrica
    * @param volumen Parámetro que define el volumen de una figura
    * geométrica
    */
    public void setVolumen(double volumen) {
        this.volumen = volumen;
    }
```

```
    /**
     * Método para establecer la superficie de una figura geométrica
     * @param superficie Parámetro que define la superficie de una
     * figura geométrica
     */
    public void setSuperficie(double superficie) {
        this.superficie = superficie;
    }

    /**
     * Método para obtener el volumen de una figura geométrica
     * @return El volumen de una figura geométrica
     */
    public double getVolumen() {
        return this.volumen;
    }

    /**
     * Método para obtener la superficie de una figura geométrica
     * @return La superficie de una figura geométrica
     */
    public double getSuperficie() {
        return this.superficie;
    }
}
```

Clase: Cilindro

```
package Figuras;

/**
 * Esta clase denominada Cilindro es una subclase de FiguraGeométrica
 * que cuenta con un radio y una altura.
 * @version 1.2/2020
 */
public class Cilindro extends FiguraGeométrica {
    private double radio; // Atributo que establece el radio de un cilindro
    private double altura; // Atributo que establece la altura de un cilindro

    /**
     * Constructor de la clase Cilindro
     * @param radio Parámetro de define el radio de un cilindro
     * @param altura Parámetro de define la altura de un cilindro
     */
```

```
    public Cilindro(double radio, double altura) {
        this.radio = radio;
        this.altura = altura;
        this.setVolumen(calcularVolumen()); /* Calcula el volumen y
            establece su atributo */
        this.setSuperficie(calcularSuperficie()); /* Calcula la superficie y
            establece su atributo */
    }

    /**
    * Método para calcular el volumen de un cilindro
    * @return El volumen de un cilindro
    */
    public double calcularVolumen() {
        double volumen = Math.PI * altura * Math.pow(radio, 2.0);
        return volumen;
    }

    /**
    * Método para calcular la superficie de un cilindro
    * @return La superficie de un cilindro
    */
    public double calcularSuperficie() {
        double áreaLadoA = 2.0 * Math.PI * radio * altura;
        double áreaLadoB = 2.0 * Math.PI * Math.pow(radio, 2.0);
        return áreaLadoA + áreaLadoB;
    }
}
```

Clase: Esfera

```
package Figuras;

/**
 * Esta clase denominada Esfera es una subclase de FiguraGeométrica
 * que cuenta con un radio.
 * @version 1.2/2020
 */
public class Esfera extends FiguraGeométrica {
    private double radio; // Atributo que identifica el radio de una esfera

    /**
    * Constructor de la clase Esfera
    * @param radio Parámetro de define el radio de una esfera
```

```
     */
    public Esfera(double radio) {
        this.radio = radio;
        this.setVolumen(calcularVolumen()); /* Calcula el volumen y
            establece su atributo */
        this.setSuperficie(calcularSuperficie()); /* Calcula la superficie y
            establece su atributo */
    }

    /**
    * Método para calcular el volumen de una esfera
    * @return El volumen de una esfera
    */
    public double calcularVolumen() {
        double volumen = 1.333 * Math.PI * Math.pow(this.radio, 3.0);
        return volumen;
    }

    /**
    * Método para calcular la superficie de una esfera
    * @return La superficie de una esfera
    */
    public double calcularSuperficie() {
        double superficie = 4.0 * Math.PI * Math.pow(this.radio, 2.0);
        return superficie;
    }
}
```

Clase: Pirámide

```
package Figuras;

/**
 * Esta clase denominada Pirámide es una subclase de FiguraGeométrica
 * que cuenta con una base, una altura y un apotema.
 * @version 1.2/2020
 */
public class Piramide extends FiguraGeométrica {
    private double base; /* Atributo que identifica la base de una
        pirámide */
    private double altura; */ Atributo que identifica la altura de una
        pirámide */
```

```
    private double apotema; /* Atributo que identifica el apotema de
        una pirámide */

    /**
    * Constructor de la clase Pirámide
    * @param base Parámetro de define la base de una pirámide
    * @param altura Parámetro de define la altura de una pirámide
    * @param apotema Parámetro de define el apotema de una pirámide
    */
    public Piramide(double base, double altura, double apotema) {
        this.base = base;
        this.altura = altura;
        this.apotema = apotema;
        this.setVolumen(calcularVolumen()); /* Calcula el volumen y
            establece su atributo */
        this.setSuperficie(calcularSuperficie()); /* Calcula la superficie y
            establece su atributo */
    }

    /**
    * Método para calcular el volumen de una pirámide
    * @return El volumen de una pirámide
    */
    public double calcularVolumen() {
        double volumen = (Math.pow(base, 2.0) * altura) / 3.0;
        return volumen;
    }

    /**
    * Método para calcular la superficie de una pirámide
    * @return La superficie de una pirámide
    */
    public double calcularSuperficie() {
        double áreaBase = Math.pow(base, 2.0);
        double áreaLado = 2.0 * base * apotema;
        return áreaBase + áreaLado;
    }
}
```

Clase: VentanaCilindro

```
package Figuras;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

/**
 * Esta clase denominada VentanaCilindro define una ventana para
 * ingresar los datos de un cilindro y calcular su volumen y superficie.
 * @version 1.2/2020
 */
public class VentanaCilindro extends JFrame implements ActionListener {
    // Un contenedor de elementos gráficos
    private Container contenedor;
    /* Etiquetas estáticas para identificar los campos de texto a ingresar
       y calcular */
    private JLabel radio, altura, volumen, superficie;
    // Campos de texto a ingresar
    private JTextField campoRadio, campoAltura;
    // Botón para realizar los cálculos numéricos
    private JButton calcular;

    /**
    * Constructor de la clase VentanaCilindro
    */
    public VentanaCilindro() {
        inicio();
        setTitle("Cilindro"); // Establece el título de la ventana
        setSize(280,210); // Establece el tamaño de la ventana
        setLocationRelativeTo(null); /* La ventana se posiciona en el
            centro de la pantalla */
        setResizable(false); /* Establece que el botón de cerrar permitirá
            salir de la aplicación */
    }

    /**
    * Método que crea la ventana con sus diferentes componentes
    * gráficos
```

```
*/
private void inicio() {
    contenedor = getContentPane(); /* Obtiene el panel de
        contenidos de la ventana */
    contenedor.setLayout(null); /* Establece que el contenedor no
        tiene un layout */

    // Establece la etiqueta y campo de texto para el radio del cilindro
    radio = new JLabel();
    radio.setText("Radio (cms):");
    radio.setBounds(20, 20, 135, 23); /* Establece la posición de la
        etiqueta de radio del cilindro */
    campoRadio = new JTextField();
    // Establece la posición del campo de texto de radio del cilindro
    campoRadio.setBounds(100, 20, 135, 23);

    // Establece la etiqueta y campo de texto para la altura del cilindro
    altura = new JLabel();
    altura.setText("Altura (cms):");
    altura.setBounds(20, 50, 135, 23); /* Establece la posición de la
        etiqueta de altura del cilindro */
    campoAltura = new JTextField();
    // Establece la posición del campo de texto de altura del cilindro
    campoAltura.setBounds(100, 50, 135, 23);

    /* Establece el botón para calcular el volumen y superficie del
        cilindro */
    calcular = new JButton();
    calcular.setText("Calcular");
    calcular.setBounds(100, 80, 135, 23); /* Establece la posición
        del botón calcular */
    /* Agrega al botón un ActionListener para que gestione eventos
        del botón */
    calcular.addActionListener(this);

    // Establece la etiqueta y el valor del volumen del cilindro
    volumen = new JLabel();
    volumen.setText("Volumen (cm3):");
    // Establece la posición de la etiqueta de volumen del cilindro
    volumen.setBounds(20, 110, 135, 23);

    // Establece la etiqueta y el valor de la superficie del cilindro
```

```
        superficie = new JLabel();
        superficie.setText("Superficie (cm2):");
        // Establece la posición de la etiqueta de superficie del cilindro
        superficie.setBounds(20, 140, 135, 23);

        // Se añade cada componente gráfico al contenedor de la ventana
        contenedor.add(radio);
        contenedor.add(campoRadio);
        contenedor.add(altura);
        contenedor.add(campoAltura);
        contenedor.add(calcular);
        contenedor.add(volumen);
        contenedor.add(superficie);
}

    /**
    * Método que gestiona los eventos generados en la ventana del
    * cilindro throws Exception Excepción al ingresar un campo nulo o
    * error en formato de número
    */
    public void actionPerformed(ActionEvent event) {
        // Se inicializan el radio y la altura del cilindro
        boolean error = false; /* Se inicializa variable para determinar si
            ocurre un error */
        double radio = 0;
        double altura = 0;

        try {
            // Se obtiene el radio del cilindro ingresado
            radio  = Double.parseDouble(campoRadio.getText());
            // Se obtiene la altura del cilindro ingresada
            altura  = Double.parseDouble(campoAltura.getText());
            Cilindro cilindro = new Cilindro(radio, altura); /* Se crea un
                objeto Cilindro */
            // Se calcula y muestra el volumen
            volumen.setText("Volumen (cm3): " + String.format("%.2f",
                cilindro.calcularVolumen()));
            // Se calcula y muestra la superficie
            superficie.setText("Superficie (cm2): " + String.format("%.2f",
                cilindro.calcularSuperficie()));
        } catch (Exception e){
            error = true; // Si ocurre una excepción
```

```
            } finally {
                if(error) { /* Si ocurre una excepción, se muestra un mensaje
                    de error */
                    JOptionPane.showMessageDialog(null,"Campo nulo o
                        error en formato de numero",
                    "Error", JOptionPane.ERROR_MESSAGE);
                }
            }
        }
    }
```

Clase: VentanaEsfera

```
package Figuras;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

/**
 * Esta clase denominada VentanaEsfera define una ventana para
 * ingresar los datos  de una esfera y calcular su volumen y superficie.
 * @version 1.2/2020
 */
public class VentanaEsfera extends JFrame implements ActionListener {
    // Un contenedor de elementos gráficos
    private Container contenedor;
    /* Etiquetas estáticas para identificar los campos de texto a ingresar
        y calcular */
    private JLabel radio, volumen, superficie;
    private JTextField campoRadio; // Campo de texto a ingresar
    private JButton calcular; /* Botón para realizar los cálculos
        numéricos */

    /**
    * Constructor de la clase VentanaEsfera
    */
    public VentanaEsfera() {
        inicio();
        setTitle("Esfera"); // Establece el título de la ventana
```

```
        setSize(280,200); // Establece el tamaño de la ventana
        setLocationRelativeTo(null); /* La ventana se posiciona en el
            centro de la pantalla */
        setResizable(false); /* Establece que el botón de cerrar permitirá
            salir de la aplicación */
    }

    /**
    * Método que crea la ventana con sus diferentes componentes
    * gráficos
    */
    private void inicio() {
        contenedor = getContentPane(); /* Obtiene el panel de
            contenidos de la ventana */
        contenedor.setLayout(null); /* Establece que el contenedor no
            tiene un layout */

        // Establece la etiqueta y campo de texto para el radio de la esfera
        radio = new JLabel();
        radio.setText("Radio (cms):");
        radio.setBounds(20, 20, 135, 23); /* Establece la posición de la
            etiqueta de radio de la esfera */
        campoRadio = new JTextField();
        // Establece la posición del campo de texto de radio de la esfera
        campoRadio.setBounds(100, 20, 135, 23);

        /* Establece el botón para calcular el volumen y superficie de la
            esfera */
        calcular = new JButton();
        calcular.setText("Calcular");
        calcular.setBounds(100, 50, 135, 23); /* Establece la posición
            del botón calcular */
        /* Agrega al botón un ActionListener para que gestione eventos
            del botón */
        calcular.addActionListener(this);

        // Establece la etiqueta y el valor del volumen de la esfera
        volumen = new JLabel();
        volumen.setText("Volumen (cm3):");
        // Establece la posición de la etiqueta de volumen de la esfera
        volumen.setBounds(20, 90, 135, 23);

        // Establece la etiqueta y el valor de la superficie de la esfera
        superficie = new JLabel();
```

```
        superficie.setText("Superficie (cm2):");
        // Establece la posición de la etiqueta de superficie de la esfera
        superficie.setBounds(20, 120, 135, 23);

        // Se añade cada componente gráfico al contenedor de la ventana
        contenedor.add(radio);
        contenedor.add(campoRadio);
        contenedor.add(calcular);
        contenedor.add(volumen);
        contenedor.add(superficie);
    }

    /**
    * Método que gestiona los eventos generados en la ventana de la
    * esfera throws Exception Excepción al ingresar un campo nulo o
    * error en formato de número
    */
    public void actionPerformed(ActionEvent evento) {
        if (evento.getSource() == calcular) { /* Si se pulsa el botón
            Calcular */
            boolean error = false;
            try {
                // Se obtiene y convierte el valor numérico del radio
                double radio  = Double.parseDouble(campoRadio.
                    getText());
                Esfera esfera = new Esfera(radio); /* Se crea un objeto
                    Esfera */
                // Se muestra el volumen
                volumen.setText("Volumen (cm3): " + String.
                    format("%.2f", esfera.calcularVolumen()));
                // Se muestra la superficie
                superficie.setText("Superficie (cm2): " + String.
                    format("%.2f",
                esfera.calcularSuperficie()));
            } catch (Exception e) {
                error = true; // Si ocurre una excepción
            } finally {
                if(error) { /* Si ocurre una excepción, se muestra un
                    mensaje de error */
                JOptionPane.showMessageDialog(null,"Campo nulo o
                    error en formato de número",
```

```
                        "Error", JOptionPane.ERROR_MESSAGE);
                }
            }
        }
    }
}
```

Clase: VentanaPirámide

```
package Figuras;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

/**
 * Esta clase denominada VentanaPirámide define una ventana para
 * ingresar los datos de una pirámide y calcular su volumen y superficie.
 * @version 1.2/2020
 */
public class VentanaPirámide extends JFrame implements
ActionListener {
    // Un contenedor de elementos gráficos
    private Container contenedor;
    /* Etiquetas estáticas para identificar los campos de texto a ingresar
       y calcular */
    private JLabel base, altura, apotema, volumen, superficie;
    // Campos de texto a ingresar
    private JTextField campoBase, campoAltura, campoApotema;
    // Botón para realizar los cálculos numéricos
    private JButton calcular;

    /**
    * Constructor de la clase VentanaPirámide
    */
    public VentanaPirámide() {
        inicio();
        setTitle("Pirámide"); // Establece el título de la ventana
        setSize(280,240); // Establece el tamaño de la ventana
```

```
        setLocationRelativeTo(null); /* La ventana se posiciona en el
            centro de la pantalla */
        setResizable(false); /* Establece que el botón de cerrar permitirá
            salir de la aplicación */
    }

    /**
    * Método que crea la ventana con sus diferentes componentes
    * gráficos
    */
    private void inicio() {
        contenedor = getContentPane(); /* Obtiene el panel de
            contenidos de la ventana */
        contenedor.setLayout(null); /* Establece que el contenedor no
            tiene un layout */

        /* Establece la etiqueta y campo de texto para la base de la
            pirámide */
        base = new JLabel();
        base.setText("Base (cms):");
        // Establece la posición de la etiqueta de la base de la pirámide
        base.setBounds(20, 20, 135, 23);
        campoBase = new JTextField();
        /* Establece la posición del campo de texto de la base de la
            pirámide */
        campoBase.setBounds(120, 20, 135, 23);

        /* Establece la etiqueta y campo de texto para la altura de la
            pirámide */
        altura = new JLabel();
        altura.setText("Altura (cms):");
        // Establece la posición de la etiqueta de la altura de la pirámide
        altura.setBounds(20, 50, 135, 23);
        campoAltura = new JTextField();
        /* Establece la posición del campo de texto de la altura de la
            pirámide */
        campoAltura.setBounds(120, 50, 135, 23);

        /* Establece la etiqueta y campo de texto para el apotema de la
            pirámide */
        apotema = new JLabel();
        apotema.setText("Apotema (cms):");
        // Establece la posición de la etiqueta del apotema de la pirámide
        apotema.setBounds(20, 80, 135, 23);
```

```
        campoApotema = new JTextField();
        /* Establece la posición del campo de texto del apotema de la
           pirámide */
        campoApotema.setBounds(120, 80, 135, 23);

        /* Establece el botón para calcular volumen y superficie de la
           pirámide */
        calcular = new JButton();
        calcular.setText("Calcular");
        calcular.setBounds(120, 110, 135, 23); /* Establece la posición
           del botón calcular */
        /* Agrega al botón un ActionListener para que gestione eventos
           del botón */
        calcular.addActionListener(this);

        // Establece la etiqueta y el valor del volumen de la pirámide
        volumen = new JLabel();
        volumen.setText("Volumen (cm3):");
        // Establece la posición de la etiqueta de volumen de la pirámide
        volumen.setBounds(20, 140, 135, 23);

        // Establece la etiqueta y el valor de la superficie de la pirámide
        superficie = new JLabel();
        superficie.setText("Superficie (cm2):");
        // Establece la posición de la etiqueta de superficie de la pirámide
        superficie.setBounds(20, 170, 135, 23);

        // Se añade cada componente gráfico al contenedor de la ventana
        contenedor.add(base);
        contenedor.add(campoBase);
        contenedor.add(altura);
        contenedor.add(campoAltura);
        contenedor.add(apotema);
        contenedor.add(campoApotema);
        contenedor.add(calcular);
        contenedor.add(volumen);
        contenedor.add(superficie);
    }

    /**
    * Método que gestiona los eventos generados en la ventana de la
    * esfera throws Exception Excepción al ingresar un campo nulo o
    * error en formato de número
```

```
        */
        public void actionPerformed(ActionEvent event) {
            Piramide pirámide;
            boolean error = false;
            double base = 0;
            double altura = 0;
            double apotema = 0;
            try {
                // Se obtiene y convierte el valor numérico de la base
                base = Double.parseDouble(campoBase.getText());
                // Se obtiene y convierte el valor numérico de la altura
                altura = Double.parseDouble(campoAltura.getText());
                // Se obtiene y convierte el valor numérico del apotema
                apotema = Double.parseDouble(campoApotema.getText());
                // Se crea un objeto Pirámide
                pirámide = new Piramide(base, altura, apotema);
                // Se muestra el volumen
                volumen.setText("Volumen (cm3): " + String.format("%.2f",
                    pirámide.calcularVolumen()));
                // Se muestra la superficie
                superficie.setText("Superficie (cm2): " + String.format("%.2f",
                        pirámide.calcularSuperficie()));
            } catch (Exception e) {
                error = true; // Si ocurre una excepción
            } finally {
                if (error) { /* Si ocurre una excepción, se muestra un mensaje
                    de error */
                    JOptionPane.showMessageDialog(null, "Campo nulo o
                        error en formato de número",
                     "Error",JOptionPane.ERROR_MESSAGE);
                }
            }
        }
    }
```

Clase: VentanaPrincipal

```
package Figuras;

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
```

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

/**
 * Esta clase denominada VentanaPrincipal define una interfaz gráfica
 * que permitirá consultar un menú principal con tres figuras
 * geométricas.
 * @version 1.2/2020
 */
public class VentanaPrincipal extends JFrame implements
ActionListener {
    // Un contenedor de elementos gráficos
    private Container contenedor;
    // Botones para seleccionar una figura geométrica determinada
    private JButton cilindro, esfera, pirámide;

    /**
    * Constructor de la clase VentanaPrincipal
    */
    public VentanaPrincipal(){
        inicio();
        setTitle("Figuras"); // Establece el título de la ventana
        setSize(350,160); // Establece el tamaño de la ventana
        setLocationRelativeTo(null); /* La ventana se posiciona en el
            centro de la pantalla */
        // Establece que el botón de cerrar permitirá salir de la aplicación
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    /**
    * Método que crea la ventana con sus diferentes componentes
    * gráficos
    */
    private void inicio() {
        contenedor = getContentPane(); /* Obtiene el panel de
            contenidos de la ventana */
        contenedor.setLayout(null); /* Establece que el contenedor no
            tiene un layout */

        // Establece el botón del cilindro
        cilindro = new JButton();
        cilindro.setText("Cilindro");
        cilindro.setBounds(20, 50, 80, 23); /* Establece la posición del
            botón del cilindro */
```

```
        /* Agrega al botón un ActionListener para que gestione eventos
           del botón */
        cilindro.addActionListener(this);

        // Establece el botón de la esfera
        esfera = new JButton();
        esfera.setText("Esfera");
        esfera.setBounds(125, 50, 80, 23); /* Establece la posición del
           botón de la esfera */
        /* Agrega al botón un ActionListener para que gestione eventos
           del botón */
        esfera.addActionListener(this);

        // Establece el botón de la pirámide
        pirámide = new JButton();
        pirámide.setText("Pirámide");
        pirámide.setBounds(225, 50, 100, 23); /* Establece la posición
           del botón de la pirámide */
        /* Agrega al botón un ActionListener para que gestione eventos
           del botón */
        pirámide.addActionListener(this);

        // Se añade cada componente gráfico al contenedor de la ventana
        contenedor.add(cilindro);
        contenedor.add(esfera);
        contenedor.add(pirámide);
    }

    /**
    * Método que gestiona los eventos generados en la ventana principal
    */
    public void actionPerformed(ActionEvent evento) {
        if (evento.getSource() == esfera) { // Si se pulsa el botón esfera
            VentanaEsfera esfera = new VentanaEsfera(); /* Crea la
               ventana de la esfera */
            esfera.setVisible(true); /* Establece que se visualice la ventana
               de la esfera */
        }
        if (evento.getSource() == cilindro) { /* Si se pulsa el botón
           cilindro */
            VentanaCilindro cilindro = new VentanaCilindro(); /* Crea la
               ventana del cilindro */
            cilindro.setVisible(true); /* Establece que se visualice la
               ventana del cilindro */
```

```
        }
        if (evento.getSource() == pirámide) { /* Si se pulsa el botón
            pirámide */
            VentanaPirámide pirámide = new VentanaPirámide(); /* Crea
                la ventana de la pirámide */
            pirámide.setVisible(true); /* Establece que se visualice la
                ventana de la pirámide */
        }
    }
}
```

Clase: Principal

```
package Figuras;
/**
 * Esta clase define el punto de ingreso al programa de figuras
 * geométricas. Por lo tanto, cuenta con un método main de acceso al
 * programa.
 * @version 1.2/2020
 */
public class Principal {

    /**
     * Método main que sirve de punto de entrada al programa
     */
    public static void main(String[] args) {
        VentanaPrincipal miVentanaPrincipal; /* Define la ventana
            principal */
        miVentanaPrincipal= new VentanaPrincipal(); /* Crea la ventana
            principal */
        miVentanaPrincipal.setVisible(true); /* Establece la ventana
            como visible */
        // Establece que la ventana no puede cambiar su tamaño
        miVentanaPrincipal.setResizable(false);
    }
}
```

Diagrama de clases

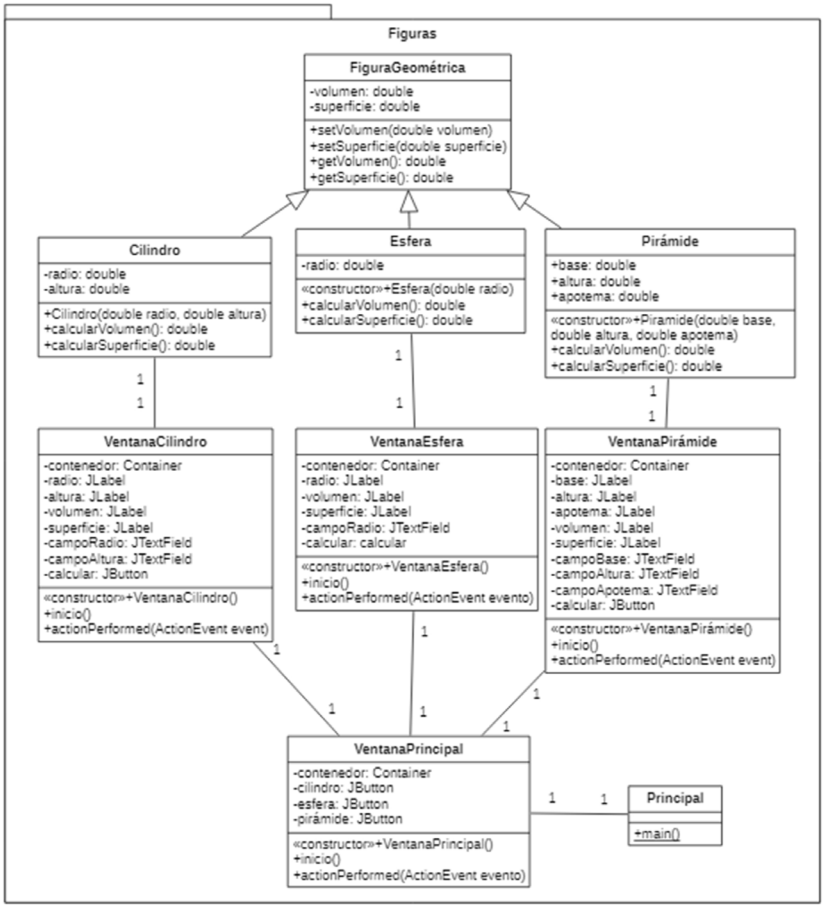


Figura 8.7. Diagrama de clases del ejercicio 8.3.

Explicación del diagrama de clases

Se ha definido un paquete denominado "Figuras" que incluye un conjunto de clases. El punto de entrada al programa es la clase Principal que cuenta con el método *main*. Mediante una relación de asociación la clase Principal está relacionada con la clase VentanaPrincipal que posee atributos privados para identificar tres botones (*JButton*) que abrirán las ventanas respectivas de las figuras geométricas: cilindro, esfera y pirámide. La clase VentanaPrincipal

cuenta con un constructor y métodos para generar la ventana gráfica con sus componentes (inicio) y gestionar los diferentes eventos surgidos al interactuar con esta ventana (*actionPerformed*).

La clase VentanaPrincipal está conectada por medio de relaciones de asociación con las clases VentanaCilindro, VentanaEsfera y VentanaPirámide. Las multiplicidades de estas relaciones son una en cada extremo, lo que indica que se creará una sola ventana para cada tipo de figura geométrica. Las clases VentanaCilindro, VentanaEsfera y VentanaPirámide cuentan con atributos para modelar los diferentes componentes gráficos de la ventana: un contenedor de componentes gráficos (*Container*), las etiquetas de los campos de texto y valores a calcular (*JLabel*), campos de texto para ingresar datos (*JTextField*) y un botón para calcular el volumen y superficie de las figuras (*JButton*).

Así, en primer lugar, la clase VentanaCilindro se conecta con la clase Cilindro por medio de una relación de asociación que tiene una multiplicidad de uno en cada extremo. Por lo tanto, la VentanaCilindro creará un único objeto Cilindro. Esta clase Cilindro cuenta con dos atributos privados que identifican el radio y altura del cilindro, con el constructor respectivo que inicializa estos atributos y métodos para calcular la superficie y volumen del cilindro.

En segundo lugar, la clase VentanaEsfera se conecta con la clase Esfera utilizando una relación de asociación con multiplicidad de uno en cada extremo; la VentanaEsfera creará un único objeto Esfera y este estará asociado a una única VentanaEsfera. La clase Esfera cuenta con un único atributo privado para identificar el radio de la esfera, con un constructor que inicializa el atributo y métodos para calcular la superficie y volumen de la esfera.

En tercer lugar, la clase VentanaPirámide se relaciona con la clase Pirámide por medio de una asociación con multiplicidad 1 en cada extremo. La clase Pirámide define tres atributos privados para identificar la base, altura y apotema de la pirámide y tiene con un constructor que inicializa sus atributos y métodos para calcular la superficie y volumen de la pirámide.

Finalmente, las clases Cilindro, Esfera y Pirámide son subclases de FiguraGeométrica, la cual es la clase padre que define atributos compartidos por las tres subclases: volumen y superficie, y posee métodos *get* y *set* para estos atributos.

Diagrama de objetos

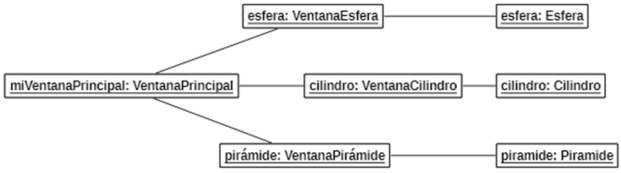
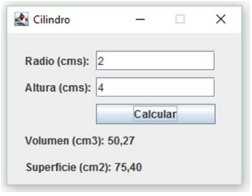


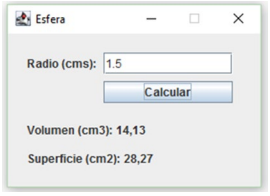
Figura 8.8. Diagrama de objetos del ejercicio 8.3.

Ejecución del programa

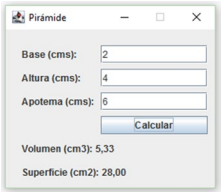
a) Ventana Principal



b) Cilindro



c) Esfera



d) Pirámide

Figura 8.9. Ejecución del programa del ejercicio 8.3.

Ejercicios propuestos

- Agregar el cálculo del volumen y superficie de otras figuras geométricas como el cubo y el prisma.
- Agregar a cada figura geométrica correspondiente su imagen correspondiente.

Ejercicio 8.4. Cuadros de diálogo

La clase *JDialog* es la clase que implementa cuadros de diálogo en *swing* (Schildt, 2018). Generalmente, dependen de una ventana principal (*JFrame*). Las ventanas y cuadros de diálogo pueden ser modales o no modales (Deitel y Deitel, 2017).

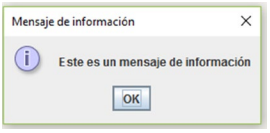
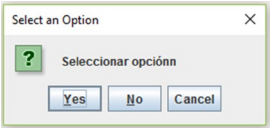
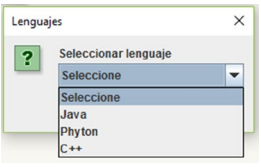
- **Cuadros de diálogo modales**: cuando un cuadro de diálogo modal es visible bloquea la entrada del usuario a todas las demás ventanas del programa. La clase *JOptionPane* crea *JDialogs* que son modales.
- **Cuadros de diálogo no modales**: permite el acceso a otras ventanas de la aplicación sin cerrar el cuadro de diálogo.

En el código, el tercer parámetro de la definición del cuadro de diálogo, es *boolean* e identifica si es un cuadro de diálogo modal o no.

JDialog miDiálogo = new JDialog(JFrame, tituloCuadroDiálogo, esModal);

La clase *JOptionPane* permite crear nuevos cuadros de diálogo o utilizar los más comunes. Los diferentes tipos de cuadros de diálogo se presentan en la tabla 8.5 (API Java, 2020).

Tabla 8.5. Tipos de cuadros de diálogo

Tipo/descripción	Código	Figura
Mensaje: para informar al usuario sobre algún hecho relevante. Pueden ser mensajes de información, advertencia, error o mensaje plano.	*JOptionPane.showMessage(JFrame, mensaje, tituloCuadro, tipo)* *Tipos:* • *JOptionPane.INFORMATION_MESSAGE* • *JOptionPane.WARNING_MESSAGE* • *JOptionPane.ERROR_MESSAGE* • *JOptionPane.PLAIN_MESSAGE*	
Confirmación: para realizar una pregunta al usuario con las posibles respuestas: sí, no o cancelar.	*JOptionPane.showConfirmDialog(JFrame, mensaje, tituloCuadro, tipo)* *Tipos:* • *JOptionPane.YES_NO_OPTION* • *JOptionPane.YES_NO_CANCEL_OPTION*	
Entrada: para solicitar una entrada del usuario.	*JOptionPane.showInputDialog(JFrame, mensaje)*	

Objetivo de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para generar diferentes cuadros de diálogo que faciliten la interacción de usuario con el programa.

Enunciado: Nómina

Se desea desarrollar un programa utilizando una interfaz gráfica de usuario que calcule la nómina de empleados de una empresa.

Para ello, el programa debe contar con una barra de menús con los siguientes ítems:

- Agregar empleado: genera una ventana donde se deben ingresar los datos de un empleado:
  - Nombre de tipo *String*.
  - Apellidos de tipo *String*.
  - Cargo, el cual puede ser directivo, estratégico u operativo, se puede implementar con un *JList*.
  - Género, el cual puede ser masculino o femenino, se puede implementar con un *JCheckBox*.
  - Salario por día de tipo *double*.
  - Días trabajados al mes desde 1 a 31, se puede implementar con un *JSpinner*.
  - Otros ingresos de tipo *double*.
  - Pagos por salud de tipo *double*.
  - Aporte pensiones de tipo *double*.

- Calcular nómina: genera una ventana donde se muestra en formato tabla los datos de los empleados ingresados. Cada fila de la tabla corresponde a un empleado. En las columnas se mostrarán los nombres, apellidos y sueldo de cada empleado. El sueldo de cada empleado se calcula como:

  *Salario mensual = (días trabajados * sueldo por día) + otros ingresos - pagos por salud - aporte pensiones*

  En la parte inferior de la ventana, se calcula el total de la nómina de la empresa, calculada como la suma de los salarios mensuales de cada empleado.

- Guardar archivo: solicita una carpeta donde se va a generar un archivo de texto denominado "*Nómina.txt*" con los datos ingresados por cada empleado, su sueldo mensual calculado y el total de la nómina de la empresa.

Instrucciones Java del ejercicio

Tabla 8.6. Instrucciones Java del ejercicio 8.4.

Clase	Método	Descripción
JFrame	*JFrame()*	Constructor de la clase *JFrame*.
	void setTitle(String título)	Establece el título de la ventana con el *String* especificado.
	void setSize(int x, int y)	Cambia el tamaño del componente para que tenga una anchura *x* y una altura *y*.
	void setLocationRelativeTo (Component c)	Establece la ubicación de la ventana en relación con el componente especificado.
	void setDefaultCloseOperation (opciones)	Usado para especificar una de las siguientes opciones del botón de cierre: EXIT_ON_CLOSE, HIDE_ON_CLOSE, DISPOSE_ ON_CLOSE o DO_NOTHING_ ON_CLOSE.
	void setResizable(boolean resizable)	Para evitar que se cambie el tamaño de la ventana.
	void setVisible(boolean b)	Muestra u oculta la ventana según el valor del parámetro b.
	void setJMenuBar(JMenuBar menubar)	Establece una barra de menú para la ventana.
ActionListener	*void actionPerformed(actionEvent e)*	Se invoca cuando ocurre un evento.
Container	*Container getContentPane()*	Retorna el objeto *ContentPane* de la ventana.
	void setLayout(LayoutManager mgr)	Establece el *layout* de la ventana.
	Component add(Component comp)	Añade el componente especificado al final del contenedor.
Component	*void addActionListener(this)*	Añade un oyente de eventos al componente actual.
	void setBounds(int x, int y, int ancho, int alto)	Mueve y cambia el tamaño del componente.
JLabel	*JLabel()*	Constructor de la clase *JLabel*.
	void setText(String text)	Define una línea de texto que mostrará este componente.

Clase	Método	Descripción
JTextField	*JTextField()*	Constructor de la clase *JTextField*.
	String getText()	Retorna el texto contenido en el componente de texto.
JButton	*JButton()*	Constructor de la clase *JButton*.
	void setText(String text)	Define una línea de texto que mostrará este componente.
JMenuBar	*JMenuBar()*	Constructor de la clase *JMenuBar*.
	JMenu add(JMenu c)	Añade el menú especificado al final de la barra de menú.
JMenu	*JMenu()*	Constructor de la clase *JMenu*.
	JMenuItem add(JMenuItem menuItem)	Añade un ítem de menú al final del menú.
JMenuItem	*JMenuItem()*	Constructor de la clase *JMenuItem*.
JSeparator	*JSeparator()*	Constructor de la clase *JSeparator*.
ButtonGroup	*ButtonGroup()*	Constructor de la clase *ButtonGroup*.
	void add(AbstractButton b)	Añade un botón al grupo.
JRadioButton	*JRadioButton(String texto, boolean selected)*	Constructor de un botón de radio con el texto especificado y el estado de selección.
JComboBox	*JComboBox()*	Constructor de la clase *JComboBox*.
JSpinner	*void setModel(SpinnerModel modelo)*	Cambia el modelo que representa el valor del *spinner*.
SpinnerNumberModel	*void setMinimum(Comparable mínimo)*	Cambia el mínimo inferior en la secuencia del *spinner*.
	void setMaximum(Comparable máximo)	Cambia el máximo superior en la secuencia del *spinner*.
	void setValue(Object valor)	Establece el valor actual en esta secuencia del *spinner*.
DefaultTableModel	*DefaultTableModel()*	Constructor de la clase *DefaultTableModel*.
JTable	*JTable(Object[][] datosFila, Object[] nombresColumnas)*	Constructor de *JTable* que presenta los valores en un *array* de dos dimensiones (datosFila) con los nombres de las columnas (nombresColomunas).

Clase	Método	Descripción
JFileChooser	*void setFileSelectionMode(int modo)*	Establece el selector de archivo, de directorio o ambos.
	showOpenDialog(Component padre)	Abre un diálogo de selección de archivo.
	File getSelectedFile()	Retorna el archivo seleccionado.
File	*String getName()*	Retorna el nombre del archivo o directorio.
	boolean createNewFile()	Crea un nuevo archivo vacío.
FileWriter	*FileWriter(File archivo)*	Construye un objeto *FileWriter* a partir de un objeto *File*.
BufferedWriter	*BufferedWriter(Writer salida)*	Construye un flujo de caracteres de salida que utiliza un *búfer* de salida.
	void write(Strings)	Escribe un *String* en el archivo.
	void close()	Cierra el flujo.
Event	*Object getSource()*	El objeto sobre el cual el evento inicialmente ha ocurrido.
JOptionPane	*void showMessageDialog (Component componentePadre, Object mensaje)*	Crea un cuadro de diálogo.
String	*String format(String formato, Object … args)*	Retorna un *String* formateado utilizando el formato y argumentos especificados.
Vector	*Object elementAt(int índice)*	Retorna el elemento en la posición especificada.
Double	*String toString()*	Retorna una representación *String* para el objeto *double*.

Solución

Dato enumerado: TipoCargo

```
package Nómina;
/**
 * Dato enumerado que especifica el tipo de cargo que tiene un empleado
 */
public enum TipoCargo {
    DIRECTIVO, ESTRATÉGICO, OPERATIVO
}
```

Dato enumerado: TipoGénero

```
package Nómina;

/**
 * Dato enumerado que especifica el tipo de género que tiene un empleado
 */
public enum TipoGénero {
    MASCULINO, FEMENINO
}
```

Clase: Empleado

```
package Nómina;

/**
 * Esta clase denominada Empleado modela un empleado de una
 * empresa que tiene como atributos su nombre, apellidos, género,
 * cargo, salario por día, otros ingresos, pagos por salud, aporte
 * pensiones y días trabajados.
 * @version 1.2/2020
 */
public class Empleado {
    private String nombre; /* Atributo que identifica el nombre de un
        empleado */
    private String apellidos; /* Atributo que identifica los apellidos de
        un empleado */
    private double salarioDía; /* Atributo que identifica el salario diario
        de un empleado */
    private double otrosIngresos; /* Atributo que identifica otros
        ingresos de un empleado */
    private double pagosSalud; /* Atributo que identifica los pagos por
        salud de un empleado */
    // Atributo que identifica el aporte de pensiones de un empleado
    private double aportePensiones;
    /* Atributo que identifica la cantidad de días trabajados por un
        empleado */
    private int díasTrabajados;
    private TipoCargo cargo; /* Atributo que identifica el cargo de un
        empleado */
```

```
private TipoGénero género; /* Atributo que identifica el gérero de
    un empleado */

/**
* Constructor de la clase Empleado
* @param nombre Parámetro que define el nombre de un empleado
* @param apellidos Parámetro que define los apellidos de un
* empleado
* @param cargo Parámetro que define el cargo de un empleado
* @param género Parámetro que define el género de un empleado
* @param salarioDía Parámetro que define el salario por día de un
* empleado
* @param díasTrabajados Parámetro que define la cantidad de días
* trabajados de un empleado
* @param pagosSalud Parámetro que define los pagos por salud de
* un empleado
* @param aportePensiones Parámetro que define el aporte de
* pensiones de un empleado
*/
public Empleado(String nombre, String apellidos, TipoCargo cargo,
    TipoGénero género, double salarioDía, int díasTrabajados,
    double otrosIngresos, double pagosSalud,
        double aportePensiones) {
    this.nombre = nombre;
    this.apellidos = apellidos;
    this.cargo = cargo;
    this.género = género;
    this.salarioDía = salarioDía;
    this.díasTrabajados = díasTrabajados;
    this.otrosIngresos = otrosIngresos;
    this.pagosSalud = pagosSalud;
    this.aportePensiones = aportePensiones;
}

/**
* Método que obtiene el nombre de un empleado
* @return El nombre de un empleado
*/
public String getNombre() {
    return nombre;
}
```

```
/**
* Método que obtiene los apellidos de un empleado
* @return Los apellidos de un empleado
*/
public String getApellidos() {
    return apellidos;
}

/**
* Método que obtiene el tipo de cargo de un empleado
* @return El tipo de cargo de un empleado
*/
public TipoCargo getCargo() {
    return cargo;
}

/**
* Método que obtiene el género de un empleado
* @return El género de un empleado
*/
public TipoGénero getGénero() {
    return género;
}

/**
* Método que obtiene el salario por día de un empleado
* @return El salario por día de un empleado
*/
public double getSalarioDía() {
    return salarioDía;
}

/**
* Método que obtiene los días trabajados al mes de un empleado
* @return Días trabajados al mes de un empleado
*/
public int getDíasTrabajados() {
    return díasTrabajados;
}

/**
* Método que obtiene otros ingresos de un empleado
* @return Otros ingresos de un empleado
*/
```

```
    public double getOtrosIngresos() {
        return otrosIngresos;
    }

    /**
    * Método que obtiene los pagos por salud de un empleado
    * @return Pagos por salud de un empleado
    */
    public double getPagosSalud() {
        return pagosSalud;
    }

    /**
    * Método que obtiene el aporte de pensiones de un empleado
    * @return Aporte de pensiones de un empleado
    */
    public double getAportePensiones() {
        return aportePensiones;
    }

    /**
    * Método que calcula el salario mensual de un empleado
    * @return Salario mensual de un empleado
    */
    public double calcularNómina() {
        return ((salarioDía * díasTrabajados) + otrosIngresos -
            pagosSalud - aportePensiones);
    }
}
```

Clase: ListaEmpleados

```
package Nómina;
import java.util.*;

/**
 * Esta clase denominada ListaEmpleados define un vector de objetos
 * Empleado y un total de la nómina de empleados.
 * @version 1.2/2020
 */
public class ListaEmpleados {
    public Vector lista; // Atributo que identifica un vector de empleados
```

```
public double totalNómina = 0; /* Atributo que identifica el total de
    la nómina de la empresa */

/**
* Constructor de la clase ListaEmpleados
*/
public ListaEmpleados() {
    lista = new Vector(); // Crea el vector de empleados
}

/**
* Método que agrega un empleado a la lista de empleados
* @param a Parámetro que define un empleado a agregar a la lista
* de empleados
*/
public void agregarEmpleado(Empleado a) {
    lista.add(a);
}

/**
* Método que calcula la nómina total mensual de la empresa
* @return La nómina total mensual de la empresa
*/
public double calcularTotalNómina() {
    for (int i = 0; i < lista.size(); i++) { /* Recorre el vector de
        empleados */
        // Obtiene un elemento de la lista de empleados
        Empleado e  = (Empleado) lista.elementAt(i);
        // Calcula el salario de un empleado y lo totaliza
        totalNómina = totalNómina + e.calcularNómina();
    }
    return totalNómina;
}

/**
* Método que convierte los datos de la lista de empleados en una
* matriz
*/
public String[][] obtenerMatriz() {
    String datos[][] = new String[lista.size()][3]; // Se crea la matriz
    for (int i = 0; i < lista.size(); i++) { // Recorre el vector de empleados
        // Obtiene un elemento de la lista de empleados
        Empleado e  = (Empleado) lista.elementAt(i);
```

```
            /* Coloca el nombre del empleado en la primera columna de
               la matriz */
            datos[i][0] = e.getNombre();
            /* Coloca los apellidos del empleado en la segunda columna
               de la matriz */
            datos[i][1] = e.getApellidos();
            /* Coloca el salario del empleado en la tercera columna de la
               matriz */
            datos[i][2] = Double.toString(e.calcularNómina());
            // Va acumulando el total de nómina mensual de la empresa
            totalNómina = totalNómina + e.calcularNómina();
        }
        return datos;
    }

    /**
    * Método que convierte los datos de la lista de empleados a texto
    */
    public String convertirTexto() {
        String texto = "";
        for (int i = 0; i < lista.size(); i++) { // Recorre el vector de empleados
            // Obtiene un elemento de la lista de empleados
            Empleado e  = (Empleado) lista.elementAt(i);
            // Concatena en una variable String los datos de un empleado
            texto = texto + "Nombre = " + e.getNombre() + "\n" +
                "Apellidos = " + e.getApellidos() + "\n" + "Cargo = " +
                e.getCargo() + "\n" + "Género = " + e.getGénero() + "\n" +
                "Salario = $" + e.getSalarioDía() + "\n" + "Días trabajados
                = " + e.getDíasTrabajados() + "\n" + "Otros ingresos =
                $" + e.getOtrosIngresos() + "\n" + "Pagos saludo = $"
                + e.getPagosSalud() + "\n" + "Aportes pensiones = $" +
                e.getAportePensiones() + "\n---------\n";
        }
        // Concatena en una variable String el total de la nómina
        texto = texto + "Total nómina = $" + String.format("%.2f",
            calcularTotalNómina());
        return texto;
    }
}
```

Clase: VentanaPrincipal

```
package Nómina;

import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.io.*;

/**
 * Esta clase denominada VentanaPrincipal define una interfaz gráfica
 * que permitirá generar la nómina de empleados.
 * @version 1.2/2020
 */
public class VentanaPrincipal extends JFrame implements
ActionListener {
    private Container contenedor;  // Un contenedor de elementos gráficos
    ListaEmpleados empleados; // Un vector de empleados
    private JMenuBar barraMenu; // Una barra de menú principal
    private JMenu menuOpciones; /* Un menú de la barra de menú
        principal */
    private JMenuItem itemMenu1; // Un ítem de menú
    private JMenuItem itemMenu2; // Un ítem de menú
    private JMenuItem itemMenu3; // Un ítem de menú

    /**
    * Constructor de la clase VentanaPrincipal
    */
    public VentanaPrincipal(){
        empleados = new ListaEmpleados(); // Se crea la lista de empleados
        inicio();
        setTitle("Nómina"); // Establece el título de la ventana
        setSize(280,380); // Establece el tamaño de la ventana
        setLocationRelativeTo(null); /* La ventana se posiciona en el
            centro de la pantalla */
        // Establece que el botón de cerrar permitirá salir de la aplicación
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setResizable(false); /* Establece que el tamaño de la ventana no
            puede cambiar */
    }
```

```
    }

    /**
    * Método que crea la ventana con sus diferentes componentes gráficos
    */
    public void inicio() {
        contenedor = getContentPane(); /* Obtiene el panel de
            contenidos de la ventana */
        contenedor.setLayout(null); /* Establece que el contenedor no
            tiene un layout */

        // Se crea la barra de menús, un menú y tres ítems de menú
        barraMenu = new JMenuBar();
        menuOpciones = new JMenu("Menú");
        itemMenu1 = new JMenuItem("Agregar empleado");
        itemMenu2 = new JMenuItem("Calcular nómina");
        itemMenu3 = new JMenuItem("Guardar archivo");
        menuOpciones.add(itemMenu1); /* Se agrega el ítem de menú 1
            al menú */
        menuOpciones.add(itemMenu2); /* Se agrega el ítem de menú 2
            al menú */
        menuOpciones.add(new JSeparator()); /* Se agrega una línea
            divisoria */
        menuOpciones.add(itemMenu3); /* Se agrega el ítem de menú 3
            al menú */
        barraMenu.add(menuOpciones); /* Se agregan las opciones de
            menú al menú */
        setJMenuBar(barraMenu); // Se agrega el menú a la ventana
        /* Agrega al ítem de menú un ActionListener para que gestione
            eventos del ítem de menú */
        itemMenu1.addActionListener(this);
        itemMenu2.addActionListener(this);
        itemMenu3.addActionListener(this);
    }

    /**
    * Método que gestiona los eventos generados en la ventana principal
    * throws Exception Excepción en la creación o escritura del archivo
    */
    @Override
    public void actionPerformed(ActionEvent evento) {
        if (evento.getSource() == itemMenu1) { /* Se selecciona el ítem
            de menú 1 */
```

```
            // Se crea la ventana de agregar empleado
            VentanaAgregarEmpleado ventanaAgregar= new
                VentanaAgregarEmpleado(empleados);
            ventanaAgregar.setVisible(true); // Se hace visible la ventana
        }
        if (evento.getSource() == itemMenu2) { /* Se selecciona el ítem
            de menú 2 */
            // Se crea la ventana de nómina
            VentanaNómina ventanaNómina = new
                VentanaNómina(empleados);
            ventanaNómina.setVisible(true); // Se hace visible la ventana
        }
        if (evento.getSource() == itemMenu3) { /* Se selecciona el ítem
            de menú 3 */
            JFileChooser fc = new JFileChooser(); /* Crea un selector de
                archivo */
            fc.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
                // Es un selector de directorio
            int respuesta = fc.showOpenDialog(this); /* Se muestra el
                selector de directorio en pantalla */
            if (respuesta == JFileChooser.APPROVE_OPTION) { /* Si se
                pulsa aceptar en el selector */
                File directorioElegido = fc.getSelectedFile(); /* Se obtiene
                    el directorio seleccionado */
                String nombre = directorioElegido.getName(); /* Se
                    obtiene el nombre del directorio */
                try {
                    // Convierte los datos de los empleados en texto
                    String contenido = empleados.convertirTexto();
                    // Se asigna el nombre del archivo de texto
                    File file = new File(nombre + "\\" + "Nómina.txt");
                    file.createNewFile(); // Se crea el archivo de texto
                    FileWriter fw = new FileWriter(file);
                    BufferedWriter bw = new BufferedWriter(fw); /* Se
                        crea el flujo de escritura de datos */
                    bw.write(contenido); /* Se escriben los datos en el
                        archivo */
                    bw.close(); // Se cierra el archivo
                    String texto = "El archivo de la nómina Nómina.txt se
                        ha creado en " + nombre;
                    // Mensaje de confirmación
```

```
                    JOptionPane.showMessageDialog(this, texto,
                        "Mensaje",
                JOptionPane.INFORMATION_MESSAGE,null);
                    } catch (Exception e) {
                        /* En caso que se presente una excepción en la
                            creación y escritura del archivo */
                        e.printStackTrace();
                    }
                }
            }
        }
}
```

Clase: VentanaAgregarEmpleado

```
package Nómina;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

/**
 * Esta clase denominada VentanaAgregarEmpleado define una ventana
 * para ingresar los datos de un empleado y agregarlo a la lista de
 * empleados.
 * @version 1.2/2020
 */
public class VentanaAgregarEmpleado extends JFrame implements
ActionListener {
    private Container contenedor; /* Un contenedor de elementos
        gráficos */
    private ListaEmpleados lista; // La lista de empleados
    // Etiquetas estáticas para indicar los datos a ingresar
    private JLabel nombre, apellidos, cargo, salarioDía, númeroDías,
        género, otrosIngresos, aportesSalud, pensiones;
    // Campos de texto a ingresar de un empleado
    private JTextField campoNombre, campoApellidos,
        campoSalarioDía, campoOtrosIngresos, campoAportesSalud,
        campoPensiones;
```

```
private ButtonGroup grupoGénero; // Grupo de botones de radio
private JRadioButton masculino, femenino; // Botones de radios
private JComboBox campoCargo; // Un combox
private JSpinner campoNúmeroDías; // Un selector de datos numérico
private SpinnerNumberModel modeloSpinner; /* Modelo numérico
    para el selector numérico */
/* Botones para agregar un empleado y para borrar la lista de
    empleados */
private JButton agregar, limpiar;

/**
* Constructor de la clase VentanaAgregarEmpleado
* @param lista Parámetro que define la lista de empleados
*/
public VentanaAgregarEmpleado(ListaEmpleados lista) {
    this.lista = lista;
    inicio();
    setTitle("Agregar Empleado"); // Establece el título de la ventana
    setSize(300,400); // Establece el tamaño de la ventana
    setLocationRelativeTo(null); /* La ventana se posiciona en el
        centro de la pantalla */
    setResizable(false); /* Establece que la ventana no puede cambiar
        de tamaño */
}

/**
* Método que crea la ventana con sus diferentes componentes gráficos
*/
public void inicio() {
    contenedor = getContentPane(); /* Obtiene el panel de
        contenidos de la ventana */
    contenedor.setLayout(null); /* Establece que el contenedor no
        tiene un layout */

    // Establece la etiqueta y el campo de texto nombre del empleado
    nombre = new JLabel();
    nombre.setText("Nombre:");
    // Establece la posición de la etiqueta nombre del empleado
    nombre.setBounds(20, 20, 135, 23);
    campoNombre = new JTextField();
    // Establece la posición del campo de texto nombre del empleado
    campoNombre.setBounds(160, 20, 100, 23);
```

```
// Establece la etiqueta y el campo de texto apellidos del empleado
apellidos = new JLabel();
apellidos.setText("Apellidos:");
// Establece la posición de la etiqueta apellidos del empleado
apellidos.setBounds(20, 50, 135, 23);
campoApellidos = new JTextField();
// Establece la posición del campo de texto apellidos del empleado
campoApellidos.setBounds(160, 50, 100, 23);

// Establece la etiqueta y el combo box del cargo del empleado
cargo = new JLabel();
cargo.setText("Cargo:");
cargo.setBounds(20, 80, 135, 23); /* Establece la posición de la
    etiqueta cargo del empleado */
campoCargo = new JComboBox();
// Agrega los tres tipos de cargo de un empleado al combobox
campoCargo.addItem("Directivo");
campoCargo.addItem("Estratégico");
campoCargo.addItem("Operativo");
// Establece la posición del combobox cargo del empleado
campoCargo.setBounds(160, 80, 100, 23);

/* Establece las etiquetas y el grupo de botones de radio para el
    género del empleado */
género = new JLabel();
género.setText(«Género:»);
// Establece la posición de la etiqueta de género del empleado
género.setBounds(20,110,100,30);
grupoGénero = new ButtonGroup(); // Crea un grupo de botones
masculino = new JRadioButton("Masculino", true);
masculino.setBounds(160,110,100,30); /* Establece la posición
    del botón de radio masculino */
grupoGénero.add(masculino); // Añade el botón al grupo
femenino = new JRadioButton("Femenino");
femenino.setBounds(160,140,100,30); /* Establece la posición
    del botón de radio femenino */
grupoGénero.add(femenino); // Añade el botón al grupo

/* Establece la etiqueta y el campo de texto salario por día del
    empleado */
salarioDía = new JLabel();
```

```
salarioDía.setText("Salario por día:");
// Establece la posición de la etiqueta salario por día del empleado
salarioDía.setBounds(20, 170, 135, 23);
campoSalarioDía = new JTextField();
/* Establece la posición del campo de texto salario por día del
    empleado */
campoSalarioDía.setBounds(160, 170,100, 23);

/* Establece la etiqueta y el campo de texto días trabajados al
    mes del empleado */
númeroDías = new JLabel();
númeroDías.setText(«Días trabajados al mes:»);
/* Establece la posición de la etiqueta días trabajados al mes del
    empleado */
númeroDías.setBounds(20, 200, 135, 23);
campoNúmeroDías = new JSpinner(); // Crea un spinner
modeloSpinner = new SpinnerNumberModel(); /* Crea un
    modelo numérico para el spinner */
// Define valor mínimo, máximo y valor inicial para el spinner
modeloSpinner.setMinimum(1);
modeloSpinner.setMaximum(31);
modeloSpinner.setValue(30);
campoNúmeroDías.setModel(modeloSpinner); /* Asocia el
    modelo numérico al spinner */
campoNúmeroDías.setBounds(160, 200, 40, 23); /* Establece la
    posición del spinner */

/* Establece la etiqueta y el campo de texto otros ingresos del
    empleado */
otrosIngresos = new JLabel();
otrosIngresos.setText("Otros ingresos:");
// Establece la posición de la etiqueta otro ingresos del empleado
otrosIngresos.setBounds(20, 230, 135, 23);
campoOtrosIngresos = new JTextField();
/* Establece la posición del campo de texto otros ingresos del
    empleado */
campoOtrosIngresos.setBounds(160, 230, 100, 23);

/* Establece la etiqueta y el campo de texto pagos por salud del
    empleado */
aportesSalud = new JLabel();
aportesSalud.setText("Pagos por salud:");
```

```
// Establece la posición de la etiqueta pagos por salud del empleado
aportesSalud.setBounds(20, 260, 135, 23);
campoAportesSalud = new JTextField();
/* Establece la posición del campo de texto pagos por salud del
    empleado */
campoAportesSalud.setBounds(160, 260, 100, 23);

/* Establece la etiqueta y el campo de texto aportes pensiones del
    empleado */
pensiones = new JLabel();
pensiones.setText("Aportes pensiones:");
// Establece la posición de la etiqueta aporte pensiones del empleado
pensiones.setBounds(20, 290, 135, 23);
campoPensiones = new JTextField();
/* Establece la posición del campo de texto aporte pensiones del
    empleado */
campoPensiones.setBounds(160, 290, 100, 23);

// Establece el botón agregar empleado
agregar = new JButton();
agregar.setText("Agregar");
agregar.setBounds(20, 320, 100, 23); /* Establece la posición del
    botón agregar */
/* Agrega al botón un ActionListener para que gestione eventos
    del botón */
agregar.addActionListener(this);

// Establece el botón borrar empleados
limpiar = new JButton();
limpiar.setText("Borrar");
limpiar.setBounds(160, 320, 80, 23); /* Establece la posición del
    botón borrar empleados */
/* Agrega al botón un ActionListener para que gestione eventos
    del botón */
limpiar.addActionListener(this);

// Se añade cada componente gráfico al contenedor de la ventana
contenedor.add(nombre);
contenedor.add(campoNombre);
contenedor.add(apellidos);
contenedor.add(campoApellidos);
contenedor.add(cargo);
```

```
        contenedor.add(campoCargo);
        contenedor.add(género);
        contenedor.add(masculino);
        contenedor.add(femenino);
        contenedor.add(salarioDía);
        contenedor.add(campoSalarioDía);
        contenedor.add(númeroDías);
        contenedor.add(campoNúmeroDías);
        contenedor.add(otrosIngresos);
        contenedor.add(campoOtrosIngresos);
        contenedor.add(aportesSalud);
        contenedor.add(campoAportesSalud);
        contenedor.add(pensiones);
        contenedor.add(campoPensiones);
        contenedor.add(agregar);
        contenedor.add(limpiar);
    }

    /**
    * Método que borra los campos de texto ingresados en la ventana de
    * agregar empleado
    */
    public void limpiarCampos() {
        campoNombre.setText("");
        campoApellidos.setText("");
        campoSalarioDía.setText("");
        campoNúmeroDías.setValue(0);
        campoOtrosIngresos.setText("");
        campoAportesSalud.setText("");
        campoPensiones.setText("");
    }

    /**
    * Método que gestiona los eventos generados en la ventana principal
    */
    public void actionPerformed(ActionEvent evento) {
        if (evento.getSource() == agregar) { /* Se pulsa el botón agregar
            empleado */
            añadirEmpleado();
        }
        if (evento.getSource() == limpiar) { // Se pulsa el botón limpiar
            limpiarCampos();
        }
```

```
}
/**
* Método que agrega un empleado a la lista de empleados
* throws Exception Excepción de campo nulo o error en formato de
* numero
*/
private void añadirEmpleado() {
    TipoCargo tipoC;
    // Obtiene el cargo seleccionado del combobox
    String itemSeleccionado = (String) campoCargo.
        getSelectedItem();
    /* De acuerdo al cargo seleccionado, se asigna el valor de
        atributo correspondiente */
    if (itemSeleccionado == "Directivo") {
        tipoC = TipoCargo.DIRECTIVO;
    } else if (itemSeleccionado == "Estratégico") {
        tipoC = TipoCargo.ESTRATÉGICO;
    } else {
        tipoC = TipoCargo.OPERATIVO;
    }
    TipoGénero tipoG;
    /* De acuerdo al género seleccionado, se asigna el valor de
        atributo correspondiente */
    if (masculino.isSelected()) {
        tipoG = TipoGénero.MASCULINO;
    } else {
        tipoG = TipoGénero.FEMENINO;
    }
    try {
        String valor1 = campoNombre.getText(); /* Se obtiene el
            valor del campo de texto nombre */
        String valor2 = campoApellidos.getText(); /* Se obtiene el
            valor del campo de texto apellidos */
        // Se obtiene y convierte el campo de texto salario
        double valor3 = Double.parseDouble(campoSalarioDía.
            getText());
        // Se obtiene el valor ingresado de días trabajados
        int valor4 = (int) campoNúmeroDías.getValue();
        // Se obtiene y convierte el campo de texto otros ingresos
        double valor5 = Double.parseDouble(campoOtrosIngresos.
            getText());
```

```
                // Se obtiene y convierte el campo de texto aportes salud
                double valor6 = Double.parseDouble(campoAportesSalud.
                    getText());
                // Se obtiene y convierte el campo de texto pensiones
                double valor7 = Double.parseDouble(campoPensiones.
                    getText());
                Empleado e = new Empleado(valor1, valor2, tipoC, tipoG,
                    valor3, valor4, valor5, valor6, valor7); // Se crea un empleado
                lista.agregarEmpleado(e); /* Se agrega un empleado a la lista
                    de empleados */
                // Mensaje de confirmación de empleado agregado a la lista
                JOptionPane.showMessageDialog(this,"El empleado ha sido
                    agregado","Mensaje", JOptionPane.INFORMATION_
                    MESSAGE,null);
                limpiarCampos();
            } catch (Exception e) { /* Si se produce algún tipo de error, se
                muestra un mensaje */
                JOptionPane.showMessageDialog(null,"Campo nulo o error
                    en formato de numero", "Error", JOptionPane.ERROR_
                    MESSAGE);
            }
        }
    }
```

Clase: VentanaNómina

```
package Nómina;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.table.DefaultTableModel;

/**
 * Esta clase denominada VentanaNómina define una interfaz gráfica que
 * permitirá consultar la lista de empleados y la nómina total.
 * @version 1.2/2020
 */
```

```
public class VentanaNómina extends JFrame {
        private Container contenedor; /* Un contenedor de elementos
            gráficos */
        private ListaEmpleados lista; // Lista de empleados de la empresa
        private JLabel empleados, nómina; /* Etiquetas estáticas
            empleados y nómina total */
        private JTable tabla; /* Tabla para mostrar datos de la lista de
            empleados */

    /**
    * Constructor de la clase VentanaNómina
    */
    public VentanaNómina(ListaEmpleados lista) {
        this.lista = lista;
        inicio();
        setTitle("Nómina de Empleados"); // Establece el título de la ventana
        setSize(350,250); // Establece el tamaño de la ventana
        setLocationRelativeTo(null); /* La ventana se posiciona en el
            centro de la pantalla */
        setResizable(false); /* Establece que la ventana no puede cambiar
            de tamaño */
    }

    /**
    * Método que crea la ventana con sus diferentes componentes
    * gráficos
    */
    public void inicio() {
        contenedor = getContentPane(); /* Obtiene el panel de
            contenidos de la ventana */
        contenedor.setLayout(null); /* Establece que el contenedor no
            tiene un layout */

        // Establece la etiqueta lista de empleados
        empleados = new JLabel();
        empleados.setText("Lista de empleados:");
        // Establece la posición de la etiqueta lista de empleados
        empleados.setBounds(20, 10, 135, 23);

        String[][] datos = lista.obtenerMatriz(); /* Convierte la lista de
            empleados a una matriz */
        String[] titulos = { "NOMBRE","APELLIDOS","SUELDO"};
        // Define cabecera de la tabla
```

```
        // Crea un modelo de tabla con su cabecera y matriz
        DefaultTableModel model = new
            DefaultTableModel(datos,titulos);
        tabla = new JTable(model); // Asocia el modelo a la tabla
        tabla.setBounds(20, 50, 310, 100); /* Establece la posición de la
            tabla de empleados */

        // Establece la etiqueta de total nómina mensual
        nómina = new JLabel();
        // Presenta el total de la nómina formateado
        nómina.setText(«Total nómina mensual = $ « + String.
            format(«%.2f», lista.totalNómina));
        // Establece la posición de la etiqueta total nómina mensual
        nómina.setBounds(20, 160, 250, 23);

        // Se añade cada componente gráfico al contenedor de la ventana
        contenedor.add(empleados);
        contenedor.add(tabla);
        contenedor.add(nómina);
    }
}
```

Clase: Principal

```
package Nómina;

/**
 * Esta clase define el punto de ingreso al programa de la nómina de
 * empleados. Por lo tanto, cuenta con un método main de acceso al
 * programa.
 * @version 1.2/2020
 */
public class Principal {

    /**
     * Método main que sirve de punto de entrada al programa
     */
    public static void main(String[] args) {
        VentanaPrincipal miVentanaPrincipal; // Define la ventana principal
        miVentanaPrincipal= new VentanaPrincipal(); /* Crea la ventana
            principal */
        miVentanaPrincipal.setVisible(true); /* Establece la ventana
            como visible */
```

```
            // Establece que la ventana no puede cambiar su tamaño
            miVentanaPrincipal.setResizable(false);
        }
    }
```

Diagrama de clases

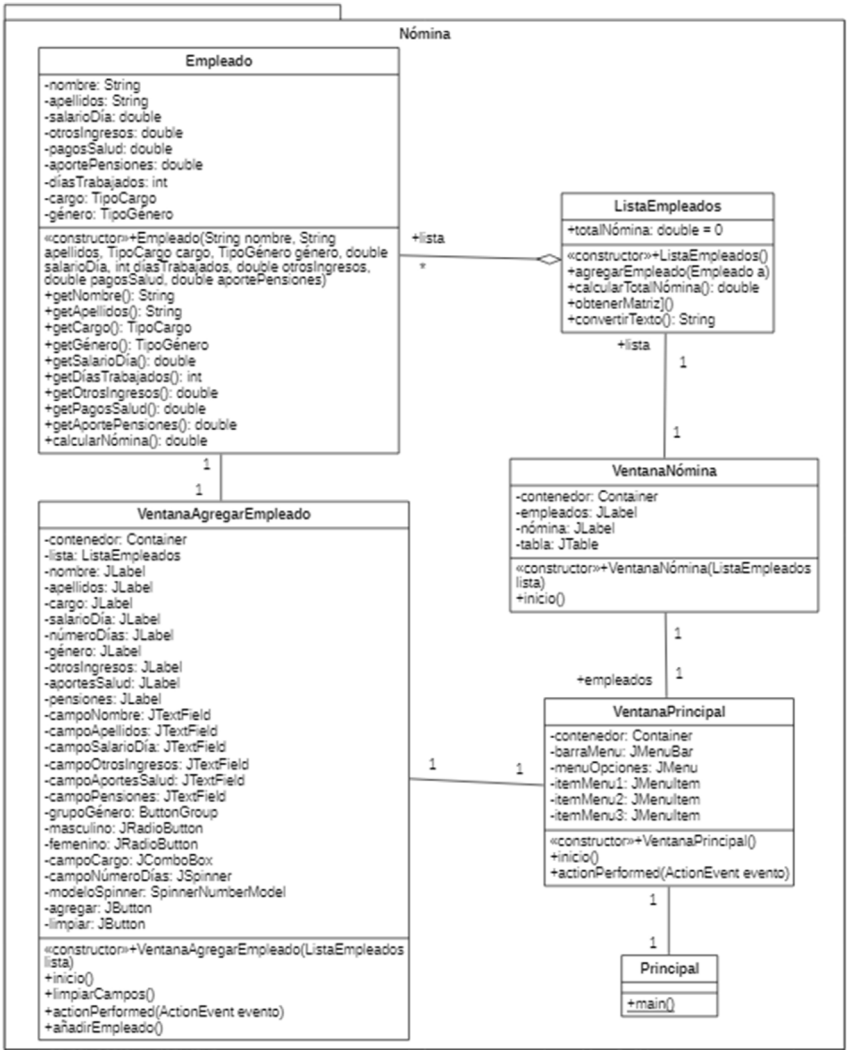


Figura 8.10. Diagrama de clases del ejercicio 8.4.

Explicación del diagrama de clases

Se ha definido un paquete denominado "Nómina" que incluye un conjunto de clases. El punto de entrada al programa es la clase Principal que cuenta con el método *main*. La clase Principal está relacionada mediante una asociación con la clase VentanaPrincipal que posee atributos privados para crear una ventana gráfica que cuenta con un contenedor de componentes gráficos (*Container*), una barra de menú (*JMenuBar*), una opción de menú (*JMenu*) y tres opciones de menú (*JMenuItem*). La clase VentanaPrincipal cuenta con un constructor y métodos para generar la ventana gráfica con sus componentes (inicio) y para gestionar los diferentes eventos surgidos al interactuar con esta ventana (*actionPerformed*).

La clase VentanaPrincipal está conectada por medio de una relación de asociación con las clases VentanaAgregarEmpleado y VentanaNómina. Las multiplicidades de estas relaciones son uno en cada extremo, lo que indica que se creará una sola ventana de su tipo cuando se invoquen desde la ventana principal.

En primer lugar, la clase VentanaAgregarEmpleado define una ventana que permite ingresar los datos de un empleado a agregar. Para ello, la clase VentanaAgregarEmpleado cuenta con atributos para definir los diferentes componentes gráficos de la ventana: un contenedor de componentes gráficos (*Container*); etiquetas para identificar los datos a ingresar (*JLabel*); campos de texto para ingresar los datos del empleado (*JTextField*); botones de radio (*JRadioButton*) agrupados (*ButtonGroup*) para identificar el género de empleado; una lista (*JComboBox*) para identificar el cargo del empleado, un selector numérico para ingresar la cantidad de días trabajados (*JSpinner* y *SpinnerNumberModel*) y botones para agregar un empleado o borrar un empleado (*JButton*). La clase VentanaAgregarEmpleado tiene un constructor y métodos para generar la ventana con sus componentes (inicio); gestionar los diferentes eventos surgidos al interactuar con esta ventana (*actionPerformed*); agregar un empleado (añadirEmpleado) y limpiar los campos de ingreso de datos (limpiarCampos).

La clase VentanaAgregarEmpleado se vincula con la clase Empleado por medio de una relación de asociación que tiene una multiplicidad de uno en cada extremo, lo que significa que la ventana creará un único empleado a la vez.

En segundo lugar, la clase VentanaNómina define una ventana que muestra en formato de tabla los principales datos de los empleados y calcula

el total de la nómina. Para ello, la clase VentanaNómina cuenta con atributos para definir los diferentes componentes gráficos de la ventana: un contenedor de componentes gráficos (*Container*); etiquetas para identificar los datos a mostrar y calcular (*JLabel*); una tabla para presentar en forma resumida los datos principales de los empleados (*JTable*) y un botón para calcular el total de la nómina (*JButton*). La clase VentanaNómina tiene un constructor y un método para generar la ventana con sus componentes (inicio).

La clase VentanaNómina para generar la tabla de empleados se relaciona con la clase ListaEmpleados por medio de una relación de asociación y a través del atributo lista de empleados, el cual es el nombre de un rol (extremo) de la relación de asociación. La clase ListaEmpleados es una clase contenedora de empleados. Para ello, cuenta con dos atributos: el total de la nómina y la lista de empleados, esta se expresa por medio de una relación de agregación con la clase Empleado. Esta relación de agregación se denota por medio de una línea continua con un rombo blanco en el extremo de la relación que se vincula con la clase ListaEmpleados (que representa el todo). La clase ListaEmpleados cuenta con un constructor y con métodos para agregar un empleado a la lista de empleados para calcular el total de la nómina, obtener la matriz de empleados y convertir a texto la información de la lista de empleados (utilizados para generar el *JTable* en VentanaNómina).

La clase Empleado tiene los siguientes atributos: nombre, apellidos, salario por día, otros ingresos, pagos por salud, aporte pensiones, días trabajados, cargo y género. La clase Empleado cuenta con un constructor que inicializa los atributos del empleado a crear y tiene métodos *get* y *set* para cada atributo. También cuenta con un método para calcular la nómina del empleado.

Diagrama de objetos

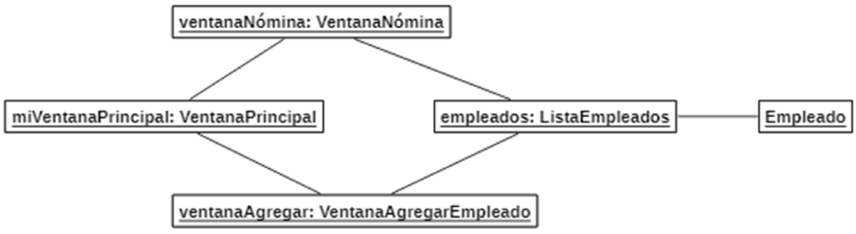
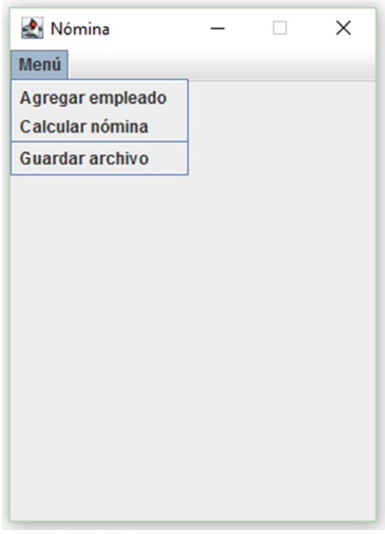
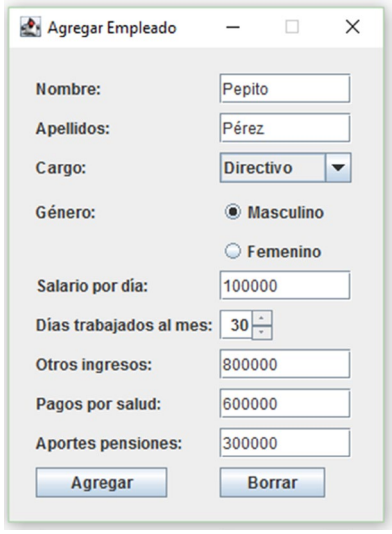


Figura 8.11. Diagrama de objetos del ejercicio 8.4.

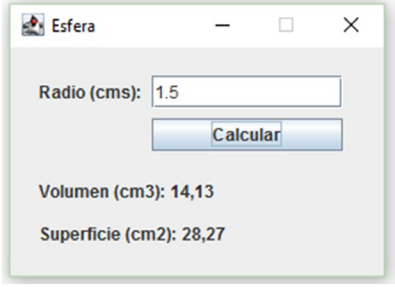
Ejecución del programa



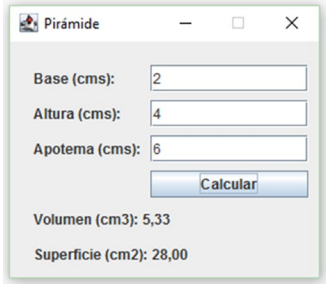
a) Ventana Principal



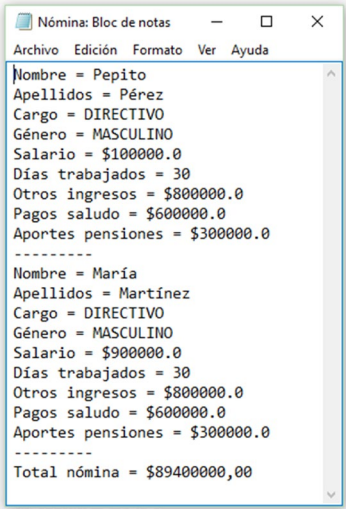
b) Agregar empleado



c) Calcular nómina



d) Guardar archivo



e) Guardar archivo

Figura 8.12. Ejecución del programa del ejercicio 8.4.

Ejercicios propuestos

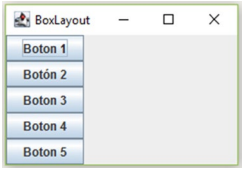
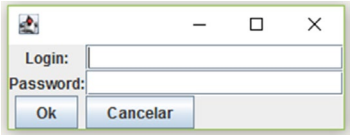
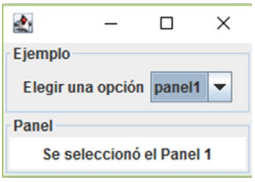
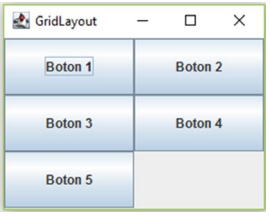
- Agregar las funcionalidades editar y eliminar empleados al programa.
- Agregar la funcionalidad guardar y leer los datos de la nómina de la empresa, pero como un archivo binario, no de texto.

Ejercicio 8.5. Gestión de contenidos

En una ventana de aplicación deben colocarse numerosos componentes de acuerdo con un cierto orden y disposición gráfica. En Java existen los administradores de diseño (objetos que implementa la interfaz *LayoutManager*) que determinan el tamaño y la posición de los componentes dentro de un contenedor (Clark, 2017). Aunque estos pueden proporcionar sugerencias de tamaño y alineación, el administrador de diseño de un contenedor tiene la última palabra sobre el tamaño y la posición de los componentes dentro del contenedor. Para su uso se requiere importar el paquete *java.awt.*;*

En la tabla 8.7 se presentan diferentes tipos de *layouts* (Deitel y Deitel, 2017, Kölling y Barnes, 2013 y API Java, 2020).

Tabla 8.7. Tipos de *layouts*

Tipo	Descripción	Figura
BorderLayout	Establece un contenedor, organizando y redimensionando para que sus componentes se ajusten a cinco regiones: Norte, Sur, Este, Oeste y centro.	
BoxLayout	Permite que varios componentes se distribuyan vertical u horizontalmente.	
GridBagLayout	Alinea los componentes verticalmente, horizontalmente o a lo largo de su línea de base, sin que los componentes sean del mismo tamaño.	
FlowLayout	Organiza los componentes en un flujo direccional, al igual que las líneas de texto en un párrafo.	
CardLayout	Trata cada componente en el contenedor como una tarteta. Solo se puede ver una tarjeta a la vez y el contenedor actúa como una pila de tarjetas.	
GridLayout	Presenta los componentes de un contenedor en una cuadrícula rectangular. El contenedor se divide en rectángulos de igual tamaño y se coloca un componente en cada rectángulo.	

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para generar interfaces gráficas de usuario utilizando diferentes *layouts*.

Enunciado: Hotel

Se requiere un programa que permita gestionar el ingreso y salida de los huéspedes de un hotel. El hotel contiene diez habitaciones simples. Las primeras cinco habitaciones tienen un precio de $120 000 por día y las otras cinco habitaciones, $160 000 por día.

El programa cuenta con dos opciones de menú:

- Consultar habitaciones: al seleccionar este ítem de menú se debe generar una ventana que presenta las diez habitaciones del hotel y su estado: disponible o no disponible. El usuario debe seleccionar el número de la habitación a ocupar. Si se ingresa una habitación ocupada se genera el mensaje de error correspondiente.

  Luego, el programa genera una ventana donde se ingresa la fecha de ingreso y los datos del huésped (nombre, apellidos y número de documento de identidad). Los campos de entrada son obligatorios y se deben validar previamente. Si el registro es correcto se genera el mensaje correspondiente. Si después se consulta el listado de habitaciones, la habitación debe aparecer como "No disponible".

- Salida de huéspedes: al seleccionar este ítem de menú se debe generar una ventana donde se solicita el número de habitación a entregar. Si la habitación no está ocupada o se ingresa un número o dato incorrecto se genera el mensaje de error correspondiente. Si el número es correcto, se genera una nueva ventana donde se identifica la habitación a entregar y se ingresa la fecha de salida del huésped. Esta fecha debe ser mayor a la fecha de ingreso al hotel. Si la fecha es correcta se habilita un botón que permite calcular el total de días de alojamiento del huésped y el total a pagar por el mismo. Si el usuario oprime el botón "Registrar salida", la habitación queda disponible.

Se deben utilizar los *layouts* que se consideren convenientes para organizar las diferentes configuraciones de los componentes visuales de las ventanas del programa.

Instrucciones Java del ejercicio

Tabla 8.8. Instrucciones Java del ejercicio 8.5.

Clase	Método	Descripción
Integer	*Integer valueOf(int i)*	Retorna un objeto *Integer* que representa el valor *int* especificado.
String	*String format(String formato, Object … args)*	Retorna un *String* formateado utilizando el formato y argumentos especificados.
JFrame	*JFrame()*	Constructor de la clase *JFrame*.
	void setTitle(String título)	Establece el título de la ventana con el *String* especificado.
	void setSize(int x, int y)	Cambia el tamaño del componente para que tenga una anchura *x* y una altura *y*.
	void setLocationRelativeTo (Component c)	Establece la ubicación de la ventana en relación con el componente especificado.
	void setDefaultCloseOperation (opciones)	Usado para especificar una de las siguientes opciones del botón de cierre: EXIT_ON_CLOSE, HIDE_ON_CLOSE, DISPOSE_ON_CLOSE o DO_ NOTHING_ON_CLOSE.
	void setResizable(boolean resizable)	Para evitar que se cambie el tamaño de la ventana.
	void setVisible(boolean b)	Muestra u oculta la ventana según el valor del parámetro b.
ActionListener	*void actionPerformed(actionEvent e)*	Se invoca cuando ocurre un evento.
Container	*Container getContentPane()*	Retorna el objeto *ContentPane* de la ventana.
	void setLayout(LayoutManager mgr)	Establece el *layout* de la ventana.
	Component add(Component comp)	Añade el componente especificado al final del contenedor.
Component	*void addActionListener(this)*	Añade un oyente de eventos al componente actual.
	void setBounds(int x, int y, int ancho, int alto)	Mueve y cambia el tamaño del componente.
JLabel	*JLabel()*	Constructor de la clase *JLabel*.
	void setText(String text)	Define una línea de texto que mostrará este componente.

Clase	Método	Descripción
JTextField	*JTextField()*	Constructor de la clase *JTextField*.
	String getText()	Retorna el texto contenido en el componente de texto.
JButton	*JButton()*	Constructor de la clase *JButton*.
	void setText(String text)	Define una línea de texto que mostrará este componente.
	int getSelectedIndex()	Devuelve el índice seleccionado.
JMenuBar	*JMenuBar()*	Constructor de la clase *JMenuBar*.
	JMenu add(JMenu c)	Añade el menú especificado al final de la barra de menú.
JMenu	*JMenu()*	Constructor de la clase *JMenu*.
	JMenuItem add(JMenuItem menuItem)	Añade un ítem de menú al final del menú.
JMenuItem	*JMenuItem()*	Constructor de la clase *JMenuItem*.
JSpinner	*void setModel(SpinnerModel modelo)*	Cambia el modelo que representa el valor del *spinner*.
SpinnerNumberModel	*void setMinimum(Comparable mínimo)*	Cambia el mínimo inferior en la secuencia.
	void setMaximum(Comparable máximo)	Cambia el máximo superior en la secuencia.
	void setValue(Object valor)	Establece el valor actual en esta secuencia.
GridBagLayout	*GridBagLayout()*	Constructor de la clase *GridBagLayout*.
GridBagConstraints	*GridBagConstraints()*	Constructor de la clase *GridBagConstraints*.
	int HORIZONTAL	Cambie el tamaño del componente horizontalmente pero no verticalmente.
	int gridx	Especifica la celda inicial en el eje *x* del área de visualización del componente.
	int gridy	Especifica la celda inicial en el eje *y* del área de visualización del componente.
Insets	*Insets(int superior, int izquierda, int inferior, int derecha)*	Crea e inicializa un objeto *Insets* con las inserciones superior, izquierda, inferior y derecha especificadas.
SimpleDateFormat	*SimpleDateFormat(String patrón)*	Construye un *SimpleDateFormat* utilizando el patrón dado y los símbolos de formato de fecha predeterminados.
Date	*Date parse(Strings)*	Convierte el *String* a tipo *date*.
	int compareTo(Date otraFecha)	Compara dos fechas según orden.
	long getTime()	Retorna el número de milisegundos desde enero 1 de 1970 00:00:00.

Clase	Método	Descripción
Event	*Object getSource()*	El objeto sobre el cual el evento inicialmente ha ocurrido.
JOptionPane	*void showMessageDialog (Component componentePadre, Object mensaje)*	Crea un cuadro de diálogo.
	void showInputDialog (Component componentePadre, Object mensaje)	Muestra un cuadro de diálogo que solicita entrada de datos.

Solución

Clase: Hotel

```
package Hotel;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.*;

/**
 * Esta clase denominada Hotel define un hotel que contiene diez
 * habitaciones a ser ocupadas y liberadas por diferentes huéspedes en
 * fechas determinadas.
 * @version 1.2/2020
 */
public class Hotel {
    // Atributo que establece el conjunto de habitaciones del hotel
    public static Vector<Habitación> listaHabitaciones;

    /**
    * Constructor de la clase Hotel
    */
    public Hotel() {
        listaHabitaciones = new Vector<Habitación>(); /* Crea el vector
            de habitaciones */
        /* Crea cada habitación con un número de habitación,
            disponibilidad inicial y precio por día */
        Habitación habitación1 = new Habitación(1, true, 120000);
        Habitación habitación2 = new Habitación(2, true, 120000);
        Habitación habitación3 = new Habitación(3, true, 120000);
        Habitación habitación4 = new Habitación(4, true, 120000);
        Habitación habitación5 = new Habitación(5, true, 120000);
        Habitación habitación6 = new Habitación(6, true, 160000);
```

```
        Habitación habitación7= new Habitación(7, true, 160000);
        Habitación habitación8= new Habitación(8, true, 160000);
        Habitación habitación9 = new Habitación(9, true, 160000);
        Habitación habitación10 = new Habitación(10, true, 160000);

        // Se agrega cada habitación al vector de habitaciones del hotel
        listaHabitaciones.add(habitación1);
        listaHabitaciones.add(habitación2);
        listaHabitaciones.add(habitación3);
        listaHabitaciones.add(habitación4);
        listaHabitaciones.add(habitación5);
        listaHabitaciones.add(habitación6);
        listaHabitaciones.add(habitación7);
        listaHabitaciones.add(habitación8);
        listaHabitaciones.add(habitación9);
        listaHabitaciones.add(habitación10);
    }

    /**
    * Método que dado un número de habitación, busca la fecha de
    * ingreso de un huésped a dicha habitación
    * @param número Número de habitación a buscar
    * @return Fecha de ingreso a la habitación
    */
    public String buscarFechaIngresoHabitación(int número) {
        for(int i =0; i < listaHabitaciones.size(); i++) { /* Recorre el vector
            de habitaciones */
            // Obtiene un elemento del vector
            Habitación habitación = (Habitación) listaHabitaciones.
                elementAt(i);
            if (habitación.getNúmeroHabitación() == número) { /* Si el
                número buscado es encontrado */
                // Se obtiene la fecha de ingreso
                Date fecha = habitación.getHuésped().getFechaIngreso();
                // Se le da formato a la fecha de ingreso
                DateFormat formatoFecha = new
                    SimpleDateFormat("yyyy/MM/dd");
                return formatoFecha.format(fecha); /* Devuelve la fecha
                    de ingreso */
            }
        }
```

```
            return ""; // En caso de no encontrar la habitación
        }

        /**
        * Método que dado un número de habitación, devuelve si la
        * habitación está ocupada o no
        * @param número Número de habitación a buscar
        * @return Valor booleano con la disponibilidad de la habitación
        * buscada
        */
        public boolean buscarHabitaciónOcupada(int número) {
            for(int i =0; i < listaHabitaciones.size(); i++) { /* Recorre el vector
                de habitaciones */
                // Obtiene un elemento del vector
                Habitación habitación = (Habitación) listaHabitaciones.
                    elementAt(i);
                if (habitación.getNúmeroHabitación() == número &&
                    !habitación.getDisponible()) {
                    // Si la habitación está disponible
                    return true;
                }
            }
            return false; // Si la habitación no está disponible
        }
    }
```

Clase: Habitación

```
package Hotel;

/**
 * Esta clase denominada Habitación define una habitación de un hotel
 * a ser ocupada y desocupada por un huésped.
 * @version 1.2/2020
 */
public class Habitación {
    private int númeroHabitación; /* Atributo que indica el número de
        la habitación */
    private boolean disponible; /* Atributo que indica la disponibilidad
        de la habitación */
    private double precioDía; /* Atributo que indica el precio por día de
        la habitación */
```

```
    private Huésped huésped; /* Atributo que indica el huésped que
        ocupa la habitación */

    /**
    * Constructor de la clase Habitación
    * @param númeroHabitación Parámetro que define el número de la
    * habitación
    * @param disponible Parámetro que define la disponibilidad de la
    * habitación
    * @param precioDía Parámetro que define el precio por día de la
    * habitación
    */
    public Habitación(int númeroHabitación, boolean disponible,
        double precioDía) {
        this.númeroHabitación = númeroHabitación;
        this.disponible = disponible;
        this.precioDía = precioDía;
    }

    /**
    * Método que obtiene el número de habitación
    * @return El número de habitación
    */
    public int getNúmeroHabitación() {
        return númeroHabitación;
    }

    /**
    * Método que obtiene la disponibilidad de la habitación
    * @return La disponibilidad de la habitación
    */
    public boolean getDisponible() {
        return disponible;
    }

    /**
    * Método que obtiene el precio por día de la habitación
    * @return El precio por día de la habitación
    */
    public double getPrecioDía() {
        return precioDía;
    }

    /**
    * Método que obtiene el huésped de la habitación
```

```
        * @return El huésped de la habitación
        */
        public Huésped getHuésped() {
            return huésped;
        }

        /**
        * Método que establece el huésped de la habitación
        * @param Parámetro que define el huésped de la habitación
        */
        public void setHuésped(Huésped huésped) {
            this.huésped = huésped;
        }

        /**
        * Método que establece la disponibilidad de la habitación
        * @param Parámetro que define la disponibilidad de la habitación
        */
        public void setDisponible(boolean disponible) {
            this.disponible = disponible;
        }
    }
```

Clase: Huésped

```
    package Hotel;
    import java.time.*;
    import java.util.*;

    /**
     * Esta clase denominada Huésped define un huésped del hotel que
     * ocupará una determinada habitación por ciertos días.
     * @version 1.2/2020
     */
    public class Huésped {
        private String nombres; /* Atributo que identifica los nombres del
            huésped */
        private String apellidos; /* Atributo que identifica los apellidos del
            huésped */
        /* Atributo que identifica el número de documento de identidad del
            huésped */
        private int documentoIdentidad;
```

```
private Date fechaIngreso; /* Atributo que identifica la fecha de
    ingreso del huésped */
private Date fechaSalida; /* Atributo que identifica la fecha de salida
    del huésped */

/**
* Constructor de la clase Huésped
* @param nombres Parámetro que define los nombres del huésped
* @param apellidos Parámetro que define los apellidos del huésped
* @param documentoIdentidad Parámetro que define el número de
* documento de identidad del huésped
*/
public Huésped(String nombres, String apellidos, int
    documentoIdentidad)  {
    this.nombres = nombres;
    this.apellidos = apellidos;
    this.documentoIdentidad = documentoIdentidad;
}

/**
* Método que establece la fecha de salida del huésped
* @param La fecha de salida del huésped
*/
public void setFechaSalida(Date fecha) {
    fechaSalida = fecha;
}

/**
* Método que establece la fecha de ingreso del huésped
* @param La fecha de ingreso del huésped
*/
public void setFechaIngreso(Date fecha) {
    fechaIngreso = fecha;
}

/**
* Método que obtiene la fecha de ingreso del huésped
* @return La fecha de ingreso del huésped
*/
public Date getFechaIngreso() {
    return fechaIngreso;
}
```

```
    /**
     * Método que calcula la cantidad de días de alojamiento del huésped
     * @param La cantidad de días de alojamiento del huésped
     */
    public int obtenerDíasAlojamiento() {
        /* Resta la fecha de ingreso de la fecha de salida utilizando el
           método getTime */
        int días = (int) ((fechaSalida.getTime() - fechaIngreso.
           getTime())/86400000);
        return días;
    }
}
```

Clase: VentanaPrincipal

```
package Hotel;

import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.io.*;

/**
 * Esta clase denominada VentanaPrincipal define una interfaz gráfica
 * que permitirá gestionar el ingreso y salida de huéspedes a un hotel.
 * @version 1.2/2020
 */
public class VentanaPrincipal extends JFrame implements
ActionListener {
    private Container contenedor; // Un contenedor de elementos gráficos
    private JMenuBar barraMenu; // Una barra de menú principal
    private JMenu menuOpciones; /* Un menú de la barra de menú
        principal */
    private JMenuItem itemMenu1; // Un ítem de menú
    private JMenuItem itemMenu2; // Un ítem de menú
    private Hotel hotel; // Definición del hotel

    /**
     * Constructor de la clase VentanaPrincipal
```

```
     * @param hotel Parámetro que define un hotel
     */
    public VentanaPrincipal(Hotel hotel) {
        this.hotel = hotel;
        inicio();
        setTitle("Hotel"); // Establece el título de la ventana
        setSize(280,380); // Establece el tamaño de la ventana
        setLocationRelativeTo(null); /* La ventana se posiciona en el
            centro de la pantalla */
        // Establece que el botón de cerrar permitirá salir de la aplicación
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setResizable(false); /* Establece que el tamaño de la ventana no
            puede cambiar */
    }

    /**
     * Método que crea la ventana con sus diferentes componentes
     * gráficos
     */
    private void inicio() {
        contenedor = getContentPane(); /* Obtiene el panel de
            contenidos de la ventana */
        contenedor.setLayout(null); /* Establece que el contenedor no
            tiene un layout */

        // Se crea la barra de menús, un menú y dos ítems de menú
        barraMenu = new JMenuBar();
        menuOpciones = new JMenu("Menú");
        itemMenu1 = new JMenuItem("Consultar habitaciones");
        itemMenu2 = new JMenuItem("Salida de huéspedes");
        menuOpciones.add(itemMenu1); /* Se agrega el ítem de menú 1
            al menú */
        menuOpciones.add(itemMenu2); /* Se agrega el ítem de menú 2
            al menú */
        barraMenu.add(menuOpciones); /* Se agregan las opciones de
            menú al menú */
        setJMenuBar(barraMenu); /* Se agrega el menú a la barra de
            menús */
        /* Agrega al ítem de menú 1 un ActionListener para que gestione
            eventos del ítem de menú */
        itemMenu1.addActionListener(this);
        /* Agrega al ítem de menú 2 un ActionListener para que gestione
            eventos del ítem de menú */
```

```
        itemMenu2.addActionListener(this);
    }

    /**
    * Método que gestiona los eventos generados en la ventana principal
    */
    @Override
    public void actionPerformed(ActionEvent evento) {
        if (evento.getSource() == itemMenu1) { /* Se selecciona el ítem
            de menú 1 */
            // Se crea la ventana habitaciones
            VentanaHabitaciones ventanaHabitaciones = new
                VentanaHabitaciones(hotel);
            ventanaHabitaciones.setVisible(true); /* Se visualiza la
                ventana habitaciones */
        }
        if (evento.getSource() == itemMenu2) { /* Se selecciona el ítem
            de menú 2 */
            try {
                // Se ingresa el número de habitación
                String númeroHabitación = JOptionPane.
                    showInputDialog(null,
                    "Ingrese número de habitación","Salida de huéspedes",
                    JOptionPane.QUESTION_MESSAGE);
                int número = Integer.valueOf(númeroHabitación); /* Se
                    convierte el texto ingresado a int */
                if (número < 1 || número > 10) {
                    /* Si el número es incorrecto, se muestra un mensaje
                        de error */
                    JOptionPane.showMessageDialog(this,"El
                        número de habitación debe estar entre 1 y
                        10","Mensaje",JOptionPane.INFORMATION_
                        MESSAGE,null);
                } else if (hotel.buscarHabitaciónOcupada(número)) {
                    // Se busca que la habitación esté ocupada
                    // Se crea ventana de salida del huésped
                    VentanaSalida ventanaSalida = new
                        VentanaSalida(hotel, número);
                    ventanaSalida.setVisible(true); /* Se visualiza la
                        ventana de salida del huésped */
                } else { /* Si la habitación no está ocupada, se muestra un
                    mensaje de error */
```

```
                    JOptionPane.showMessageDialog(this,"La habitación
                        ingresada no ha sido ocupada","Mensaje",
                        JOptionPane.INFORMATION_MESSAGE,null);
                }
            } catch (Exception e) { // Si se presenta algún tipo de error
                    JOptionPane.showMessageDialog(null,"Campo nulo o
                        error en formato de numero","Error", JOptionPane.
                        ERROR_MESSAGE);
            }
        }
    }
}
```

Clase: VentanaHabitaciones

```
package Hotel;

import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.util.*;

/**
 * Esta clase denominada VentanaHabitaciones define una ventana que
 * muestra las diez habitaciones del hotel y su disponibilidad
 * @version 1.2/2020
 */
public class VentanaHabitaciones extends JFrame implements
ActionListener {
    private Container contenedor; /* Un contenedor de elementos
        gráficos */
    // Etiquetas de cada habitación
    private JLabel habitación1, habitación2, habitación3, habitación4,
        habitación5;
    private JLabel habitación6, habitación7, habitación8, habitación9,
        habitación10;
    // Etiquetas de disponibilidad cada habitación
    private JLabel disponibleHab1, disponibleHab2, disponibleHab3,
        disponibleHab4, disponibleHab5;
```

```
private JLabel disponibleHab6, disponibleHab7, disponibleHab8,
    disponibleHab9, disponibleHab10;
private JLabel habitaciónSeleccionada; /* Etiqueta para indicar la
    habitación a ocupar */
private JSpinner campoHabitaciónSeleccionada; /* Selector
    numérico de habitación a ocupar */
private SpinnerNumberModel modeloSpinner; /* Modelo de datos
    del selector */
private JButton botónAceptar; // Botón aceptar
private Hotel hotel; // Objeto Hotel

/**
* Constructor de la clase VentanaHabitaciones
* @param hotel Hotel con habitaciones y huéspedes
*/
public VentanaHabitaciones(Hotel hotel) {
    this.hotel = hotel;
    inicio();
    setTitle("Habitaciones"); // Establece el título de la ventana
    setSize(760,260); // Establece el tamaño de la ventana
    setLocationRelativeTo(null); /* La ventana se posiciona en el
        centro de la pantalla */
    setResizable(false); /* Establece que el tamaño de la ventana no
        puede cambiar */
}

/**
* Método que crea la ventana con sus diferentes componentes gráficos
*/
private void inicio() {
    contenedor = getContentPane(); /* Obtiene el panel de
        contenidos de la ventana */
    contenedor.setLayout(null); /* Establece que el contenedor no
        tiene un layout */
    Habitación habitación;

    // Establece las etiquetas de la habitación 1 y su disponibilidad
    habitación1 = new JLabel();
    habitación1.setText("Habitación 1");
    // Establece la posición de la etiqueta de la habitación 1
    habitación1.setBounds(20, 30, 130, 23);
```

```
disponibleHab1 = new JLabel("Disponible");
/* Establece la posición de la etiqueta de la disponibilidad de la
    habitación 1 */
disponibleHab1.setBounds(20, 50, 100, 23);
// Obtiene la habitación 1 de la lista de habitaciones
habitación = (Habitación) hotel.listaHabitaciones.elementAt(0);
if (!habitación.getDisponible()) { /* Determina si la habitación 1
    está disponible o no */
    disponibleHab1.setText("No disponible");
}

// Establece las etiquetas de la habitación 2 y su disponibilidad
habitación2 = new JLabel();
habitación2.setText("Habitación 2");
// Establece la posición de la etiqueta de la habitación 2
habitación2.setBounds(160, 30, 130, 23);
disponibleHab2 = new JLabel("Disponible");
/* Establece la posición de la etiqueta de la disponibilidad de la
    habitación 2 */
disponibleHab2.setBounds(160, 50, 100, 23);
// Obtiene la habitación 2 de la lista de habitaciones
habitación = (Habitación) hotel.listaHabitaciones.elementAt(1);
if (!habitación.getDisponible()) { /* Determina si la habitación 2
    está disponible o no */
disponibleHab2.setText("No disponible");
}

// Establece las etiquetas de la habitación 3 y su disponibilidad
habitación3 = new JLabel();
habitación3.setText("Habitación 3");
// Establece la posición de la etiqueta de la habitación 3
habitación3.setBounds(300, 30, 130, 23);
disponibleHab3 = new JLabel("Disponible");
/* Establece la posición de la etiqueta de la disponibilidad de la
    habitación 3 */
disponibleHab3.setBounds(300, 50, 100, 23);
// Obtiene la habitación 3 de la lista de habitaciones
habitación = (Habitación) hotel.listaHabitaciones.elementAt(2);
if (!habitación.getDisponible()) { /* Determina si la habitación 3
    está disponible o no */
    disponibleHab3.setText("No disponible");
}
```

```
// Establece las etiquetas de la habitación 4 y su disponibilidad
habitación4 = new JLabel();
habitación4.setText("Habitación 4");
// Establece la posición de la etiqueta de la habitación 4
habitación4.setBounds(440, 30, 130, 23);
disponibleHab4 = new JLabel("Disponible");
/* Establece la posición de la etiqueta de la disponibilidad de la
    habitación 4 */
disponibleHab4.setBounds(440, 50, 100, 23);
// Obtiene la habitación 4 de la lista de habitaciones
habitación = (Habitación) hotel.listaHabitaciones.elementAt(3);
if (!habitación.getDisponible()) { /* Determina si la habitación 4
    está disponible o no */
    disponibleHab4.setText("No disponible");
}

// Establece las etiquetas de la habitación 5 y su disponibilidad
habitación5 = new JLabel();
// Establece la posición de la etiqueta de la habitación 5
habitación5.setText("Habitación 5");
habitación5.setBounds(580, 30, 135, 23);
disponibleHab5 = new JLabel("Disponible");
/* Establece la posición de la etiqueta de la disponibilidad de la
    habitación 5 */
disponibleHab5.setBounds(580, 50, 100, 23);
// Obtiene la habitación 5 de la lista de habitaciones
habitación = (Habitación) hotel.listaHabitaciones.elementAt(4);
if (!habitación.getDisponible()) { /* Determina si la habitación 5
    está disponible o no */
    disponibleHab5.setText("No disponible");
}

// Establece las etiquetas de la habitación 6 y su disponibilidad
habitación6 = new JLabel();
habitación6.setText("Habitación 6");
// Establece la posición de la etiqueta de la habitación 6
habitación6.setBounds(20, 120, 130, 23);
disponibleHab6 = new JLabel("Disponible");
/* Establece la posición de la etiqueta de la disponibilidad de la
    habitación 6 */
disponibleHab6.setBounds(20, 140, 100, 23);
// Obtiene la habitación 6 de la lista de habitaciones
habitación = (Habitación) hotel.listaHabitaciones.elementAt(5);
```

```
if (!habitación.getDisponible()) { /* Determina si la habitación 6
    está disponible o no */
    disponibleHab6.setText("No disponible");
}

// Establece las etiquetas de la habitación 7 y su disponibilidad
habitación7 = new JLabel();
habitación7.setText("Habitación 7");
// Establece la posición de la etiqueta de la habitación 7
habitación7.setBounds(160, 120, 130, 23);
disponibleHab7 = new JLabel("Disponible");
/* Establece la posición de la etiqueta de la disponibilidad de la
    habitación 7 */
disponibleHab7.setBounds(160, 140, 100, 23);
// Obtiene la habitación 7 de la lista de habitaciones
habitación = (Habitación) hotel.listaHabitaciones.elementAt(6);
if (!habitación.getDisponible()) { /* Determina si la habitación 7
    está disponible o no */
    disponibleHab7.setText("No disponible");
}

// Establece las etiquetas de la habitación 8 y su disponibilidad
habitación8 = new JLabel();
habitación8.setText("Habitación 8");
// Establece la posición de la etiqueta de la habitación 8
habitación8.setBounds(300, 120, 130, 23);
disponibleHab8 = new JLabel("Disponible");
/* Establece la posición de la etiqueta de la disponibilidad de la
    habitación 8 */
disponibleHab8.setBounds(300, 140, 100, 23);
// Obtiene la habitación 8 de la lista de habitaciones
habitación = (Habitación) hotel.listaHabitaciones.elementAt(7);
if (!habitación.getDisponible()) { /* Determina si la habitación 8
    está disponible o no */
    disponibleHab8.setText("No disponible");
}

// Establece las etiquetas de la habitación 9 y su disponibilidad
habitación9 = new JLabel();
habitación9.setText("Habitación 9");
// Establece la posición de la etiqueta de la habitación 9
habitación9.setBounds(440, 120, 130, 23);
disponibleHab9 = new JLabel("Disponible");
```

```
/* Establece la posición de la etiqueta de la disponibilidad de la
    habitación 9 */
disponibleHab9.setBounds(440, 140, 100, 23);
// Obtiene la habitación 9 de la lista de habitaciones
habitación = (Habitación) hotel.listaHabitaciones.elementAt(8);
if (!habitación.getDisponible()) { /* Determina si la habitación 9
    está disponible o no */
    disponibleHab9.setText("No disponible");
}

// Establece las etiquetas de la habitación 10 y su disponibilidad
habitación10 = new JLabel();
habitación10.setText("Habitación 10");
// Establece la posición de la etiqueta de la habitación 10
habitación10.setBounds(580, 120, 135, 23);
disponibleHab10 = new JLabel("Disponible");
/* Establece la posición de la etiqueta de la disponibilidad de la
    habitación 10 */
disponibleHab10.setBounds(580, 140, 100, 23);
// Obtiene la habitación 10 de la lista de habitaciones
habitación = (Habitación) hotel.listaHabitaciones.elementAt(9);
if (!habitación.getDisponible()) { /* Determina si la habitación 10
    está disponible o no */
    disponibleHab10.setText("No disponible");
}

// Establece la etiquetas de la habitación a reservar
habitaciónSeleccionada = new JLabel();
habitaciónSeleccionada.setText("Habitación a reservar:");
// Establece la posición de la etiqueta de la habitación a reservar
habitaciónSeleccionada.setBounds(250, 180, 135, 23);
campoHabitaciónSeleccionada = new JSpinner(); /* Crea un
    selector numérico */
modeloSpinner = new SpinnerNumberModel(); /* Crea un
    modelo numérico para el selector */
// Define valores mínimo, máximo y actual del modelo numérico
modeloSpinner.setMinimum(1);
modeloSpinner.setMaximum(10);
modeloSpinner.setValue(1);
// Asocia el modelo numérico al spinner
campoHabitaciónSeleccionada.setModel(modeloSpinner);
// Establece la posición del selector numérico
```

```
        campoHabitaciónSeleccionada.setBounds(380, 180, 40, 23);

        // Establece el botón aceptar
        botónAceptar = new JButton("Aceptar");
        botónAceptar.setBounds(500, 180, 100, 23); /* Establece la
            posición del botón aceptar */

        /* Se agregan los componentes gráficos al contenedor de la
            ventana */
        contenedor.add(habitación1);
        contenedor.add(disponibleHab1);
        contenedor.add(habitación2);
        contenedor.add(disponibleHab2);
        contenedor.add(habitación3);
        contenedor.add(disponibleHab3);
        contenedor.add(habitación4);
        contenedor.add(disponibleHab4);
        contenedor.add(habitación5);
        contenedor.add(disponibleHab5);
        contenedor.add(habitación6);
        contenedor.add(disponibleHab6);
        contenedor.add(habitación7);
        contenedor.add(disponibleHab7);
        contenedor.add(habitación8);
        contenedor.add(disponibleHab8);
        contenedor.add(habitación9);
        contenedor.add(disponibleHab9);
        contenedor.add(habitación10);
        contenedor.add(disponibleHab10);
        contenedor.add(habitaciónSeleccionada);
        contenedor.add(campoHabitaciónSeleccionada);
        contenedor.add(botónAceptar);
        /* Agrega al botón un ActionListener para que gestione eventos
            del botón */
        botónAceptar.addActionListener(this);
    }

    /**
    * Método que gestiona los eventos generados en la ventana principal
    */
    @Override
    public void actionPerformed(ActionEvent evento) {
```

```
            if (evento.getSource() == botónAceptar) { /* Se pulsa el botón
                Aceptar */
                // Se obtiene la habitación seleccionada
                int habitación = (int) campoHabitaciónSeleccionada.
                    getValue();
                if (!hotel.buscarHabitaciónOcupada(habitación)) { /* Si la
                    habitación no está ocupada */
                    // Se registra ingreso en la habitación
                    VentanaIngreso ventanaIngreso = new
                        VentanaIngreso(hotel, habitación);
                    setVisible(false); /* La ventana con el listado de
                        habitaciones se cierra */
                    ventanaIngreso.setVisible(true); /* La ventana de ingreso
                        del húesped se hace visible */
                } else {
                    /* Si la habitación está ocupada se genera un cuadro de
                        diálogo informando la situación */
                        JOptionPane.showMessageDialog(this,"La habitación
                            está ocupada","Mensaje",
                  JOptionPane.INFORMATION_MESSAGE,null);
                }
            }
        }
    }
```

Clase: VentanaIngreso

```
package Hotel;

import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.util.*;
import java.text.SimpleDateFormat;
import java.text.ParseException;

/**
 * Esta clase denominada VentanaIngreso define una ventana que
 * permite ingresar los datos del huésped que tomará una habitación
 * previamente seleccionada.
```

```
 * @version 1.2/2020
 */
public class VentanaIngreso extends JFrame implements ActionListener
{
    private Container contenedor; // Un contenedor de elementos gráficos
    private JLabel habitación; /* Etiqueta para identificar la habitación
        seleccionada */
    private JButton aceptar, cancelar; /* Botones para aceptar o cancelar
        el ingreso del huésped */
    private JLabel nombre, apellidos, documentoIdentidad; /* Etiquetas
        para los campos de texto */
    private JLabel huésped; // Etiqueta para identificar el huésped
    // Campos de texto para ingresar datos del huésped
    private JTextField campoNombre, campoApellidos,
        campoDocumentoIdentidad;
    private JLabel fechaIngreso; // Etiqueta de la fecha de ingreso
    private JTextField campoFechaIngreso; /* Campo de texto de la
        fecha de ingreso */
    private int númeroHabitaciónReservada; /* Número de la habitación
        reservada */
    private Hotel hotel; // Objeto Hotel
    private Date fechaInicial; /* Fecha de inicio del alojamiento del
        huésped */
    private Habitación habitaciónReservada; // Habitación reservada

    /**
    * Constructor de la clase VentanaIngreso
    * @param hotel Parámetro que define el hotel con habitaciones y
    * huéspedes
    * @param númeroHabitaciónReservada Parámetro que define el
    * número de la habitación reservada
    */
    public VentanaIngreso(Hotel hotel, int númeroHabitaciónReservada)
        {
        this.hotel = hotel;
        this.númeroHabitaciónReservada = númeroHabitaciónReservada;
        inicio();
        setTitle("Ingreso"); // Establece el título de la ventana
        setSize(290,250); // Establece el tamaño de la ventana
```

```
        setLocationRelativeTo(null); /* La ventana se posiciona en el
            centro de la pantalla */
        setResizable(false); /* Establece que el tamaño de la ventana no
            puede cambiar */
    }

    /**
    * Método que crea la ventana con sus diferentes componentes
    * gráficos
    */
    private void inicio() {
        contenedor = getContentPane(); /* Obtiene el panel de
            contenidos de la ventana */
        // Establece que el contenedor tendrá un GridBagLayout
        contenedor.setLayout(new GridBagLayout());
        GridBagConstraints c = new GridBagConstraints(); /* Define las
            restricciones del layout */
        c.fill = GridBagConstraints.HORIZONTAL; /* El layout es
            horizontal */
        c.insets = new Insets(3,3,3,3); /* Define los bordes del
            contenedor */

        // Establece la etiqueta del número de la habitación reservada
        habitación = new JLabel();
        habitación.setText("Habitación: " +
            númeroHabitaciónReservada);
        /* Localización de la etiqueta del número de la habitación en el
            layout (0,0) */
        c.gridx = 0;
        c.gridy = 0;
        contenedor.add(habitación, c); /* Se agrega la etiqueta al
            contenedor de la ventana */

        // Establece la etiqueta y campo de texto de la fecha de ingreso
        fechaIngreso = new JLabel();
        fechaIngreso.setText("Fecha (aaaa-mm-dd):");
        // Localización de la etiqueta fecha de ingreso en el layout (0,1)
        c.gridx = 0;
        c.gridy = 1;
        /* Se agrega la etiqueta de fecha de ingreso al contenedor de la
            ventana */
        contenedor.add(fechaIngreso, c);
        campoFechaIngreso = new JTextField();
```

```
/* Localización del campo de texto de fecha de ingreso en el
    layout (1,1) */
c.gridx = 1;
c.gridy = 1;
/* Se agrega el campo de texto fecha de ingreso al contenedor de
    la ventana */
contenedor.add(campoFechaIngreso, c);

// Establece la etiqueta del huésped
huésped = new JLabel();
huésped.setText("Huésped");
// Localización de la etiqueta huésped en el layout (0,2)
c.gridx = 0;
c.gridy = 2;
contenedor.add(huésped, c); /* Se agrega la etiqueta huésped al
    contenedor de la ventana */

// Establece la etiqueta y el campo de texto nombre del huésped
nombre = new JLabel();
nombre.setText("Nombre: ");
// Localización de la etiqueta nombre en el layout (0,3)
c.gridx = 0;
c.gridy = 3;
contenedor.add(nombre, c); /* Se agrega la etiqueta nombre al
    contenedor de la ventana */
campoNombre = new JTextField();
// Localización del campo de texto nombre en el layout
c.gridx = 1;
c.gridy = 3;
// Se agrega el campo de texto nombre al contenedor de la ventana
contenedor.add(campoNombre, c);

// Establece la etiqueta y el campo de texto apellidos del huésped
apellidos = new JLabel();
apellidos.setText("Apellidos: ");
// Localización de la etiqueta apellidos en el layout (0,4)
contenedor.add(apellidos, c); /* Se agrega la etiqueta apellidos al
    contenedor de la ventana */
campoApellidos = new JTextField();
// Localización del campo de texto apellidos en el layout (1,4)
c.gridx = 0;
c.gridy = 4;
```

```
c.gridx = 1;
c.gridy = 4;
// Se agrega el campo de texto apellidos al contenedor de la ventana
contenedor.add(campoApellidos, c);

/* Establece la etiqueta y el campo de texto documento de
   identidad del huésped */
documentoIdentidad = new JLabel();
documentoIdentidad.setText("Doc. Identidad: ");
/* Localización de la etiqueta documento de identidad en el
   layout (0,5) */
c.gridx = 0;
c.gridy = 5;
/* Se agrega la etiqueta documento de identidad al contenedor
   de la ventana */
contenedor.add(documentoIdentidad, c);
campoDocumentoIdentidad = new JTextField();
/* Localización del campo de texto documento de identidad en
   el layout (1,5) */
c.gridx = 1;
c.gridy = 5;
/* Se agrega el campo de texto documento de identidad al
   contenedor de la ventana */
contenedor.add(campoDocumentoIdentidad, c);

// Establece el botón aceptar y cancelar de la ventana
aceptar = new JButton("Aceptar");
// Localización del botón aceptar en el layout (0,6)
c.gridx = 0;
c.gridy = 6;
contenedor.add(aceptar, c); /* Se agrega el botón aceptar al
   contenedor de la ventana */
/* Agrega al botón un ActionListener para que gestione eventos
   del botón */
aceptar.addActionListener(this);
cancelar = new JButton("Cancelar");
// Localización del botón cancelar en el layout (1,6)
c.gridx = 1;
c.gridy = 6;
contenedor.add(cancelar, c); /* Se agrega el botón cancelar al
   contenedor de la ventana */
```

```
        /* Agrega al botón un ActionListener para que gestione eventos
           del botón */
        cancelar.addActionListener(this);
    }

    /**
    * Método que gestiona los eventos generados en la ventana de
    * ingreso de huéspedes
    * @throws ParseException Excepción generada cuando la fecha no
    * está en el formato solicitado
    * @throws Exception Excepción generada cuando hay un campo nulo
    * o error en formato de numero
    */
    @Override
    public void actionPerformed(ActionEvent evento) {
        if (evento.getSource() == aceptar) { // Se pulsa el botón Aceptar
            int posición = -1;
            for (int i = 0; i < hotel.listaHabitaciones.size(); i++) { /* Se
                recorre el vector de habitaciones */
                // Se obtiene un elemento del vector
                Habitación habitación = (Habitación) hotel.
                    listaHabitaciones.elementAt(i);
                if (habitación.getNúmeroHabitación() == this.
                    númeroHabitaciónReservada) {
                    // Si la habitación está reservada
                    try {
                        posición = i;
                        // Obtiene la fecha de ingreso tecleada
                        String fechaIngresada = campoFechaIngreso.
                            getText();
                        // Establece formato de fecha
                        SimpleDateFormat formatoFecha = new
                            SimpleDateFormat("yyyy-MM-dd");
                        // Convierte la fecha de ingreso al formato
                        Date fecha = formatoFecha.parse(fechaIngresada);
                        //Crea un objeto huésped
                        Huésped huésped = new Huésped(nombre.
                            getText(), apellidos.getText(),
                        Integer.parseInt(campoDocumentoIdentidad.
                            getText()));
```

```
                        huésped.setFechaIngreso(fecha); /* Establece fecha
                            de ingreso del huésped */
                        habitación.setHuésped(huésped); /* Establece el
                            huésped para la habitación */
                        habitación.setDisponible(false); /* Coloca la
                            habitación como no disponible */
                        habitaciónReservada = habitación;
                        /* Coloca la habitación como ocupada en la lista
                            de habitaciones */
                        hotel.listaHabitaciones.set(posición, habitación);
                        JOptionPane.showMessageDialog(this,"El huésped
                            ha sido registrado","Mensaje", JOptionPane.
                            INFORMATION_MESSAGE,null);
                        setVisible(false); /* La ventana de ingreso no está
                            visible */
                        break;
                    } catch (ParseException e) { /* Si la fecha no está en el
                        formato indicado */
                        JOptionPane.showMessageDialog(this,"La fecha
                            no está en el formato solicitado", "Mensaje",
                            JOptionPane.ERROR_MESSAGE);
                    } catch (Exception e) { /* Si ocurre un error por datos
                        ingresados nulos o no numéricos */
                        JOptionPane.showMessageDialog(this,"Campo
                            nulo o error en formato de numero",
                        "Error", JOptionPane.ERROR_MESSAGE);
                    }
                }
            }
        }
        if (evento.getSource() == cancelar) { /* Si se pulsa el botón
            cancelar */
            setVisible(false); // La ventana de ingreso no está visible
        }
    }
}
```

Clase: VentanaSalida

```
package Hotel;

import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.text.SimpleDateFormat;
import java.text.ParseException;
import java.util.*;

/**
 * Esta clase denominada VentanaSalida define una ventana que permite
 * registrar la salida de un huésped y su pago correspondiente de
 * acuerdo al número de días de alojamiento.
 * @version 1.2/2020
 */
public class VentanaSalida extends JFrame implements ActionListener {
    private Container contenedor; /* Un contenedor de elementos
        gráficos*/
    private JLabel habitación; // Etiqueta de habitación ocupada
    // Etiquetas de fecha de ingreso, de salida y días de alojamiento
    private JLabel fechaIngreso, fechaSalida, cantidadDías;
    private JTextField campoFechaSalida; /* Campo de texto para
        ingresar fecha de salida */
    private JLabel totalPago; // Etiqueta del total a pagar por alojamiento
    /* Botón para calcular el valor a pagar y para registrar la salida del
        huésped */
    private JButton calcular, registrarSalida;
    private Hotel hotel; // Objeto Hotel
    private int númeroHabitación; // Número de la habitación ocupada
    private int posiciónHabitación; /* Posición de la habitación en el
        vector de habitaciones */
    private Habitación habitaciónOcupada; /* Habitación ocupada por
        el huésped */

    /**
    * Constructor de la clase VentanaSalida
    * @param hotel Parámetro que define el hotel con habitaciones y
    * huéspedes
```

```
     * @param número Parámetro que define el número de habitación
     * ocupada
     */
    public VentanaSalida(Hotel hotel, int número) {
        this.hotel = hotel;
        this.númeroHabitación = número;
        inicio();
        setTitle("Salida huéspedes"); // Establece el título de la ventana
        setSize(260,260); // Establece el tamaño de la ventana
        setLocationRelativeTo(null); /* La ventana se posiciona en el
            centro de la pantalla */
        setResizable(false); /* Establece que el tamaño de la ventana no
            puede cambiar */
    }

    /**
     * Método que crea la ventana con sus diferentes componentes
     * gráficos
     */
    private void inicio() {
        contenedor = getContentPane(); /* Obtiene el panel de
            contenidos de la ventana */
        // Establece que el contenedor tendrá un GridBagLayout
        contenedor.setLayout(new GridBagLayout());
        GridBagConstraints c = new GridBagConstraints(); /* Define las
            restricciones del layout */
        c.fill = GridBagConstraints.HORIZONTAL; /* El layout es
            horizontal */
        c.insets = new Insets(3,3,3,3); /* Define los bordes del
            contenedor */

        // Establece la etiqueta de la habitación
        habitación = new JLabel();
        habitación.setText("Habitación: " + númeroHabitación);
        // Localización de la etiqueta habitación en el layout (0,0)
        c.gridx = 0;
        c.gridy = 0;
        contenedor.add(habitación); /* Se agrega la etiqueta habitación al
            contenedor de la ventana */

        // Obtiene la fecha de ingreso para un número de habitación
        String fecha = hotel.
            buscarFechaIngresoHabitación(númeroHabitación);
```

```
fechaIngreso = new JLabel();
// Concatena la etiqueta con la fecha de ingreso obtenida
fechaIngreso.setText("Fecha de ingreso: " + fecha);
// Localización de la etiqueta fecha de ingreso en el layout (0,1)
c.gridx = 0;
c.gridy = 1;
/* Se agrega la etiqueta fecha de ingreso al contenedor de la
    ventana */
contenedor.add(fechaIngreso, c);

/* Establece la etiqueta y campo de texto de la fecha de salida de
    la habitación */
fechaSalida = new JLabel();
fechaSalida.setText("Fecha de salida (aaaa-mm-dd): ");
// Localización de la etiqueta fecha de salida en el layout (0,2)
c.gridx = 0;
c.gridy = 2;
/* Se agrega la etiqueta fecha de salida al contenedor de la
    ventana */
contenedor.add(fechaSalida, c);
campoFechaSalida = new JTextField();
/* Localización del campo de texto fecha de salida en el layout
    (0,3) */
c.gridx = 0;
c.gridy = 3;
/* Se agrega el campo de texto de fecha de salida al contenedor
    de la ventana */
contenedor.add(campoFechaSalida, c);

// Establece el botón calcular
calcular = new JButton("Calcular");
// Localización del botón en el layout (0,4)
c.gridx = 0;
c.gridy = 4;
contenedor.add(calcular, c); /* Se agrega el botón calcular al
    contenedor de la ventana */
/* Agrega al botón un ActionListener para que gestione eventos
    del botón */
calcular.addActionListener(this);

// Establece la etiqueta cantidad de días de alojamiento
cantidadDías = new JLabel();
```

```
        cantidadDías.setText("Cantidad de días: ");
        // Localización de la etiqueta cantidad de días en el layout (0,5)
        c.gridx = 0;
        c.gridy = 5;
        /* Se agrega la etiqueta cantidad de días al contenedor de la
           ventana */
        contenedor.add(cantidadDías, c);

        // Establece la etiqueta de total a pagar
        totalPago = new JLabel();
        totalPago.setText("Total: $");
        // Localización de la etiqueta total a pagar en el layout (0,6)
        c.gridx = 0;
        c.gridy = 6;
        // Se agrega la etiqueta total a pagar al contenedor de la ventana
        contenedor.add(totalPago, c);

        // Establece el botón registrar salida del huésped
        registrarSalida = new JButton("RegistrarSalida");
        // Localización del botón registrar salida en el layout (0,7)
        c.gridx = 0;
        c.gridy = 7;
        // Se agrega el botón registrar salida al contenedor de la ventana
        contenedor.add(registrarSalida, c);
        /* Deshabilita inicialmente el botón de registro de salida de
           huésped */
        registrarSalida.setEnabled(false);
        /* Agrega al botón un ActionListener para que gestione eventos
           del botón */
        registrarSalida.addActionListener(this);
    }

    /**
    * Método que gestiona los eventos generados en la ventana de salida
    * de huéspedes
    */
    @Override
    public void actionPerformed(ActionEvent evento) {
        if (evento.getSource() == calcular) { // Se pulsa el botón Calcular
            String fechaS = campoFechaSalida.getText(); /* Se obtiene la
               fecha de salida ingresada */
            // Se busca la fecha de ingreso según su número de habitación
            String fechaI = hotel.
                buscarFechaIngresoHabitación(númeroHabitación);
```

```
for (int i = 0; i < hotel.listaHabitaciones.size(); i++) { /* Se
    recorre el vector de habitaciones */
    // Se obtiene un elemento del vector de habitaciones
    habitaciónOcupada = (Habitación) hotel.
        listaHabitaciones.elementAt(i);
    if (habitaciónOcupada.getNúmeroHabitación() == this.
        númeroHabitación) {
        // Se encuentra la habitación buscada
        try {
            posiciónHabitación = i;
            SimpleDateFormat formatoFecha = new
                SimpleDateFormat("yyyy-MM-dd");
            Date fecha2 = formatoFecha.parse(fechaS); /* Da
                formato a la fecha de salida */
            /* Establece la fecha de salida del huésped de una
                habitación */
            habitaciónOcupada.getHuésped().
                setFechaSalida(fecha2);
            // Obtiene la fecha de ingreso del huésped
            Date fecha1 = habitaciónOcupada.getHuésped().
                getFechaIngreso();
            /* Si la fecha de ingreso es menor a la fecha de
                salida */
            if (fecha1.compareTo(fecha2) < 0) {
                // Calcula cantidad de días de alojamiento
                int cantidad = habitaciónOcupada.
                    getHuésped().obtenerDíasAlojamiento();
                // Actualiza la etiqueta de cantidad de días
                cantidadDías.setText("Cantidad de días: " +
                    cantidad);
                // Calcula el total a pagar
                double valor = cantidad * habitaciónOcupada.
                    getPrecioDía();
                totalPago.setText("Total: $" + valor);
                    // Actualiza la etiqueta de total a pagar
                registrarSalida.setEnabled(true); /* Habilita el
                    botón de registrar salida del huésped */
            } else {
                /* Si la fecha de ingreso es mayor que la de
                    salida, se genera mensaje de error */
                JOptionPane.showMessageDialog(this,"La fecha
                    de salida es menor que la de
```

```
                                ingreso","Mensaje", JOptionPane.ERROR_
                                    MESSAGE);
                            }
                        } catch (ParseException e) {
                            /* Si la fecha no está en el formato indicado, se
                                genera mensaje de error */
                            JOptionPane.showMessageDialog(this,"La fecha
                                no está en el formato solicitado","Mensaje",
                                JOptionPane.ERROR_MESSAGE);
                        }
                    }
                }
            }
            if (evento.getSource() == registrarSalida) { /* Si se pulsa el botón
                de Registrar salida */
                habitaciónOcupada.setHuésped(null); /* La habitación
                    liberada no tendrá huésped */
                habitaciónOcupada.setDisponible(true); /* La habitación se
                    coloca como disponible */
                // Se actualiza el vector de habitaciones
                hotel.listaHabitaciones.set(posiciónHabitación,
                    habitaciónOcupada);
                // Se muestra mensaje de confirmación
                JOptionPane.showMessageDialog(this,"Se ha registrado
                    la salida del huésped","Mensaje", JOptionPane.
                    INFORMATION_MESSAGE,null);
                setVisible(false); /* La ventana de registro de salida no se
                    muestra */
            }
        }
    }
```

Clase: Principal

```
package Hotel;

/**
 * Esta clase define el punto de ingreso al programa de ingreso y salida
 * de huéspedes de un hotel. Por lo tanto, cuenta con un método main
 * de acceso al programa.
```

```
 * @version 1.2/2020
 */
public class Principal {

    /**
     * Método main que sirve de punto de entrada al programa
     */
    public static void main(String[] args) {
        VentanaPrincipal miVentanaPrincipal; // Define la ventana principal
        Hotel hotel = new Hotel(); // Crea un objeto Hotel
        miVentanaPrincipal= new VentanaPrincipal(hotel); /* Crea la
            ventana principal */
        miVentanaPrincipal.setVisible(true); /* Establece la ventana
            como visible */
    }
}
```

Diagrama de clases

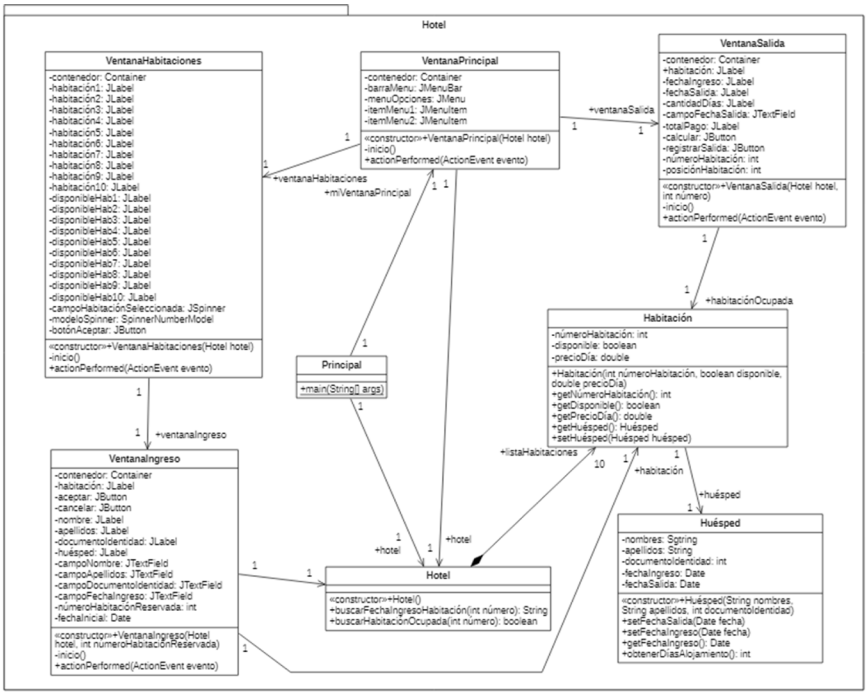


Figura 8.13. Diagrama de clases del ejercicio 8.5.

Explicación del diagrama de clases

Se ha definido un paquete denominado "Hotel" que incluye un conjunto de clases. El punto de entrada al programa es la clase Principal que cuenta con el método *main*. La clase Principal está relacionada mediante una relación de asociación con las clases VentanaPrincipal y Hotel.

La clase Hotel tiene un constructor y métodos para buscar la fecha de ingreso de una habitación a partir de su número de habitación y determinar si una habitación está ocupada a partir de su número de habitación. La clase Hotel está compuesta por diez habitaciones, lo cual se presenta por medio de la relación de composición (línea continua entre Hotel y Habitación con un rombo negro adyacente a la clase Hotel que es el todo y la habitación es la parte en la relación de composición).

La clase Habitación modela una habitación concreta del hotel que cuenta con atributos como su número de habitación, su disponibilidad y precio por día. Además, cuenta con un constructor y métodos *get* y *set* para cada atributo. La clase Habitación está asociada con la clase Huésped por medio de una relación de asociación. Según la multiplicidad de esta relación, una habitación puede ser ocupada por un único huésped. Un huésped tiene como atributos su nombre, apellidos, documento de identidad, fecha de ingreso y fecha de salida. La clase Huésped posee su constructor y métodos *get* y *set* para cada atributo. También tiene un método para calcular la cantidad de días que se alojó el huésped en la habitación.

La clase VentanaPrincipal posee atributos privados para crear una ventana gráfica que cuenta con un contenedor de componentes gráficos (*Container*), una barra de menú (*JMenuBar*), una opción de menú (*JMenú*) y dos opciones de menú (*JMenuItem*). La clase VentanaPrincipal cuenta con un constructor y métodos para generar la ventana gráfica con sus componentes (inicio) y para gestionar los diferentes eventos surgidos al interactuar con esta ventana (*actionPerformed*).

La clase VentanaPrincipal se relaciona con dos clases: VentanaHabitaciones y VentanaSalida, a través de los eventos generados al pulsar los dos ítems de menú. La clase VentanaHabitaciones genera una ventana gráfica que muestra diez habitaciones junto con su disponibilidad. En esta ventana se debe ingresar el número de la habitación a ocupar. Para ello, la ventana define como atributos un contenedor de componentes gráficos (*Container*), diferentes etiquetas para identificar las habitaciones y su disponibilidad (*JLabel*), un selector numérico de la habitación a ocupar

(*JSpinner* y *SpinnerNumberModel*) y un botón para aceptar el ingreso del número de habitación. La clase VentanaHabitaciones cuenta con su constructor y métodos para generar la ventana gráfica con sus componentes (inicio) y para gestionar la pulsación del botón (*actionPerformed*).

En primer lugar, la clase VentanaHabitaciones se vincula con la clase VentanaIngreso que permite ingresar los datos del huésped que ocupará la habitación. Para ello, la clase VentanaIngreso define como atributos un contenedor de componentes gráficos (*Container*), diferentes etiquetas para identificar la habitación y los campos a ingresar (*JLabel*), campos de texto para ingresar los datos del huésped (*JTextField*) y atributos para identificar la habitación a ocupar y la fecha de ingreso. Además, la clase VentanaHabitaciones posee un constructor y métodos para generar la ventana gráfica con sus componentes (inicio) y para gestionar la pulsación del botón (*actionPerformed*).

En segundo lugar, la clase VentanaSalida genera una ventana gráfica que permite registrar la entrega de una habitación y la salida de un huésped. Para ello, la clase cuenta con atributos como un contenedor de componentes gráficos (*Container*), diferentes etiquetas para identificar los campos a ingresar y calcular (*JLabel*), campos de texto para ingresar datos (*JTextField*) y botones para calcular el total a pagar, registrar la salida del huésped y liberar la habitación (*JButton*). Además, la clase VentanaSalida posee un constructor y métodos para generar la ventana gráfica con sus componentes (inicio) y gestionar la pulsación del botón de calcular pago y registrar salida (*actionPerformed*).

Diagrama de objetos

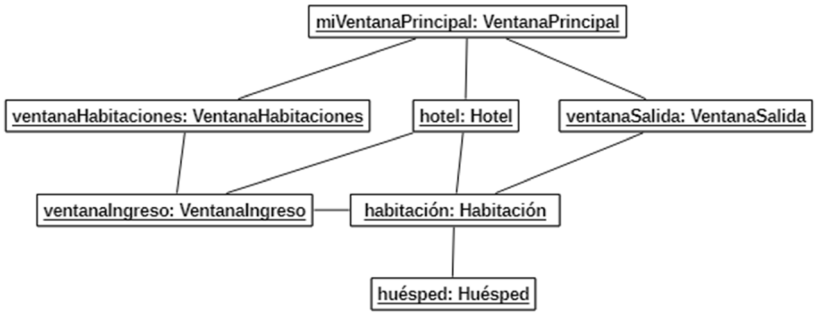
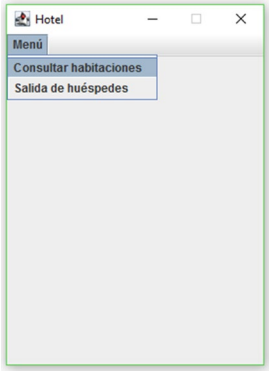
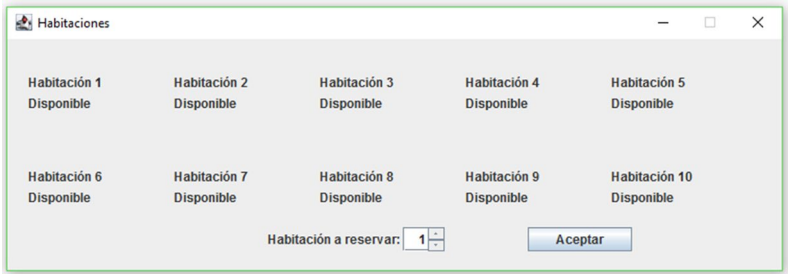


Figura 8.14. Diagrama de objetos del ejercicio 8.5.

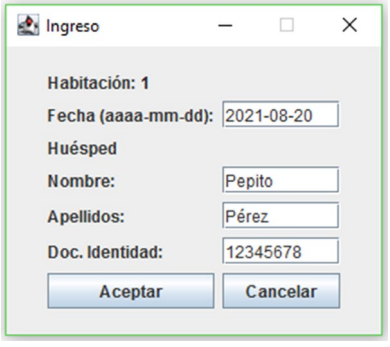
Ejecución del programa



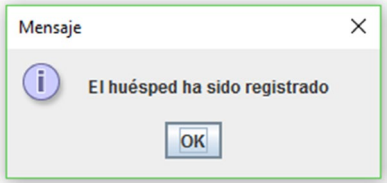
a) Menú principal



b) Selección de la habitación a ocupar

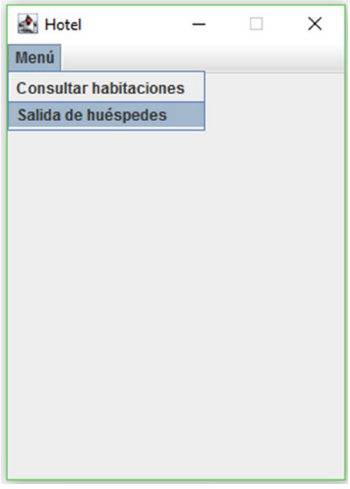


c) Ingreso del huésped

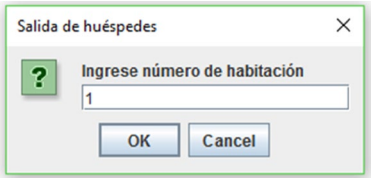


d) Confirmación del registro del huésped

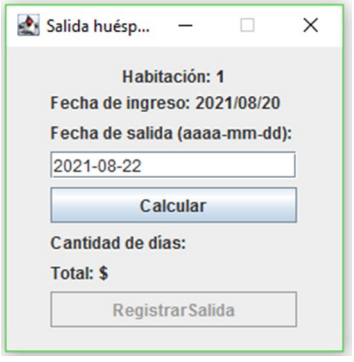
e) La habitación seleccionada aparece como "No disponible"



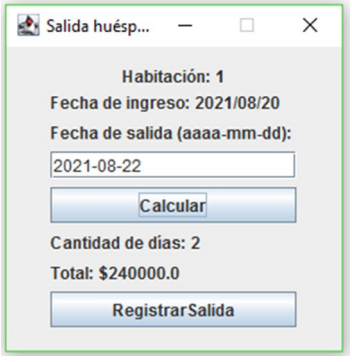
f) Menú salida de huésped



g) Ingresar número de habitación



h) Ingreso de la fecha de salida



i) Cálculo del total a pagar

Figura 8.15. Ejecución del programa del ejercicio 8.5.

Ejercicios propuestos

Modificar el programa del hotel para que soporte las siguientes funcionalidades:

- Las habitaciones pueden ser simples, dobles y triples. Cada tipo de habitación tiene un precio distinto. Cuando se registra el ingreso a una habitación se ingresan los datos de los huéspedes correspondientes.
- Se requiere de un informe histórico de los tiempos ocupados por cada habitación.
- Se requiere de un informe histórico de las habitaciones ocupadas por un determinado huésped.

Capítulo 9
JavaFX

El propósito general del noveno capítulo es realizar una descripción muy general de JavaFX, un marco de código abierto basado en Java para desarrollar aplicaciones de cliente enriquecido. JavaFX también es visto como el sucesor de *Swing* en el campo de la tecnología de desarrollo de interfaz gráfica de usuario (GUI) en la plataforma Java. En este capítulo se presentan siete ejercicios sobre diferentes clases que conforman estructuras de programación en JavaFX.

- **Ejercicio 9.1. Escenarios**

  Las aplicaciones JavaFX están conformadas por elementos denominados escenarios que incluyen escenas donde se incorporan los diferentes componentes gráficos. El primer ejercicio propone el desarrollo de una ventana que tiene un escenario, una escena y diferentes componentes gráficos.

- **Ejercicio 9.2. Componentes gráficos**

  Las aplicaciones JavaFX constan de numerosos tipos de componentes gráficos que tienen diversas funcionalidades y propósitos de acuerdo con los objetivos del usuario final. El segundo ejercicio plantea el desarrollo de una ventana que incorpore etiquetas, campos de texto, botones e imágenes.

- **Ejercicio 9.3. Figuras 2D**
  JavaFX incluye una muy completa y heterogénea colección de clases y métodos orientadas a construir figuras en dos dimensiones. El objetivo del tercer ejercicio es conocer y presentar en ventanas gráficas las diferentes formas bidimensionales que se pueden dibujar utilizando JavaFX.

- **Ejercicio 9.4. Figuras 3D**
  JavaFX también incluye un paquete para el desarrollo de figuras en tres dimensiones. El cuarto ejercicio plantea el desarrollo de una ventana gráfica con diversos objetos tridimensionales a los cuales se les debe agregar una textura y una fuente de luz.

- **Ejercicio 9.5. Transformaciones**
  Los objetos gráficos generalmente no se mantienen estáticos y son manipulados cambiando su posición y apariencia. El quinto ejercicio plantea un problema: definir un objeto gráfico y realizarle algunas transformaciones como: traslado, rotación, corte y escalamiento.

- **Ejercicio 9.6. Animaciones**
  Las aplicaciones actuales requieren que los objetos se trasladen de un sitio de la pantalla a otro. JavaFX permite generar animaciones mediante el establecimiento de líneas de tiempo y diferentes transiciones de objetos. El sexto ejercicio plantea un problema para desarrollar dos pequeñas animaciones utilizando las transiciones predefinidas de JavaFX.

- **Ejercicio 9.7. Gráficas**
  Actualmente, el análisis de datos es un área de conocimiento de alto impacto y con muchas aplicaciones. JavaFX incorpora un paquete específico que permite generar en forma rápida diferentes tipos de gráficos con impacto visual, fácil construcción y adaptación. Este último ejercicio del capítulo propone el uso y aplicación de un tipo específico de gráfica para presentar datos en pantalla.

Ejercicio 9.1. Escenarios

Con JavaFX las ventanas principales son instancias de la clase *Application*, la cual pertenece al paquete *javafx*. Para iniciar una aplicación se debe invocar al método *launch()* desde el método *main*. Al iniciarse, la aplicación llamará al método *start()*. Por lo general, el método *start()* crea y muestra una ventana de algún tipo (Sharan, 2015).

También se deben definir los métodos *init()* y *stop()*. El método *init()* es automáticamente invocado por la aplicación para inicializar objetos antes que la aplicación inicie. Cuando la aplicación termina, la aplicación llamará inmediatamente al método *stop()* para cerrar archivos, liberar recursos, etc.

```
import javafx.application.Application;
import javafx.stage.Stage;
public class MyApplication extends Application {
        public void init() { ... }
        public void start(Stage escenario) { ... }
        public void stop() { ... }
        public static void main(String[] args) {
                launch(args);
        }
}
```

El método *start()* cuenta con un único parámetro de tipo *stage*, el cual es un objeto que representa un escenario (un tipo especial de ventana). Para hacer algo visible, se debe crear un objeto *Scene*, que define una escena (la apariencia de la aplicación). El objeto *Scene* contiene componentes gráficos denominados nodos y agrupados en contenedores que pertenecen a la clase *Pane*.

Un evento es algo que sucede en el programa en función de algún tipo de entrada que, generalmente, es causada por la interacción del usuario, como presionar una tecla en el teclado o hacer clic con el ratón. El origen de un evento es el componente para el que se generó el evento, es decir, cuando se gestionan clics de un botón, el componente botón es el origen. Un controlador de eventos es un procedimiento que contiene el código que se ejecutará cuando ocurra un tipo específico de evento en el programa (Dea *et al.*, 2014).

Estos eventos se generan cuando el usuario interactúa con la GUI de la siguiente manera:

1. El usuario provoca un evento haciendo clic en un botón, presionando una tecla, seleccionando un elemento de la lista, etc.
2. Se generan eventos.
3. Se llama al controlador de eventos apropiado.
4. El código de manejo de eventos cambia el modelo (estados de los objetos) de alguna manera.
5. La interfaz de usuario se actualiza para reflejar estos cambios en el modelo.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Crear una ventana gráfica utilizando JavaFX.
- Incorporar componentes gráficos a los contenedores de una ventana gráfica.

Enunciado: Contactos

Se requiere desarrollar un programa que permita (en una ventana) ingresar los datos de un contacto en una agenda personal. Los datos de un contacto son:

- Nombres y apellidos: se ingresan en objetos TextField independientes.
- Fecha de nacimiento: se debe desplegar un calendario utilizando la clase *DatePicker*.
- Dirección, teléfono, correo electrónico: se ingresa en un *TextField*.

Una vez se ingresan los datos, se debe oprimir un botón denominado "Agregar" que permite que el contacto se añada a una lista (clase *ListView*) ubicada en la parte inferior de la ventana.

Instrucciones Java del ejercicio

Tabla 9.1. Instrucciones Java del ejercicio 9.1.

Clase	Método	Descripción
Label	*Label()*	Constructor de la clase *Label*.
TextField	*TextField()*	Constructor de la clase *TextField*.
	String getText()	Obtiene el valor del texto ingresado.
	void setText(String valor)	Establece el valor del texto.
DatePicker	*DatePicker()*	Constructor de la clase *DatePicker*.
	LocalDate getValue()	Obtiene el valor seleccionado.
	void setValue(T valor)	Establece el valor del objeto.
ListView	*ListView()*	Constructor de la clase *ListView*.
	ObservableList<T> getItems()	Retorna una lista que contiene los elementos mostrados al usuario.
	void getItems().add(String valor)	Añade valores a la lista.
Button	*Button()*	Constructor de la clase *Button*.
	void setMaxWidth(Double.MAX_VALUE)	Establece la anchura del botón.
GridPane	*GridPane()*	Constructor de la clase *GridPane*.
	void setHgap(double valor)	Establece espacio horizontal entre componentes.
	void setVgap(double valor)	Establece espacio vertical entre componentes.
	void add(Component componente, fila, columna)	Añade un componente en la fila y columna especificada.
	void setStyle(String valor)	Establece un estilo CSS para el panel.
VBox	*VBox()*	Constructor de la clase VBox.
Stage	*Stage(Parent root, double anchura, double altura)*	Crea un objeto *Scene* para un nodo específico con un tamaño específico.
	void setScene(Scene valor)	Especifica la escena utilizada en este escenario.
	void setTitle()	Establece el título del escenario.
	void sizeToScene()	Establece el tamaño del escenario.
	void show()	Muestra la ventana.
Alert	*void setTitle(String valor)*	Establece título de la ventana del cuadro de diálogo.
	void setHeaderText(String valor)	Establece la cabecera del mensaje.
	void setContentText(String valor)	Establece el contenido de la ventana.
	void showAndWait()	Muestra el cuadro de diálogo.
Component	*setOnAction()*	Gestiona eventos de un componente.

Solución

Clase: VentanaContacto

```
package ventanacontacto;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.DatePicker;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.VBox;
import javafx.scene.control.ListView;
import java.time.LocalDate;
import javafx.scene.control.Alert;
import javafx.stage.Stage;

/**
 * Esta clase denominada VentanaContacto crea una ventana que
 * permite agregar un contacto.
 * @version 1.0/2020
 */
public class VentanaContacto extends Application {
    // Etiquetas
    Label nombres = new Label("Nombres:");
    Label apellidos = new Label("Apellidos:");
    Label fechaNacimiento = new Label("Fecha nacimiento:");
    Label dirección = new Label("Dirección");
    Label correo = new Label("Correo");
    Label teléfono = new Label("Teléfono");
    // Campos de texto
    TextField campoNombres = new TextField();
    TextField campoApellidos = new TextField();
    DatePicker campoFechaNacimiento = new DatePicker();
    // Componente gráfico calendario
    TextField campoDirección = new TextField();
    TextField campoCorreo = new TextField();
    TextField campoTeléfono = new TextField();
    ListView lista = new ListView(); /* Componente gráfico con lista de
        elementos */
```

```
// Botón
Button agregar = new Button("Agregar");

/**
* Método main que lanza la aplicación
* @param args Parámetro que define los argumentos de la aplicación
*/
public static void main(String[] args) {
    Application.launch(args);
}

/**
* Método start que inicia la aplicación
* @param stage El escenario donde se ejecutará la aplicación
*/
@Override
public void start(Stage stage) throws Exception {
    // Establece un grid para los componentes gráficos
    GridPane grid = new GridPane();
    /* Establece espacios horizontales y verticales entre filas y
        columnas */
    grid.setHgap(5);
    grid.setVgap(5);
    // Coloca los controles en el grid
    grid.add(nombres, 0, 0); // columna=0, fila=0
    grid.add(apellidos, 0, 1); // columna=0, fila=1
    grid.add(fechaNacimiento, 0, 2); // columna=0, fila=2
    grid.add(dirección, 0, 3); // columna=0, fila=3
    grid.add(teléfono, 0, 4); // columna=0, fila=4
    grid.add(correo, 0, 5); // columna=0, fila=5
    grid.add(campoNombres, 1, 0); // columna=1, fila=0
    grid.add(campoApellidos, 1, 1); // columna=1, fila=1
    grid.add(campoFechaNacimiento, 1, 2); // columna=1, fila=2
    grid.add(campoDirección, 1, 3); // columna=1, fila=3
    grid.add(campoTeléfono, 1, 4); // columna=1, fila=4
    grid.add(campoCorreo, 1, 5); // columna=1, fila=5
    grid.add(lista, 2, 0, 1, 7); /* columna=2, fila=0, colspan=1,
        rowspan=7 */
    // Agrega el botón
    VBox buttonBox = new VBox(agregar);
    agregar.setMaxWidth(Double.MAX_VALUE);
    grid.add(buttonBox, 0, 6, 1, 2); /* columna=0, fila=6, colspan=1,
        rowspan=2 */
```

```
        // Muestra los datos en el área de texto al presionar botón salvar
        agregar.setOnAction(e -> mostrarDatos());
        // Establece un CSS para el GridPane
        grid.setStyle("-fx-padding: 10;"
            + "-fx-border-style: solid inside;"
            + "-fx-border-width: 2;"
            + "-fx-border-insets: 5;"
            + "-fx-border-radius: 5;"
            + "-fx-border-color: green;");
        Scene scene = new Scene(grid, 600, 300); /* Crea una escena
            con el tamaño definido */
        stage.setScene(scene); // Establece para el escenario una escena
        stage.setTitle("Detalles del contacto"); /* Establece el título del
            escenario */
        stage.sizeToScene();
        stage.show(); // Muestra el escenario (ventana)
    }

    /**
    * Método que captura los datos ingresados de un contacto, crea un
    * contacto, lo añade a la lista de contactos y a la lista gráfica
    */
    private void mostrarDatos() {
        // Captura los datos ingresados
        String a = campoNombres.getText();
        String b = campoApellidos.getText();
        LocalDate c = campoFechaNacimiento.getValue();
        String d = campoDirección.getText();
        String e = campoTeléfono.getText();
        String f = campoCorreo.getText();
        // Evalua que los campos no estén vacíos
        if (a.equals("") || b.equals("") || d.equals("") ||
                e.equals("") || f.equals("")) {
            // Si los campos están vacíos, se genera una alerta
            Alert mensaje = new Alert(Alert.AlertType.INFORMATION);
            mensaje.setTitle("Mensaje");
            mensaje.setHeaderText("Error en ingreso de datos");
            mensaje.setContentText("No se permiten campos vacíos");
            mensaje.showAndWait();
        } else {
            // Si los datos se han ingresado correctamente
```

```
                Contacto contacto = new Contacto(a,b,c,d,e,f); /*Crea un
                    contaco */
                ListaContactos listaContactos = new ListaContactos(); /* Crea
                    la lista de contactos */
                listaContactos.agregarContacto(contacto); /* Añade el
                    contacto a la lista */
                String data = a + "-" + b + "-" + c + "-" + d + "-" + e + "-" + f;
                lista.getItems().add(data); /* Añade el contacto a la lista
                    gráfica */
                /* Todos los campos quedan vacíos para ingresar un nuevo
                    contacto */
                campoNombres.setText("");
                campoApellidos.setText("");
                campoFechaNacimiento.setValue(null);
                campoDirección.setText("");
                campoTeléfono.setText("");
                campoCorreo.setText("");
            }
        }
    }
```

Clase: Contacto

```
package ventanacontacto;

import java.time.LocalDate;

/**
 * Esta clase denominada Contacto define un contacto para una agenda
 * de contactos.
 * @version 1.0/2020
 */
public class Contacto {
    String nombres; // Atributo que define los nombres de un contacto
    String apellidos; // Atributo que define los apellidos de un contacto
    LocalDate fechaNacimiento; /* Atributo que define la fecha de
        nacimiento de un contacto */
    String dirección; // Atributo que define la dirección de un contacto
    String teléfono; // Atributo que define el teléfono de un contacto
    String correo; // Atributo que define el correo de un contacto
```

```
    /**
     * Constructor de la clase Contacto
     * @param nombres Parámetro que define los nombres de un
     * contacto
     * @param apellidos Parámetro que define los apellidos de un
     * contacto
     * @param fechaNacimiento Parámetro que define la fecha de
     * nacimiento de un contacto
     * @param dirección Parámetro que define la dirección de un
     * contacto
     * @param teléfono Parámetro que define el teléfono de un contacto
     * @param correo Parámetro que define el correo de un contacto
     */
    Contacto(String nombres, String apellidos, LocalDate
        fechaNacimiento, String dirección, String teléfono, String correo)
        {
        this.nombres = nombres;
        this.apellidos = apellidos;
        this.fechaNacimiento = fechaNacimiento;
        this.dirección = dirección;
        this.teléfono = teléfono;
        this.correo = correo;
    }
}
```

Clase: ListaContacto

```
package ventanacontacto;

import java.util.Vector;

/**
 * Esta clase denominada ListaContactos define una lista de objetos de
 * tipo Contacto.
 * @version 1.0/2020
 */
public class ListaContactos {
    Vector lista;
```

```
    /**
     * Constructor de la clase ListaContactos
     */
    ListaContactos() {
        lista = new Vector(); // Crea un vector
    }

    /**
     * Método que agrega un contacto a la lista de contactos
     * @param contacto Parámetro que define el contacto a agregar
     */
    void agregarContacto(Contacto contacto) {
        lista.add(contacto);
    }
}
```

Diagrama de clases

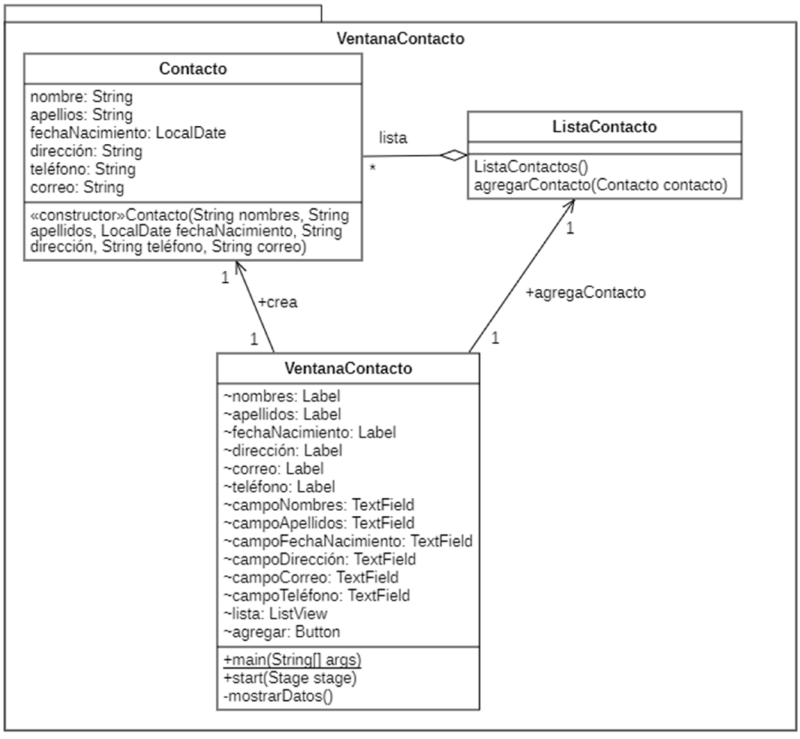


Figura 9.1. Diagrama de clases del ejercicio 9.1.

Explicación del diagrama de clases

El diagrama de clases presenta un solo paquete denominado "VentanaContacto", que contiene tres clases: Contacto, ListaContacto y VentanaContacto. La clase Contacto tiene seis atributos: nombres y apellidos (de tipo *String*), fechaNacimiento (de tipo *LocalDate*), dirección, teléfono y correo (de tipo *String*). También cuenta con un constructor para inicializar estos atributos.

En primer lugar, la clase ListaContactos está relacionada con la clase Contacto por medio de una relación de agregación que se denota con una flecha con un rombo sin relleno en un extremo. Esta relación indica que una lista de contactos tiene asociada una colección de contactos con una multiplicidad de muchos (indicada con el asterisco *). La relación de agregación indica una relación con semántica débil entre las dos clases; de tal manera que, si la lista de contactos se elimina, los contactos siguen existiendo. En el código, esta relación de agregación se presenta como un atributo de la clase ListaContactos denominado lista, como indica el nombre de la relación de agregación.

En segundo lugar, la clase VentanaContacto tiene como atributos todos los componentes gráficos que conforman la ventana: etiquetas (*Label*), campos de ingreso de texto (*TextField*), un botón (*Button*) y una lista (*ListView*). También cuenta con los métodos: *start* que inicia la aplicación gráfica; el método *main* (punto de entrada a la aplicación); y el método mostrarDatos. El método *main* es un método estático, por lo cual se representa con su texto subrayado. La clase VentanaClase está relacionada con las dos clases anteriores ya que, durante su ejecución, se crea un contacto y dicho contacto se agrega a la lista de contactos. Los métodos *start* y *main* son públicos, lo cual se indica con el símbolo +. El método mostrarDatos es privado y se indica con el símbolo -.

Diagrama de objetos

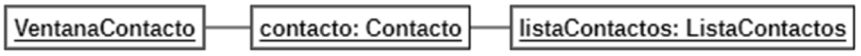
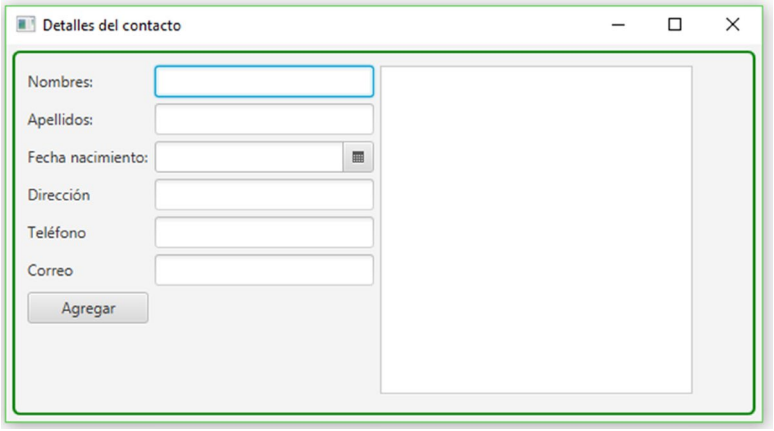
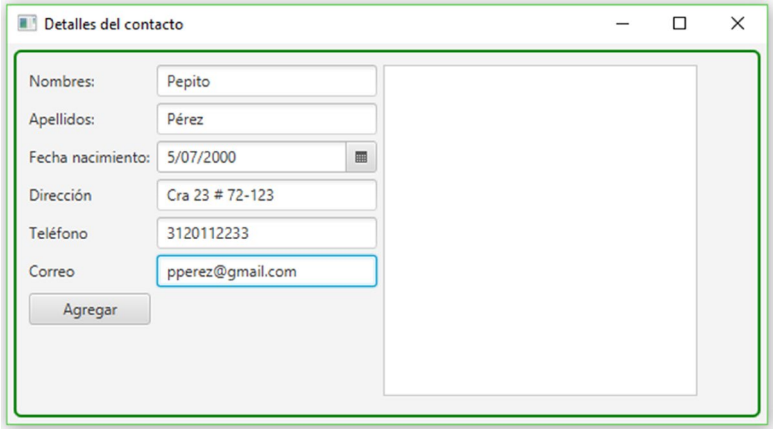


Figura 9.2. Diagrama de objetos del ejercicio 9.1.

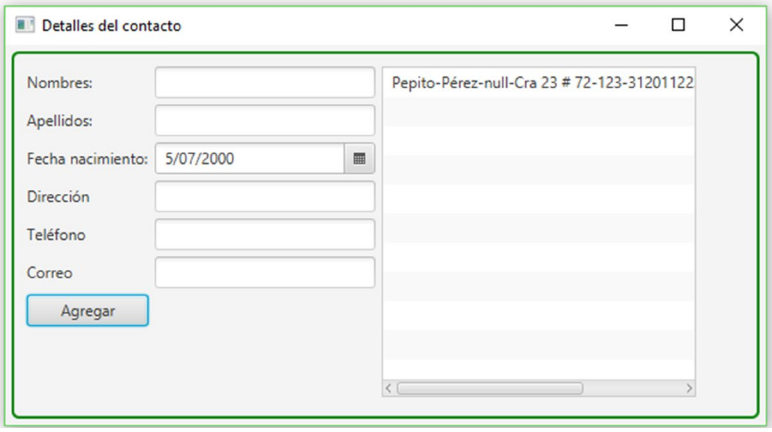
Ejecución del programa



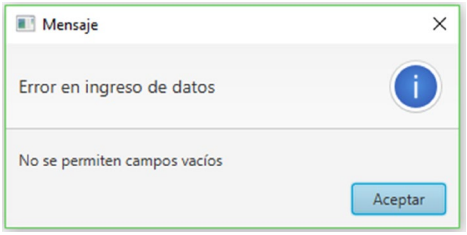
a) Ventana de contacto sin datos



b) Ingreso de datos



c) Contacto agregado a la lista



d) Mensaje de error

Figura 9.3. Ejecución del programa del ejercicio 9.1.

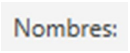
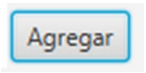
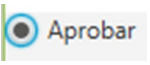
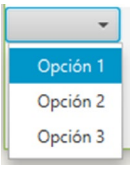
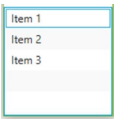
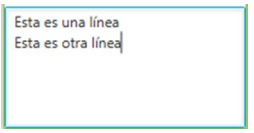
Ejercicios propuestos

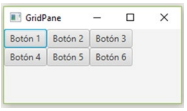
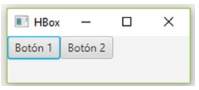
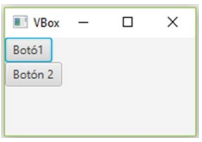
- Realizar un programa que permita ingresar un valor numérico en una ventana y calcular: su logaritmo natural, su logaritmo en base 10, su raíz cuadrada y si es un número primo.
- Realizar un programa que permita convertir grados Fahrenheit a grados Celsius y viceversa utilizando JavaFX.
- Realizar el ejercicio 8.3. (figuras geométricas) utilizando JavaFX.

Ejercicio 9.2. Componentes gráficos

La tabla 9.2 presenta un listado abreviado de diferentes componentes gráficos (nodos) incluidos en JavaFX (API Java, 2020).

Tabla 9.2. Componentes gráficos de JavaFX

Elementos	Descripción	Figura
Label	Control de texto no editable.	
Button	Un simple control de botón. El cual puede contener texto o un gráfico.	
CheckBox	Un control de selección que aparece como un cuadro con una marca de verificación seleccionada o no.	
RadioButton	Serie de elementos donde solo se puede seleccionar un elemento.	
ChoiceBox	Permite a los usuarios elegir una opción de una lista predefinida de opciones.	
ListView	Muestra una lista horizontal o vertical de elementos que el usuario puede seleccionar.	
ScrollBar	Una barra horizontal o vertical con botones de desplazamiento.	
Spinner	Un campo de texto de una sola línea que permite seleccionar un número en una secuencia ordenada.	
TextArea	Componente de entrada de texto que permite al usuario ingresar varias líneas de texto sin formato.	
TextField	Componente de entrada de texto que permite al usuario ingresar una sola línea de texto sin formato.	

Elementos	Descripción	Figura
GridPane	Presenta sus elementos dentro de una cuadrícula flexible de filas y columnas.	
HBox	Presenta sus elementos secundarios en una sola fila horizontal.	
VBox	Presenta sus elementos secundarios en una sola columna vertical.	

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Utilizar *GridPane* para distribuir componentes en una ventana.
- Insertar imágenes en ventanas gráficas.
- Insertar diferentes componentes gráficos en un escenario gráfico.

Enunciado: Banderas

Se requiere desarrollar un programa que muestre en una ventana y en una primera columna el texto de cinco países. En otra columna habrá imágenes de cinco banderas identificadas con un número. El usuario debe ingresar en un campo de texto al lado del país, el número de la bandera que corresponde. El programa debe calcular, en la parte inferior de la ventana, la cantidad de aciertos que obtuvo el usuario.

Instrucciones Java del ejercicio

Tabla 9.3. Instrucciones Java del ejercicio 9.2.

Clase	Método	Descripción
Label	*Label()*	Constructor de la clase *Label*.
	void setText(String valor)	Establece el texto de la etiqueta.
	void setFont(Font valor)	Establece la fuente de la etiqueta.
	void setEffect(Effect valor)	Establece el efecto de la etiqueta.

Clase	Método	Descripción
DropShadow	*DropShadow()*	Constructor de un efecto de alto nivel que genera una sombra del contenido dado.
TextField	*TextField()*	Constructor de la clase *TextField.*
Button	*Button()*	Constructor de la clase *Button*.
GridPane	GridPane()	Constructor de la clase *GridPane*.
	void add(Component componente, fila, columna)	Añade un componente en la fila y columna especificada.
	void setStyle(String valor)	Establece un estilo CSS para el panel.
Image	*Image()*	Constructor de elemento que representa imágenes gráficas.
ImageView	*ImageView()*	Constructor de un nodo utilizado para pintar imágenes cargadas con la clase Image.
	void setX(double valor)	Establece la posición en el eje *x*.
	void setY(double valor)	Establece la posición en el eje *y*.
	void setFitHeight(double valor)	Establece la altura de la imagen.
	void setFitWidth(double valor)	Establece la anchura de la imagen.
	void setPreserveRatio(boolean valor)	Preserva el tamaño de la imagen.
Group	*Group()*	Constructor de un contenedor con una lista de elementos que se representan en orden.
Stage	*Stage(Parent root, double anchura, double altura)*	Crea un objeto *Scene* para un nodo específico con un tamaño específico.
	void setScene(Scene valor)	Especifica la escena utilizada en este escenario.
	void setTitle(String valor)	Establece el título del escenario.
	void sizeToScene()	Establece el tamaño del escenario.
	void show()	Muestra la ventana.
Component	*setOnAction()*	Gestiona eventos de un componente.

Solución

Clase: Banderas

```
package banderas;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import javafx.application.Application;
```

```
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.effect.DropShadow;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.GridPane;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.Stage;

/**
 * Esta clase denominada Banderas muestra 5 banderas para las cuales
 * hay que asignarles el país correspondiente. El programa determinará
 * la cantidad de aciertos logrados.
 * @version 1.0/2020
 */
public class Banderas extends Application {
    // Etiquetas de países
    Label etiqueta = new Label("Seleccione la bandera");
    Label belice = new Label("Belice:");
    Label eslovenia = new Label("Eslovenia:");
    Label sudáfrica = new Label("Sudáfrica:");
    Label bután = new Label("Bután:");
    Label australia = new Label("Australia:");
    // Campos de texto de las respuestas
    TextField respuesta1 = new TextField();
    TextField respuesta2 = new TextField();
    TextField respuesta3 = new TextField();
    TextField respuesta4 = new TextField();
    TextField respuesta5 = new TextField();
    // Etiquetas de números de banderas
    Label uno = new Label("1");
    Label dos = new Label("2");
    Label tres = new Label("3");
    Label cuatro = new Label("4");
    Label cinco = new Label("5");
    // Botón aceptar
    Button aceptar = new Button("Aceptar");
    // Texto de resultados
    Text puntaje = new Text();
```

```
// Matriz de respuestas ingresadas
String[] respuestasIngresadas;

/**
* Método que lanza la aplicación
* @param stage El escenario donde se ejecutará la aplicación
*/
@Override
public void start(Stage stage) throws FileNotFoundException {
    GridPane grid = new GridPane(); /* Define un grid de elementos
        gráficos */
    grid.setHgap(5); // Define espacio entre columnas
    grid.setVgap(5); // Define espacio entre filas

    // Define la imagen 1
    Image banderaAustralia = new Image(new FileInputStream("C:/
        BanderaAustralia.jpg"));
    ImageView imageView1 = new ImageView(banderaAustralia);
    imageView1.setX(50); // Posición x de la imagen 1
    imageView1.setY(25); // Posición y de la imagen 1
    imageView1.setFitHeight(200); // Define altura de la imagen 1
    imageView1.setFitWidth(100);  // Define anchura de la imagen 1
    imageView1.setPreserveRatio(true);  // Preserva radio de la imagen 1

    // Define la imagen 2
    Image banderaSudáfrica = new Image(new FileInputStream("C:/
        BanderaSudafrica.jpg"));
    ImageView imageView2 = new ImageView(banderaSudáfrica);
    imageView2.setX(50);  // Posición x de la imagen 2
    imageView2.setY(125); // Posición y de la imagen 2
    imageView2.setFitHeight(200);  // Define altura de la imagen 2
    imageView2.setFitWidth(100);  // Define anchura de la imagen 2
    imageView2.setPreserveRatio(true);  // Preserva radio de la imagen 2

    // Define la imagen 3
    Image banderaBelice = new Image(new FileInputStream("C:/
        BanderaBelice.jpg"));
    ImageView imageView3 = new ImageView(banderaBelice);
    imageView3.setX(50);  // Posición x de la imagen 3
    imageView3.setY(225);  // Posición y de la imagen 3
    imageView3.setFitHeight(200);  // Define altura de la imagen 3
    imageView3.setFitWidth(100);  // Define anchura de la imagen 3
```

```
imageView3.setPreserveRatio(true);  // Preserva radio de la imagen 3

// Define la imagen 4
Image banderaEslovenia = new Image(new FileInputStream("C:/
    BanderaEslovenia.png"));
ImageView imageView4 = new ImageView(banderaEslovenia);
imageView4.setX(50);  // Posición x de la imagen 4
imageView4.setY(325);  // Posición y de la imagen 4
imageView4.setFitHeight(200);  // Define altura de la imagen 4
imageView4.setFitWidth(100);   // Define anchura de la imagen 4
imageView4.setPreserveRatio(true);  // Preserva radio de la imagen
    4

// Define la imagen 5
Image banderaBután = new Image(new FileInputStream("C:/
    BanderaBután.png"));
ImageView imageView5 = new ImageView(banderaBután);
imageView5.setX(50);  // Posición x de la imagen 5
imageView5.setY(425);  // Posición y de la imagen 5
imageView5.setFitHeight(200);  // Define altura de la imagen 5
imageView5.setFitWidth(100);  // Define anchura de la imagen 5
imageView5.setPreserveRatio(true);  // Preserva radio de la imagen 5

// Agrega los componentes gráficos a un grid
grid.add(etiqueta, 0, 0);
grid.add(belice, 0, 1);
grid.add(respuesta1, 1, 1);
grid.add(uno, 2, 1);
grid.add(imageView1, 3, 1);
grid.add(eslovenia, 0, 2);
grid.add(respuesta2, 1, 2);
grid.add(dos, 2, 2);
grid.add(imageView2, 3, 2);
grid.add(sudáfrica, 0, 3);
grid.add(respuesta3, 1, 3);
grid.add(tres, 2, 3);
grid.add(imageView3, 3, 3);
grid.add(bután, 0, 4);
grid.add(respuesta4, 1, 4);
grid.add(cuatro, 2, 4);
grid.add(imageView4, 3, 4);
grid.add(australia, 0, 5);
```

```
        grid.add(respuesta5, 1, 5);
        grid.add(cinco, 2, 5);
        grid.add(imageView5, 3, 5);
        grid.add(aceptar, 0, 6);
        grid.add(puntaje, 0, 7);

        // Establece evento para el botón
        aceptar.setOnAction(e -> calcular());

        // Establece propiedes CSS para el grid
        grid.setStyle("-fx-padding: 10;"
            + "-fx-border-style: solid inside;"
            + "-fx-border-width: 2;"
            + "-fx-border-insets: 5;"
            + "-fx-border-radius: 5;"
            + "-fx-border-color: green;");

        Group root = new Group(grid); /* Crea grupo de componentes a
            partir del grid */
        Scene scene = new Scene(root, 480, 450); /* Crea escena con
            grupo especificando su tamaño */
        stage.setTitle("Banderas"); // Establece título de la escena
        stage.setScene(scene); // Establece la escena para el escenario
        stage.show(); // Muestra el escenario (ventana)
    }

    /**
    * Método main que lanza la aplicación
    * @param args Parámetro que define los argumentos de la aplicación
    */
    public static void main(String args[]) {
        launch(args);
    }

    /**
    * Método que calcula el puntaje de aciertos obtenidos por el jugador
    */
    private void calcular() {
        respuestasIngresadas = new String[5]; // Crea un array
        // Asigna las respuestas ingresadas al array
        respuestasIngresadas[0] = respuesta1.getText();
        respuestasIngresadas[1] = respuesta2.getText();
        respuestasIngresadas[2] = respuesta3.getText();
```

```
            respuestasIngresadas[3] = respuesta4.getText();
            respuestasIngresadas[4] = respuesta5.getText();
            int total = 0;
            /* Evalúa si las respuestas fueron correctas, si lo son incrementa
                el total de aciertos */
            if (respuestasIngresadas[0].equals("3"))
                total++;
            if (respuestasIngresadas[1].equals("4"))
                total++;
            if (respuestasIngresadas[2].equals("2"))
                total++;
            if (respuestasIngresadas[3].equals("5"))
                total++;
            if (respuestasIngresadas[4].equals("1"))
                total++;
            puntaje.setText("Total : " + total + "/5"); /* Muestra el total de
                aciertos en pantalla */
            puntaje.setFont(Font.font(24)); /* Establece fuente del puntaje
                de aciertos obtenidos */
            puntaje.setEffect(new DropShadow()); /* Establece efecto
                aplicado al texto del puntaje */
        }
    }
```

Diagrama de clases

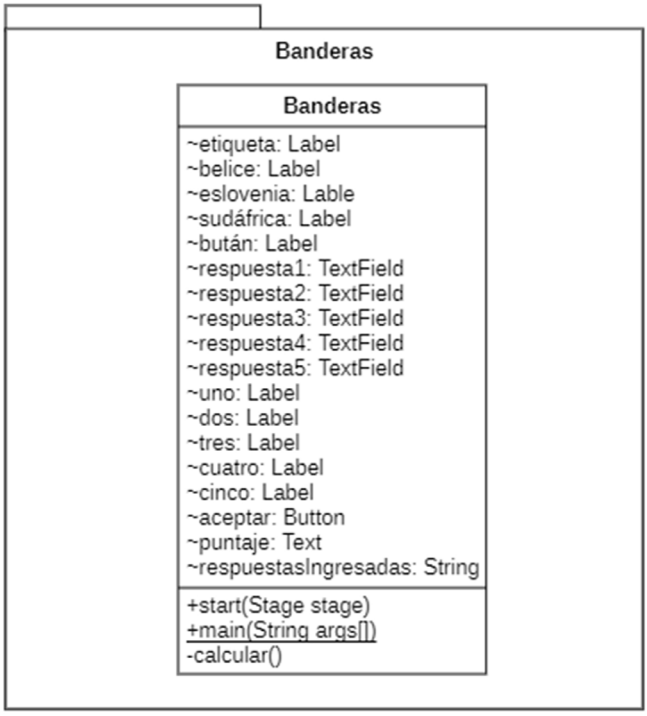


Figura 9.4. Diagrama de clases del ejercicio 9.2.

Explicación del diagrama de clases

El diagrama de clases muestra un solo paquete denominado "Banderas", el cual contiene una sola clase denominada también "Banderas". La clase cuenta con atributos que indican los diferentes componentes gráficos de una ventana gráfica: etiquetas (*Label*), campos de ingreso de texto (*TextField*), un botón (*Button*), un texto (*Text*) y un atributo que almacena las respuestas ingresadas por el usuario como un *array* de *String*.

La clase Banderas también cuenta con los métodos: *start* que inicia la aplicación gráfica; el método *main* (punto de entrada a la aplicación); y el método calcular que obtiene el puntaje de respuestas válidas ingresadas por el usuario. El método *main* es un método estático, por lo cual se representa con su texto subrayado. Los métodos *start* y *main* son públicos, esto se indica con el símbolo +. Mientras que el método calcular es privado que se indica con el símbolo -.

Diagrama de objetos

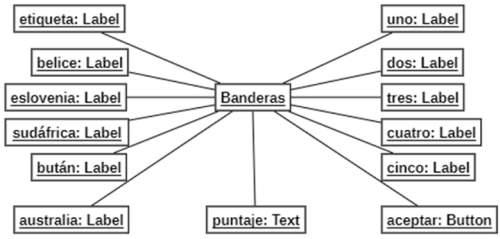
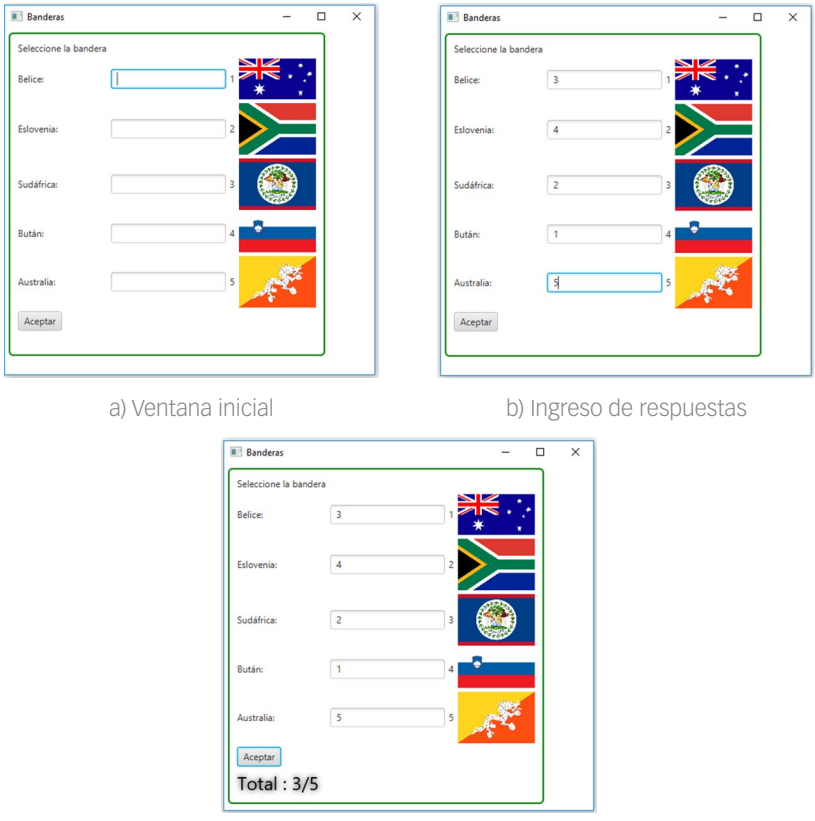


Figura 9.5. Diagrama de objetos del ejercicio 9.2.

Ejecución del programa



a) Ventana inicial

b) Ingreso de respuestas

c) Presentación de resultados

Figura 9.6. Ejecución del programa del ejercicio 9.2.

Ejercicios propuestos

- Modificar el programa anterior para que se muestre al usuario cuáles fueron las respuestas correctas e incorrectas.
- Modificar el programa anterior para que se amplíe el juego a diez países con sus banderas.
- Modificar el programa anterior para que de un conjunto de veinte países con sus banderas, el programa seleccione y muestre en pantalla en forma aleatoria solo cinco países.

Ejercicio 9.3. Figuras 2D

Una figura en dos dimensiones es cualquier forma que se pueda dibujar en un plano bidimensional. JavaFX ofrece una variedad de nodos para dibujar diferentes figuras bidimensionales. Todas las clases de figuras 2D forman parte del paquete *javafx.scene.shape*. En la figura 9.7 se observan las diferentes figuras 2D que soporta JavaFX, las cuales son subclases de *Shape*.

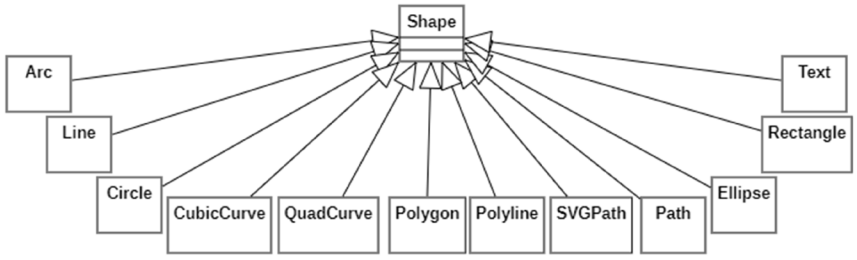


Figura 9.7. Jerarquía de clases de figuras 2D.

Un objeto *Shape* tiene un tamaño y una posición, que se definen por sus propiedades, las cuales varían de acuerdo con el tipo de forma. Los objetos *Shape* tienen un interior y una línea de contorno ("*stroke*"). La propiedad *fill* especifica el color para rellenar el interior de la forma. El valor por defecto del relleno es de color negro (BLACK). La propiedad *stroke* especifica el color de la línea de contorno, que es *null* de forma predeterminada, excepto para *Line*, *Polyline* y *Path*, que tienen Color.BLACK como predeterminado (Sharan, 2018).

A continuación, se presenta un resumen de las principales propiedades y métodos de las subclases de *Shape* (API Java, 2020).

Tabla 9.4. Algunos métodos de la clase *Shape* y sus subclases

Clase	Método	Descripción
Shape	*void setStroke(Paint valor)*	Establece la línea de contorno.
	void setStrokeWidth(double valor)	Establece la anchura de la línea de contorno.
	setFill(Paint valor)	Establece el relleno de la forma.
Arc	*Arc(double centroX, double centroY, double radioX, double radioY, double ánguloInicio, double longitud)*	Crea un arco con las propiedades pasadas como parámetros.
	void setFill(Paint valor)	Establece el relleno del arco.
	setType(ArcType valor)	Establece el tipo de arco.
Line	*Line(double inicioX, double inicioY, double finX, double finY)*	Crea una línea con las propiedades pasadas como parámetros.
Circle	*Circle(double radio)*	Crea un círculo con un radio determinado.
CubicCurve	*CubicCurve(double inicioX, double inicioY, double controlX1, double controlY1, double controlX2, double controlY2, double finX, double finY)*	Crea una curva paramétrica de Bezier de grado 3.
QuadCurve	*QuadCurve(double inicioX, double inicioY, double controlX1, double controlY1, double controlX2, double controlY2, double finX, double finY)*	Crea una curva cuadrática de grado 2.
Polygon	*Polygon()*	Crea un objeto polígono.
	getPoints().addAll(double … puntos)	Establece las coordenadas de un polígono.
PolyLine	*Polyline(double … puntos)*	Crea un objeto similar al polígono, pero no es una forma cerrada.
SVGPath	*SVGPath()*	Crea una forma simple que se construye analizando datos de una ruta SVG desde un *String*.
	void setContent(String valor)	Establece el contenido como un conjunto de coordenadas SVG.
Path	*Path()*	Crea una forma simple y proporciona las facilidades para la gestión de una ruta geométrica.
Ellipse	*Ellipse(double centroX, double centroY, double radioX, double radioY)*	Crea una elipse.

Clase	Método	Descripción
Rectangle	*Rectangle(double anchura, double altura)*	Crea un rectángulo con un tamaño dado.
	void setArcWidth(double valor)	Establece el diámetro vertical de los arcos del rectángulo.
	void setArcHeight(double valor)	Establece el diámetro horizontal del arco del rectángulo.
Text	*Text(double x, double y, String texto)*	Crea un texto en cierta posición.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Dibujar y presentar en ventanas gráficas diferentes figuras geométricas en 2D.
- Aplicar diferentes métodos para configurar la presentación visual de las figuras geométricas en 2D.

Enunciado: Figuras 2D

Crear una ventana gráfica con las siguientes figuras geométricas 2D (Sharan, 2015):

- Una línea amarilla con ancho de línea 4.0 px de (0,0) a (70, 70).
- Un rectángulo verde de anchura 100 px y altura 75 px.
- Un rectángulo de color agua de anchura 100 px y altura 75 px con bordes redondeados.
- Un paralelogramo rojo con borde negro con vértices en (30, 0), (130, 0), (100, 75) y (0, 75).
- Un hexágono gris con borde negro y con vértices en (100, 0), (120, 25), (120, 50), (100, 75), (80, 50), (80, 25) y (100, 0).
- Un círculo azul con borde negro de ancho 3.0 px.
- Una elipse amarilla con borde azul de ancho 3.0 px.
- Un arco cerrado sin relleno y con borde naranja.
- Un texto con el valor "JavaFX".

Instrucciones Java del ejercicio

Tabla 9.5. Instrucciones Java del ejercicio 9.3.

Instrucción	Método	Descripción
Stage	*Stage(Parent root, double anchura, double altura)*	Crea un objeto *Scene* para un nodo específico con un tamaño específico.
	void setScene(Scene valor)	Especifica la escena utilizada en este escenario.
	void setTitle(String valor)	Establece el título del escenario.
	void sizeToScene()	Establece el tamaño del escenario.
	void show()	Muestra la ventana.
HBox	*HBox()*	Constructor de HBox.
	void setSpacing(double valor)	Establece un valor de espaciado.
	void setStyle(String valor)	Establece un *String* con CSS asociado.

Solución

Clase: Figuras2D

```
package figuras2d;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Arc;
import javafx.scene.shape.ArcType;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Ellipse;
import javafx.scene.shape.Line;
import javafx.scene.shape.Polygon;
import javafx.scene.shape.Polyline;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Text;
import javafx.stage.Stage;

/**
 * Esta clase denominada Figuras permite crear varias figuras en 2D con
 * diferentes formatos de presentación.
 * @version 1.0/2020
 */
```

```
public class Figuras2D extends Application {
    Line línea; // Atributo que define una línea
    Rectangle rectángulo1; // Atributo que define un rectángulo
    Rectangle rectángulo2; /* Atributo que define un rectángulo con
        bordes redondeados */
    Polygon paralelogramo; // Atributo que define un polígono
    Polyline hexágono; // Atributo que define una polilínea
    Circle círculo; // Atributo que define un círculo
    Ellipse elipse; // Atributo que define una elipse
    Arc arco; // Atributo que define un arco
    Text texto; // Atributo que define un texto

    /**
    * Método main que lanza la aplicación
    * @param args Parámetro que define los argumentos de la aplicación
    */
    public static void main(String[] args) {
        Application.launch(args);
    }

    /**
    * Método start que inicia la aplicación
    * @param stage El escenario donde se ejecutará la aplicación
    */
    @Override
    public void start(Stage stage) {
        // Crea una línea amarilla de ancho 4.0 px
        línea = new Line(0, 0, 70, 70);
        línea.setStrokeWidth(4.0);
        línea.setStroke(Color.YELLOW);

        // Crea un rectángulo verde
        rectángulo1 = new Rectangle(100, 75);
        rectángulo1.setFill(Color.GREEN);

        // Crea un rectángulo agua con bordes redondeados
        rectángulo2 = new Rectangle(100, 75);
        rectángulo2.setFill(Color.AQUA);
        rectángulo2.setArcWidth(30);
        rectángulo2.setArcHeight(20);

        // Crea un paralelogramo rojo con borde negro
        paralelogramo = new Polygon();
        paralelogramo.getPoints().addAll(30.0, 0.0,
            130.0, 0.0,
```

```
        100.00, 75.0,
        0.0, 75.0);
paralelogramo.setFill(Color.RED);
paralelogramo.setStroke(Color.BLACK);

// Crea un hexágono gris con borde negro
hexágono = new Polyline(100.0, 0.0,
        120.0, 25.0,
        120.0, 50.0,
        100.0, 75.0,
        80.0, 50.0,
        80.0, 25.0,
        100.0, 0.0);
hexágono.setFill(Color.GREY);
hexágono.setStroke(Color.BLACK);

// Crea un círculo azul con borde negro de ancho 3.0 px
círculo = new Circle(40);
círculo.setFill(Color.BLUE);
círculo.setStroke(Color.BLACK);
círculo.setStrokeWidth(3.0);

// Crea una elipse amarilla con borde azul de ancho 3.0 px
elipse = new Ellipse(50, 50, 50, 25);
elipse.setFill(Color.YELLOW);
elipse.setStroke(Color.BLUE);
elipse.setStrokeWidth(3.0);

// Crea un arco cerrado sin relleno y con borde naranja
arco = new Arc(0, 0, 50, 80, 0, 120);
arco.setFill(Color.TRANSPARENT);
arco.setStroke(Color.ORANGE);
arco.setType(ArcType.CHORD);

// Crea un texto con el valor "JavaFX"
texto = new Text(10,10,"JavaFx");

// Agrega las figuras a un HBox
HBox root = new HBox(línea, rectángulo1, rectángulo2,
    paralelogramo, hexágono, círculo, elipse, arco, texto);
root.setSpacing(10);
root.setStyle("-fx-padding: 10;"
    + "-fx-border-style: solid inside;"
    + "-fx-border-width: 2;"
```

```
                + "-fx-border-insets: 5;"
                + "-fx-border-radius: 5;"
                + "-fx-border-color: brown;");
        Scene scene = new Scene(root); // Crea un escenario con el root
        stage.setScene(scene); // Establece la escena para el escenario
        stage.setTitle("Figuras 2D"); // Establece título del escenario
        stage.show(); // Muestra el escenario (ventana)
    }
}
```

Diagrama de clases

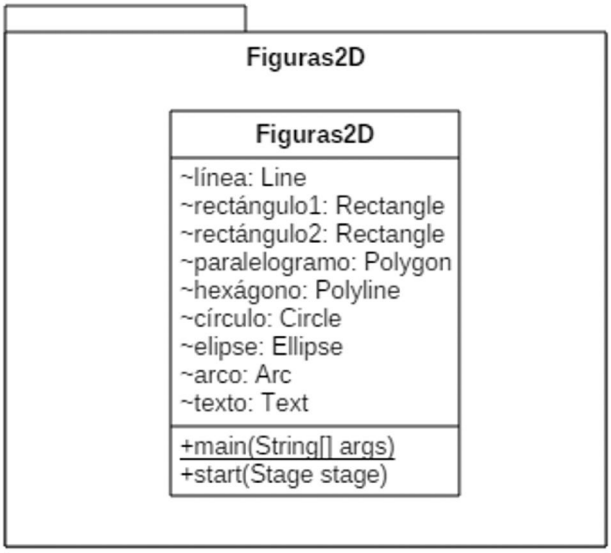


Figura 9.8. Diagrama de clases del ejercicio 9.3.

Explicación del diagrama de clases

El diagrama de clases muestra un paquete denominado "Figuras2D", el cual contiene una sola clase denominada también "Figuras2D". Esta clase es una ventana gráfica que tiene atributos para indicar sus diferentes formas gráficas: una línea (*Line*), dos rectángulos (*Rectangle*), un paralelogramo (*Polygon*), un hexágono (*Polyline*), un círculo (*Circle*), una elipse (*Ellipse*), un arco (*Arc*) y un texto (*Text*). Los atributos tienen visibilidad de paquete indicada con el símbolo ~. La clase cuenta con dos métodos: *start*

que inicia la aplicación gráfica y el método *main* (punto de entrada a la aplicación). El método *main* es un método estático, por lo cual se representa con su texto subrayado. Los dos métodos son públicos, lo cual se indica con el símbolo +.

Diagrama de objetos

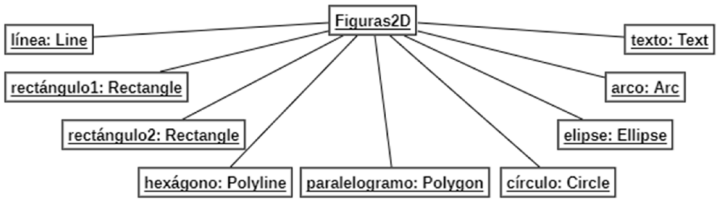


Figura 9.9. Diagrama de objetos del ejercicio 9.3.

Ejecución del programa

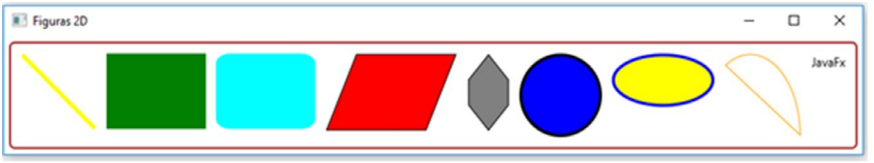


Figura 9.10. Ejecución del programa del ejercicio 9.3.

Ejercicios propuestos

- Agregar al programa del ejercicio anterior las siguientes figuras 2D:
  - Una curva cuadrática.
  - Una curva cúbica.
- Realizar una ventana gráfica con las siguientes figuras 2D y colocarlas en el formato presentado:

Figura 9.11. Figuras 2D

Ejercicio 9.4. Figuras 3D

Cualquier forma dibujada que tenga tres dimensiones (largo, ancho y profundidad) se conoce como forma 3D. Algunos ejemplos de figuras 3D son los cubos, las esferas y las pirámides.

La figura 9.12 muestra un diagrama de clases que representa las formas 3D en JavaFX. Las clases de formas 3D están en el paquete *javafx.scene.shape*. Las clases *Box*, *Sphere* y *Cylinder* representan tres formas 3D predefinidas que posee JavaFX y son subclases de *Shape3D*.

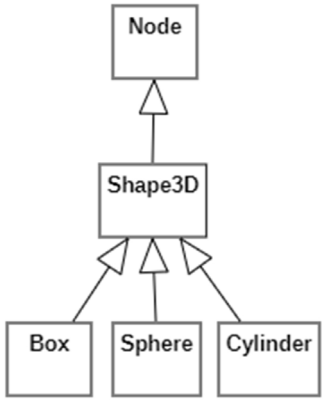


Figura 9.12. Jerarquía de clases de figuras 3D

En la tabla 9.6, se presentan los principales métodos de estas clases para crear figuras 3D (API Java, 2020).

Tabla 9.6. Algunos métodos de la clase *Shape* 3D y sus subclases

Clase	Método	Descripción
Shape3D	*void setTranslateX(double valor)*	Define la coordenada *x* de la forma.
	void setTranslateY(double valor)	Define la coordenada *y* de la forma.
	void setTranslateZ(double valor)	Define la coordenada *z* de la forma.
Box	*Box(double largo, double ancho, double profundidad)*	Crea un cubo especificando su tamaño.
Sphere	*Sphere(double radio, int divisiones)*	Crea una esfera y su resolución.
Cylinder	*Cylinder(double radio, double altura, int divisiones)*	Crea una esfera especificando su tamaño y resolución.

Un objeto 3D tiene tres dimensiones: *x*, *y*, *z*. La figura 9.13 muestra el sistema de coordenadas 3D utilizado en JavaFX. La dirección positiva del eje *x* apunta a la derecha desde el origen; la dirección positiva del eje *y* apunta hacia abajo; y la dirección positiva del eje *z* apunta a la pantalla. Las direcciones negativas en los ejes (no se muestran), se extienden en direcciones opuestas al origen.

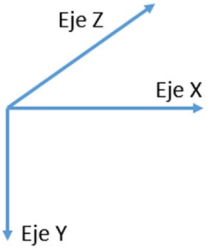


Figura 9.13. Sistema de coordenadas 3D

El material de textura de una figura 3D se especifica con la clase *PhongMaterial*. Esta clase tiene métodos como *setDiffuseColor* que especifica el color del material. Con el método *setSpecularColor* se define el color especular del material (brillo que produce una fuente de luz aplicada a la figura). Luego, se invoca al método *setMaterial* de la figura pasándole como parámetro el objeto *PhongMaterial*.

Con el método *setDrawMode* se establece como los polígonos que conforman la textura de la figura 3D serán dibujados.

La visualización en 3D en JavaFX se logra utilizando luces (clase *PointLight*) y cámaras (clase *PerspectiveCamera*). Las luces y las cámaras también son nodos que se agregan a la escena, la escena se enciende con luces y se ve con una cámara. Las posiciones de luces y cámaras en el espacio determinan las áreas iluminadas y visibles de la escena (Sharan, 2015).

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Entender las clases que representan figuras 3D en JavaFX.
- Dibujar figuras 3D predefinidas.
- Utilizar fuentes de luz aplicadas a figuras 3D en JavaFX.

Enunciado: Figuras 3D

Desarrollar una ventana gráfica con las siguientes figuras 3D:

- Un *Box* de tamaño 100 en sus 3 dimensiones de color amarillo en la posición: x=150, y=0; z=400.
- Una esfera de radio 60 de color naranja con 40 divisiones en su textura, en la posición: x =300, y=-5, z=400.
- Un cilindro de radio 75 y altura 100 de color difuso y especular naranja en la posición: x=500, y=5, z=500.

El punto de luz de las figuras está ubicado en la posición: x=350, y=100, z=300 y la cámara está ubicada en x=100, y=-50, z=300. Adaptado de Sharan, (2015).

Instrucciones Java utilizadas en el ejercicio

Tabla 9.7. Instrucciones Java del ejercicio 9.4.

Clase	Método	Descripción
Shape3D	*void setMaterial(material1)*	Establece el material de la superficie de la figura 3D.
	void setDrawMode(DrawModel valor)	Establece cómo los polígonos de la textura de la figura 3D serán dibujados.
Stage	*Stage(Parent root, double anchura, double altura)*	Crea un objeto *Scene* para un nodo específico con un tamaño específico.
	void setScene(Scene valor)	Especifica la escena utilizada en este escenario.
	void setTitle(String valor)	Establece el título del escenario.
	void sizeToScene()	Establece el tamaño del escenario.
	void show()	Muestra la ventana.
Scene	*void setCamera(Camera valor)*	Establece una cámara para una escena.
PhongMaterial	*PhongMaterial()*	Crea un objeto *PhongMaterial*.
	void setDiffuseColor(Color valor)	Establece el color del material.
	void setSpecularColor(Color valor)	Establece el color especular (brillo que produce la fuente de luz).
PointLight	*PointLight()*	Crea un objeto fuente de luz.
PerspectiveCamera	*PerspectiveCamera()*	Especifica una cámara para renderizar una escena.

Clase	Método	Descripción
Group	*Group()*	Constructor de un contenedor con una lista de elementos que se presentan en orden.

Solución

Clase: Figuras3D

```
package figuras3d;

import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.PerspectiveCamera;
import javafx.scene.PointLight;
import javafx.scene.paint.Color;
import javafx.scene.Scene;
import javafx.scene.shape.Box;
import javafx.scene.paint.PhongMaterial;
import javafx.scene.shape.Cylinder;
import javafx.scene.shape.DrawMode;
import javafx.scene.shape.Sphere;
import javafx.stage.Stage;

/**
 * Esta clase denominada Figuras3D permite crear varias figuras en tres
 * dimensiones y se muestran en diferentes formatos de presentación
 * @version 1.0/2020
 */
public class Figuras3D extends Application {
    Box caja; // Atributo que define un cubo
    PhongMaterial material1; // Atributo que el material del cubo
    Sphere esfera; // Atributo que define una esfera
    PhongMaterial material2; // Atributo que define el material de la esfera
    Cylinder cilindro; // Atributo que define un cilindro
    PointLight luz; // Atributo que define un punto de luz

    /**
    * Método main que lanza la aplicación
    * @param args Parámetro que define los argumentos de la aplicación
    */
```

```
    public static void main(String[] args) {
        Application.launch(args);
    }

    /**
    * Método start que inicia la aplicación
    * @param stage El escenario donde se ejecutará la aplicación
    */
    @Override
    public void start(Stage stage) {
        // Crea un Box de tamaño 100
        caja = new Box(100, 100, 100);
        // Se establece la posición del box
        caja.setTranslateX(150);
        caja.setTranslateY(0);
        caja.setTranslateZ(400);

        // Establece material del box
        material1 = new PhongMaterial();
        material1.setDiffuseColor(Color.YELLOW);
        caja.setMaterial(material1);

        // Crea una Sphere con radio de 60 y divisiones = 40
        esfera = new Sphere(60, 40);
        esfera.setDrawMode(DrawMode.LINE);
        // Se establece la posición de la esfera
        esfera.setTranslateX(300);
        esfera.setTranslateY(-5);
        esfera.setTranslateZ(400);

        // Establece material del cilindro
        material2 = new PhongMaterial();
        material2.setDiffuseColor(Color.ORANGE);
        material2.setSpecularColor(Color.ORANGE);

        // Crea un Cylinder de radio 75 y altura 100
        cilindro = new Cylinder(75, 100);
        // Se establece la posición del cilindro
        cilindro.setTranslateX(500);
        cilindro.setTranslateY(5);
        cilindro.setTranslateZ(500);
        cilindro.setMaterial(material2);

        // Crea un punto de luz y lo ubica en x=350, y= 100, z = 300
        luz = new PointLight();
```

```
            luz.setTranslateX(350);
            luz.setTranslateY(100);
            luz.setTranslateZ(300);
            Group root = new Group(caja, esfera, cilindro, luz); /* Agrega los
                elementos al grupo */

            // Crea una escena con buffer de profundidad habilitado
            Scene scene = new Scene(root, 400, 100, true);
            // Establece una cámara para ver figuras 3D
            PerspectiveCamera camera = new PerspectiveCamera(false);
            // Establece la posición de la cámara
            camera.setTranslateX(100);
            camera.setTranslateY(-50);
            camera.setTranslateZ(300);
            scene.setCamera(camera); // Agrega la cámara a la escena
            stage.setScene(scene); // Agrega la escena al escenario
            stage.setTitle("Figuras 3D");  // Establece título del escenario
            stage.show(); // Muestra el escenario (ventana)
        }
    }
```

Diagrama de clases

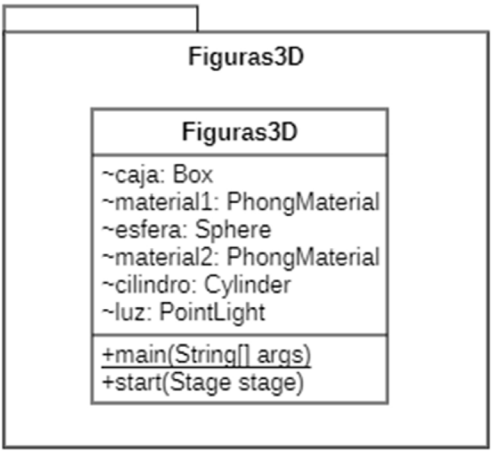


Figura 9.14. Diagrama de clases del ejercicio 9.4.

Explicación del diagrama de clases

El diagrama de clases muestra un paquete denominado "Figuras3D", que contiene una sola clase denominada también "Figuras3D". Esta clase es una ventana gráfica que tiene atributos para indicar diferentes figuras tridimensionales que se presentarán en la ventana: un cubo (*Box*), una esfera (*Sphere*) y un cilindro (*Cylinder*). Los atributos tienen visibilidad de paquete indicada con el símbolo ~. La clase cuenta con dos métodos: *start* que inicia la aplicación gráfica y el método *main* (punto de entrada a la aplicación). El método *main* es un método estático, por lo cual se representa con su texto subrayado. Los dos métodos son públicos, lo cual se indica con el símbolo +.

Diagrama de objetos

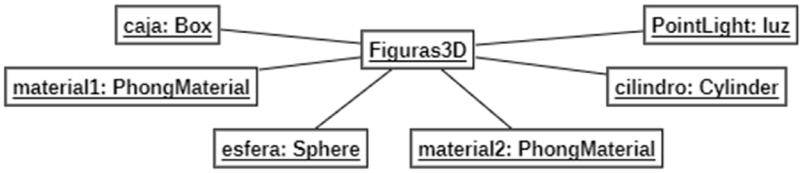


Figura 9.15. Diagrama de objetos del ejercicio 9.4.

Ejecución del programa

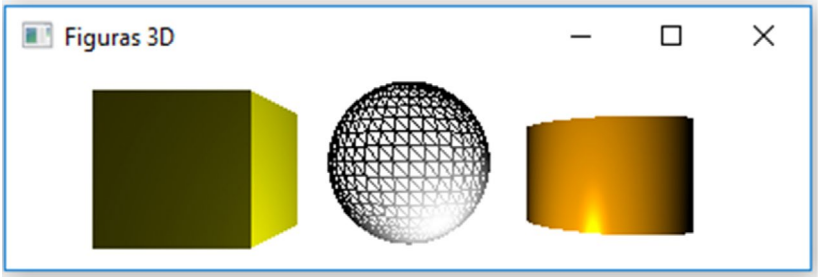


Figura 9.16. Ejecución del programa del ejercicio 9.4.

Ejercicios propuestos

- Desarrollar la siguiente escena 3D:

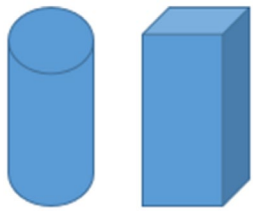


Figura 9.17. Escena 3D.

Fuente: API Java (2020).

Ejercicio 9.5. Transformaciones

Una transformación es un mapeo de puntos en un espacio de coordenadas para preservar distancias y direcciones entre ellos. Se pueden aplicar varios tipos de transformaciones a puntos en un espacio de coordenadas (Sharan, 2018).

En la figura 9.18 se muestran los diferentes tipos de transformación que permite JavaFX.

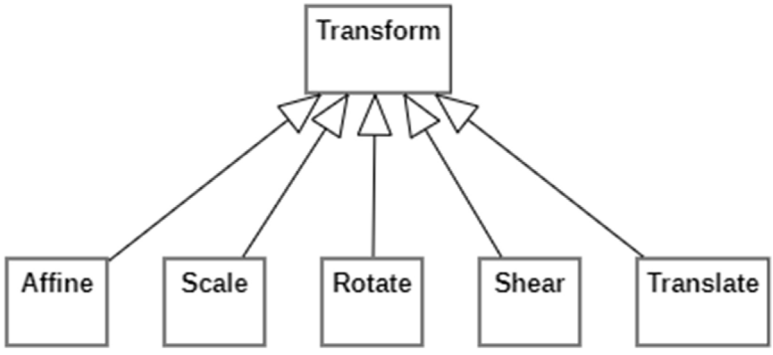


Figura 9.18. Jerarquía de clases de elementos de transformación

En la tabla 9.8 se describen brevemente las principales clases utilizadas para desarrollar transformaciones en formas de JavaFX (API Java, 2020).

Tabla 9.8. Algunas clases para realizar transformaciones

Clase	Descripción
Affine	Esta transformación realiza un mapeo lineal desde coordenadas 2D/3D a otras coordenadas 2D/3D mientras preserva la rectitud y paralelismo de las líneas.
Scale	Representa una transformación que escala las coordenadas por los factores especificados.
Rotate	Esta transformación rota las coordenadas alrededor de un punto de anclaje.
Shear	Esta transformación corta coordenadas por los multiplicadores especificados.
Translate	Esta transformación traslada coordenadas por los factores especificados.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Comprender qué es una transformación de figuras en JavaFX.
- Entender y aplicar las transformaciones de traslación, rotación, escala y corte.
- Aplicar diferentes transformaciones a componentes gráficos.

Enunciado: Transformaciones

Construir una ventana gráfica con cinco rectángulos:

- El primer rectángulo es de color rojo oscuro y tiene un tamaño de 100x50.
- El segundo rectángulo es de color índigo, tamaño de 100x50 y debe estar trasladado a la coordenada (100, 70).
- El tercer rectángulo es de color amarillo, tamaño de 100x50, debe estar ubicado en la coordenada (170, 60) y rotar 45 grados.
- El cuarto rectángulo es de color azul, tamaño de 100x50, debe estar localizado en la coordenada (250, 260) y se debe aumentar su escala un 30%.
- El quinto rectángulo es de color rojo, tamaño de 100x50, debe estar ubicado en la coordenada (360, 340) y se le debe aplicar un corte de 0.5 en ambos ejes.

Instrucciones Java del ejercicio

Tabla 9.9. Instrucciones Java del ejercicio 9.5.

Clase	Método	Descripción
Rectangle	*Rectangle(double anchura, double altura, Color value)*	Crea un rectángulo con un tamaño y color especificados.
	setTranslateX(double valor)	Traslada la figura a la coordenada *x* especificada.
	setTranslateY(double valor)	Traslada la figura a la coordenada *y* especificada.
	setRotate(double valor)	Rota la figura la cantidad de grados especificada.
	setScaleX(double valor)	Cambia la escala *x* de la figura el valor dado.
	setScaleY(double valor)	Cambia la escala *y* de la figura el valor dado.
	getTransforms().addAll(Shear shear)	Aplicar el recorte especificado a la figura.
Shear	*Shear(double valorX, double valorY)*	Crea un objeto *Shear*.
Group	*Group()*	Constructor de un contenedor con una lista de elementos que se representan en orden.
Stage	*Stage(Parent root, double anchura, double altura)*	Crea un objeto *Scene* para un nodo específico con un tamaño específico.
	void setScene(Scene valor)	Especifica la escena utilizada en este escenario.
	void setTitle(String valor)	Establece el título del escenario.
	void show()	Muestra la ventana.
Scene	*Scene(Parent root, double valorX, double valorY)*	Construye una escena para un nodo específico con el tamaño especificado.

Solución

Clase: Transformaciones

```
package transformaciones;

import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
```

```
import javafx.scene.transform.Shear;
import javafx.stage.Stage;

/**
 * Esta clase denominada Transformaciones crea varias figuras 2D a las
 * cuales se les realizará varias transformaciones
 * @version 1.0/2020
 */
public class Transformaciones extends Application {
    Rectangle rectángulo1; // Atributo que define un rectángulo fijo
    Rectangle rectángulo2; // Atributo que define un rectángulo a trasladar
    Rectangle rectángulo3; // Atributo que define un rectángulo a rotar
    Rectangle rectángulo4; // Atributo que define un rectángulo a escalar
    Rectangle rectángulo5; // Atributo que define un rectángulo a cortar

    /**
    * Método main que lanza la aplicación
    * @param args Parámetro que define los argumentos de la aplicación
    */
    public static void main(String[] args) {
        Application.launch(args);
    }

    /**
    * Método start que inicia la aplicación
    * @param stage El escenario donde se ejecutará la aplicación
    */
    @Override
    public void start(Stage stage) {
        rectángulo1 = new Rectangle(100, 50, Color.DARKRED);
        rectángulo2 = new Rectangle(100, 50, Color.INDIGO);
        rectángulo3 = new Rectangle(100, 50, Color.YELLOW);
        rectángulo4 = new Rectangle(100, 50, Color.BLUE);
        rectángulo5 = new Rectangle(100, 50, Color.RED);

        // Aplica transformaciones al rectángulo2
        rectángulo2.setTranslateX(100); // Traslación en el eje x
        rectángulo2.setTranslateY(70); // Traslación en el eje y

        // Aplica transformaciones al rectángulo3
        rectángulo3.setTranslateX(170); // Traslación en el eje x
        rectángulo3.setTranslateY(160); // Traslación en el eje y
        rectángulo3.setRotate(45); // Rotación de 45 grados
```

```
            // Aplica transformaciones al rectángulo4
            rectángulo4.setTranslateX(250); // Traslación en el eje x
            rectángulo4.setTranslateY(260); // Traslación en el eje y
            rectángulo4.setScaleX(1.3); // Ampliación de escala en el eje x
            rectángulo4.setScaleY(1.3); // Ampliación de escala en el eje y

            // Aplica transformaciones al rectángulo5
            rectángulo5.setTranslateX(360); // Traslación en el eje x
            rectángulo5.setTranslateY(340); // Traslación en el eje y
            Shear shear = new Shear(0.5, 0.5);
            rectángulo5.getTransforms().addAll(shear); /* Establece recorte
                de la figura */

            // Agrega elementos gráficos al grupo
            Group root = new Group(rectángulo1, rectángulo2, rectángulo3,
                rectángulo4, rectángulo5);
            Scene scene = new Scene(root, 500, 450); /* Agrega el grupo a la
                escena */
            stage.setScene(scene); // Establece escena para el escenario
            stage.setTitle("Transformaciones"); /* Establece título del
                escenario */
            stage.show(); // Muestra el escenario (ventana)
        }
    }
```

Diagrama de clases

Figura 9.19. Diagrama de clases del ejercicio 9.5.

Explicación del diagrama de clases

El diagrama de clases muestra un paquete denominado "Transformaciones", este contiene una sola clase denominada también "Transformaciones". Esta clase es una ventana gráfica que tiene atributos para indicar diferentes rectángulos (clase *Rectangle*), a los cuales se les aplicarán varias transformaciones. Los atributos tienen visibilidad de paquete indicada con el símbolo ~. La clase cuenta con dos métodos: *start* que inicia la aplicación gráfica y el método *main* (punto de entrada a la aplicación). El método *main* es un método estático, por lo cual se representa con su texto subrayado. Los dos métodos son públicos, lo cual se indica con el símbolo +.

Diagrama de objetos

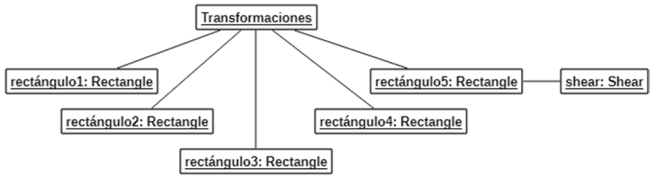


Figura 9.20. Diagrama de objetos del ejercicio 9.5.

Ejecución del programa

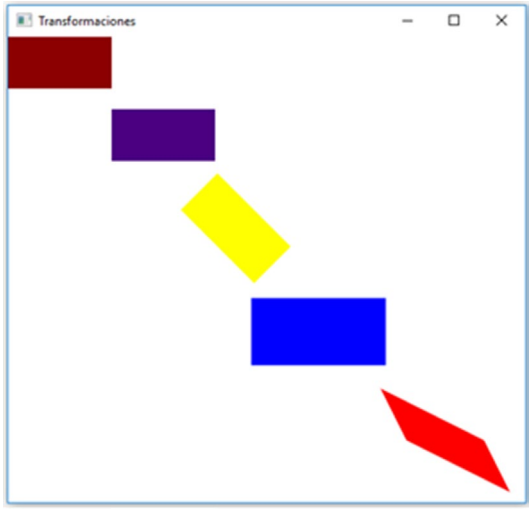


Figura 9.21. Ejecución del programa del ejercicio 9.5.

Ejercicios propuestos

- Dibujar el teclado numérico y de operaciones aritméticas de una calculadora.
- Dibujar las siguientes figuras realizando transformaciones de JavaFX.

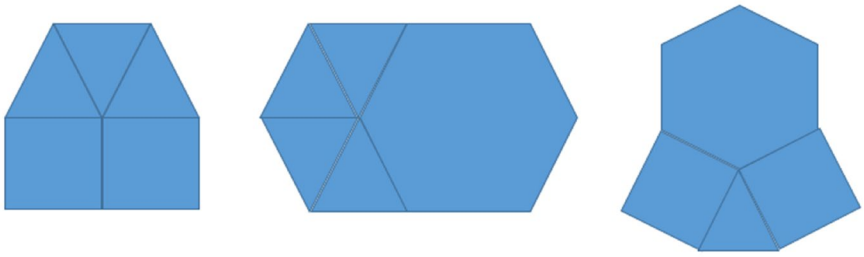


Figura 9.22. Diferentes figuras 2D

Ejercicio 9.6. Animaciones

La animación implica algún tipo de movimiento, que se genera al mostrar imágenes en rápida sucesión. En JavaFX, la animación se define como el cambio de la propiedad de un nodo a lo largo del tiempo. Si la propiedad que cambia determina la ubicación del nodo, la animación en JavaFX producirá una ilusión de movimiento (Vos, Chin, Gao, Weaver y Iverson, 2012).

Las clases que proporcionan animación en JavaFX están en el paquete *javafx.animation*, excepto la clase *Duration*, que está en el paquete *javafx.util*. La figura 9.23 muestra un diagrama de clase con la mayoría de las clases relacionadas con la animación.

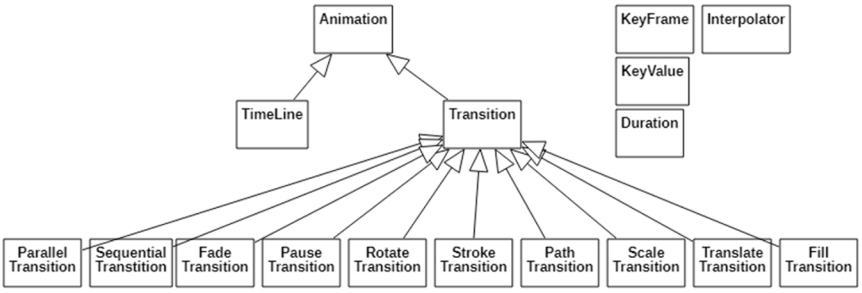


Figura 9.23. Jerarquía de clases de animaciones

En la tabla 9.10 se describen brevemente las principales clases utilizadas en JavaFX para realizar animaciones de figuras (API Java, 2020).

Tabla 9.10. Algunas clases para realizar animaciones

Clase	Descripción
Animation	Proporciona la funcionalidad básica a todas las animaciones en JavaFx.
Timeline	Una animación es caracterizada por sus propiedades asociadas, como tamaño, ubicación, color, etc. *Timeline* proporciona la capacidad de actualizar los valores de las propiedades a lo largo del tiempo.
Transition	Proporciona los medios para incorporar animaciones en una línea de tiempo interna.
ParallelTransition	Una transición paralela ejecuta varias transiciones simultáneamente.
SequentialTransition	Una transición secuencial ejecuta varias transiciones una después de otra.
FadeTransition	Esta transición crea una animación de efecto de opacidad que abarca su duración.
PauseTransition	Esta transición es utilizada para pausar entre múltiples animaciones aplicadas a un nodo de manera secuencial.
RotateTranstion	Crea una animación de rotación que abarca su duración.
StrokeTransition	Realiza una animación del color del borde del nodo para que pueda fluctuar entre dos valores de color durante la duración especificada.
PathTransition	Mueve un nodo a lo largo de cierto recorrido especificado de un extremo al otro durante un tiempo determinado.
ScaleTransition	Esta transición realiza una animación de la escala del nodo durante la duración especificada por un factor determinado en cualquiera o en todas las tres direcciones *x*, *y*, *z*.
TranslateTransition	Traduce el nodo de una posición a otra durante la duración especificada.
FillTransition	Esta transición crea una animación que cambia el relleno de una forma a lo largo de una duración.

Clase	Descripción
Duration	Define una duración de tiempo. La duración se puede crear utilizando el constructor o uno de los métodos de construcción estáticos, como segundos (*double*) o minutos (*double*).
KeyValue	Define un valor clave para ser interpolado en un intervalo particular a lo largo de la animación.
KeyFrame	Define valores objetivo en un punto específico en el tiempo para un conjunto de variables que se interpolan a lo largo de un *Timeline*.

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Comprender qué es y cómo se realizan animaciones en JavaFX.
- Conocer las clases en JavaFX que se utilizan para realizar animaciones.
- Aplicar la clase *RotationTransition* para realizar animaciones de rotación de figuras.

Enunciado: Animaciones

Se requiere una ventana gráfica que muestre las siguientes figuras con sus animaciones respectivas:

- Un cuadrado de tamaño 100x100 que rote 2 segundos en un sentido y luego rote en forma contraria.
- Una elipse amarilla ubicada en (50, 50) y con radioX = 50 y radioY = 25, con línea de borde azul de 3.0 px y que tenga un cambio de opacidad de 2 segundos.

Instrucciones Java del ejercicio

Tabla 9.11. Instrucciones Java del ejercicio 9.6.

Clase	Método	Descripción
Rectangle	*Rectangle(double anchura, double altura, Paint relleno)*	Crea un rectángulo con un tamaño dado y un color de relleno.
Ellipse	*Ellipse(double centroX, double centroY, double radioX, double radioY)*	Crea una elipse.
	void setStroke(Paint valor)	Establece la línea de contorno.
	void setStrokeWidth(double valor)	Establece la anchura de la línea de contorno.
	setFill(Paint valor)	Establece el relleno de la forma.
RotateTransition	*RotateTransition(Duration duración, Node nodo)*	Crea una transición de rotación con una duración determinada para un nodo específico.
	void setFromAngle(double valor)	Especifica el ángulo inicial de rotación de la figura.
	void setToAngle(double valor)	Especifica el ángulo de detención de rotación de la figura.
	void setCycleCount(int valor)	Define el número de ciclos de la rotación.
	void setAutoReverse(boolean valor)	Define si la rotación invierte su dirección en ciclos alternos.
	void play()	Reproduce la rotación desde la posición actual en la dirección indicada por la velocidad.
FadeTransition	*FadeTransition(Duration duración, Node nodo)*	Crea una transición de opacidad con una duración para un nodo específico.
	void setFromValue(double valor)	Especifica el valor de inicio de la opacidad.
	void setToValue(double valor)	Especifica el valor de detención de la opacidad.
	void setCycleCount(int valor)	Define el número de ciclos de opacidad.
	void setAutoReverse(boolean valor)	Define si la opacidad invierte su dirección en ciclos alternos.
	void play()	Reproduce la opacidad desde la posición actual en la dirección indicada por la velocidad.

Clase	Método	Descripción
HBox	*HBox()*	Construye un HBox que permite presentar elementos horizontalmente.
	void setMargin(Node nodo, Inset valor)	Establece el margen para un nodo.
Scene	*Scene(Parent root)*	Construye una escena para un nodo específico.
Stage	*void setScene(Scene valor)*	Especifica la escena utilizada en este escenario.
	void setTitle(String valor)	Establece el título del escenario.
	void sizeToScene()	Establece el tamaño del escenario.
	void show()	Muestra la ventana.

Solución

Clase: Animaciones

```
package animaciones;

import javafx.animation.FadeTransition;
import javafx.animation.RotateTransition;
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Ellipse;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
import javafx.util.Duration;

/**
 * Esta clase denominada Animaciones crea un cuadrado verde y una
 * elipse amararilla. El cuadrado rotará con respecto a su centro en dos
 * direcciones consecutivas y la elipse tendrá cambio en su opacidad una
 * cierta duración.
 * @version 1.0/2020
 */
```

```
public class Animaciones extends Application {
    Rectangle rectángulo; // Rectángulo animado presente en la ventana
    Ellipse elipse; // Elipse animada presente en la ventana

    /**
    * Método main que lanza la aplicación
    * @param args Parámetro que define los argumentos de la aplicación
    */
    public static void main(String[] args) {
        Application.launch(args);
    }

    /**
    * Método start que inicia la aplicación
    * @param stage El escenario donde se ejecutará la aplicación
    */
    @Override
    public void start(Stage stage) {
        rectángulo = new Rectangle(100, 100, Color.GREEN);
        // Inicializa una transición de rotación del rectángulo de duración
            2 segundos
        RotateTransition rt = new RotateTransition(Duration.seconds(2),
            rectángulo);
        rt.setFromAngle(0.0); // Establece ángulo de inicio de la rotación
        rt.setToAngle(360.0); /* Establece ángulo de detención de la
            rotación */
        rt.setCycleCount(RotateTransition.INDEFINITE); /* Número de
            ciclos de rotación */
        rt.setAutoReverse(true); /* Invierte la dirección de rotación
        rt.play(); */

        elipse = new Ellipse(50, 50, 50, 25); // Crea una elipse
        elipse.setFill(Color.YELLOW); // Color de relleno de la elipse
        elipse.setStroke(Color.BLUE); // Color de la línea de borde
        elipse.setStrokeWidth(3.0); // Anchura del borde
        /* Inicializa una transición de opacidad de la elipse por 2
            segundos */
        FadeTransition fadeInOut = new FadeTransition(Duration.
            seconds(2), elipse);
        fadeInOut.setFromValue(1.0); /* Especifica el valor de inicio de
            la opacidad */
        fadeInOut.setToValue(.20); /* Especifica el valor de detención de
            la opacidad */
```

```
            // Establece el número de ciclos de opacidad
            fadeInOut.setCycleCount(FadeTransition.INDEFINITE);
            fadeInOut.setAutoReverse(true); /* Invierte la dirección de la
                opacidad */
            fadeInOut.play();

            // Establece un contenedor HBox con sus márgenes
            HBox.setMargin(rectángulo, new Insets(80));
            HBox.setMargin(elipse, new Insets(80));
            HBox root = new HBox(rectángulo, elipse); /* Crea un HBox con
                el rectángulo y la elipse */
            Scene escena = new Scene(root); // Crea una escena con el root
            stage.setScene(escena); // Establece la escena para el escenario
            stage.setTitle("Rotación"); // Establece título de la escena
            stage.show(); // Muestra la escena (ventana)
        }
    }
```

Diagrama de clases

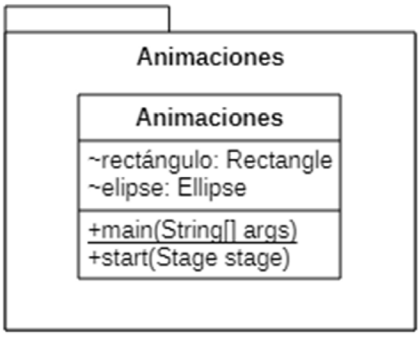


Figura 9.24. Diagrama de clases del ejercicio 9.6.

Explicación del diagrama de clases

El diagrama de clases muestra un paquete denominado "Animaciones", este contiene una sola clase denominada también "Animaciones". Esta clase es una ventana gráfica que tiene atributos para indicar dos figuras: un rectángulo (*Rectangle*) y una elipse (*Ellipse*), a estas se les aplicarán varios efectos de animación. Los atributos tienen visibilidad de paquete indicada con el símbolo ~. La clase cuenta con dos métodos: *start* que inicia la aplicación

gráfica y el método *main* (punto de entrada a la aplicación). El método *main* es un método estático, por lo cual se representa con su texto subrayado. Los dos métodos son públicos, lo cual se indica con el símbolo +.

Diagrama de objetos

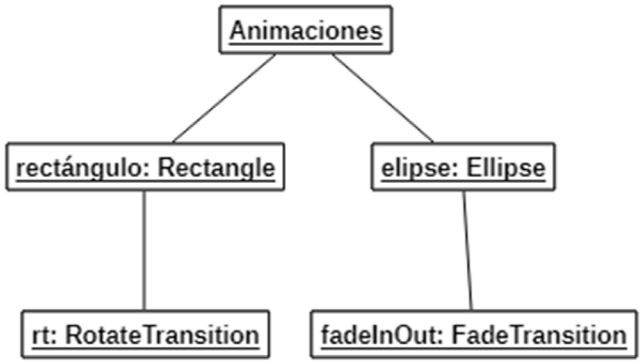


Figura 9.25. Diagrama de objetos del ejercicio 9.6.

Ejecución del programa

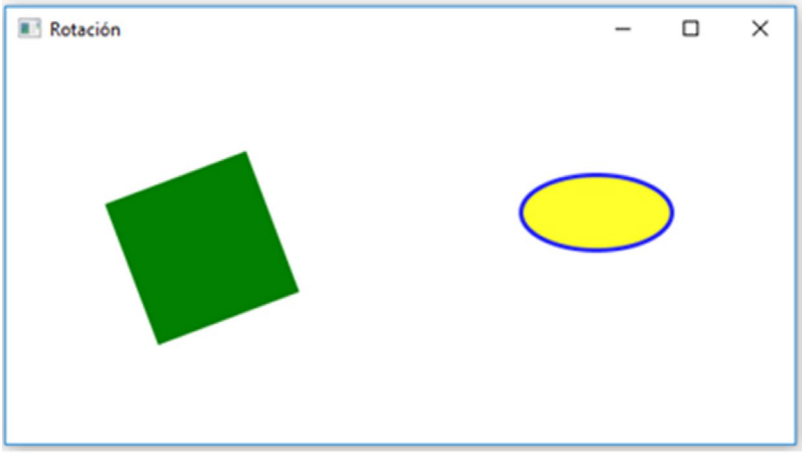


Figura 9.26. Ejecución del programa del ejercicio 9.6.

Ejercicios propuestos

- Realizar un programa que muestre en la parte superior de una ventana gráfica el texto "Programación Orientada a Objetos" y lo mueva de izquierda a derecha hasta llegar al borde derecho y luego lo traslade de derecha a izquierda. Utilice la clase *TimeLine*.
- Realizar un programa que muestre en una ventana gráfica un cierto recorrido (*Path*) y permita que un círculo rojo con bordes negros de radio 10 se mueva por ese recorrido en cinco segundos.

Ejercicio 9.7. Gráficas

Las gráficas son representaciones visuales de datos que proporcionan una manera más fácil de analizar un gran volumen de datos. Existen diferentes tipos de gráficos que difieren en la forma en que representan los datos. JavaFX incluye gráficos que pueden integrarse en una aplicación Java escribiendo pocas líneas de código. JavaFX contiene una API de gráficos completa y extensible que proporciona soporte integrado para varios tipos de gráficos (Sharan, 2015).

La API *Chart* de JavaFX consta de varias clases predefinidas en el paquete *javafx.scene.chart*. La figura 9.27 muestra un diagrama de clase que representa diferentes tipos de gráficas definidos en JavaFX.

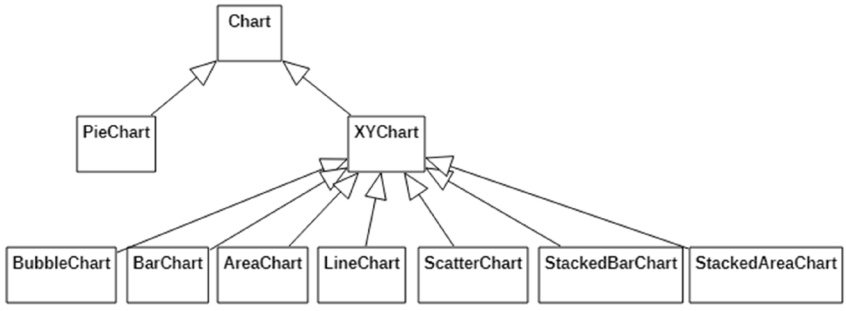
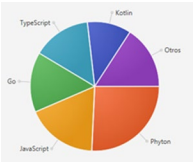
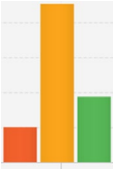
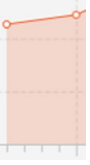
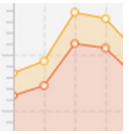


Figura 9.27. Jerarquía de clases de gráficas

En la tabla 9.12 se describen brevemente las principales clases utilizadas en JavaFX para hacer diferentes gráficas (API Java, 2020).

Tabla 9.12. Algunas clases para construir gráficas

Clase	Descripción	Gráfica
Chart	Clase base para todas las gráficas. Tiene tres partes: título, leyenda y contenido de la gráfica.	-
PieChart	Muestra un gráfico circular. El contenido del gráfico se rellena con sectores circulares basados en los datos establecidos.	
XYChart<X,Y>	Clase base para todas las gráficas de 2 ejes. Es responsable de dibujar los dos ejes y el contenido de la gráfica.	-
BubbleChart<X,Y>	Tipo de gráfica que traza burbujas para los puntos de datos en una serie.	
BarChart<X,Y>	Una gráfica que traza barras que indican valores de datos para una categoría. Las barras pueden ser verticales u horizontales.	
AreaChart<X,Y>	Traza una área entre la línea que conecta los puntos de datos y la línea 0 en el eje Y.	
LineChart<X,Y>	Traza una línea que conecta los puntos de datos en una serie.	
ScatterChart<X,Y>	Tipo de gráfico que traza símbolos para los puntos de datos en una serie.	

Clase	Descripción	Gráfica
StackedBarChat<X,Y>	Variación de *BarChart* que traza barras que indican valores de datos para una categoría.	
StackedAreaChat<X,Y>	Variación de *AreaChart* que muestra las tendencias de la contribución de cada valor, por ejemplo, el tiempo.	

Objetivos de aprendizaje

Al finalizar este ejercicio, el lector tendrá la capacidad para:

- Comprender la API *Chart* de JavaFX.
- Crear diferentes tipos de gráficas utilizando API *Chart* de JavaFX.

Enunciado: Gráfica de torta

Se desea desarrollar una gráfica de torta (*PieChart*) que muestre en formato gráfico los siguientes datos correspondientes a porcentajes de uso de lenguajes de programación en el 2020:

- Phyton: 25.7 %
- JavaScript: 17.8 %
- Go: 15 %
- TypeScript: 14.6 %
- Kotlin: 11 %
- Otros: 15.8 %

Instrucciones Java del ejercicio

Tabla 9.13. Instrucciones Java del ejercicio 9.7.

Clase	Método	Descripción
ObservableList<E>	*ObservableList<PieChart.Data>*	Crea una lista que permite a los oyentes rastrear los cambios cuando ocurren en el *PieChart*.
	boolean add(Element e)	Añade el elemento especificado al final de la lista.
<E> FXCollections <E>	*observableArrayList()*	Crea una nueva lista observable vacía que está respaldada por un *arrayList*.
PieChart	*PieChart()*	Constructor de gráfica de torta.
	void setTitle(String valor)	Establece el título de la gráfica.
	void setLegendSide(Side valor)	Establece el lado donde se colocará la leyenda de la gráfica.
	void setData(ObservableList <PieChart.Data>)	Establece los valores de la gráfica.
StackPane	*StakedPane(Node valor)*	Constructor del objeto *StackedPane*.
Scene	*Scene (Node valor)*	Construye una escena para un nodo específico.
Stage	*void setScene(Scene valor)*	Especifica la escena utilizada en este escenario.
	void setTitle(String valor)	Establece el título del escenario.
	void show()	Muestra la ventana.

Solución

Clase: GráficoTorta

```
package gráficatorta;

import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.scene.chart.PieChart;
```

```
/**
 * Esta clase denominada GráficaTorta contiene los porcentajes de
 * utilización de los lenguajes de programación en el año 2020
 * @version 1.0/2020
 */
public class GráficaTorta {

    /**
     * Método estático que define y obtiene los datos del PieChart
     * @return Los datos del PieChart
     */
    public static ObservableList<PieChart.Data> obtenerDatos() {
        // Crea una lista de datos para el PieChart
        ObservableList<PieChart.Data> data = FXCollections.
            observableArrayList();
        // Añade los datos al PieChart
        data.add(new PieChart.Data("Phyton", 25.7));
        data.add(new PieChart.Data("JavaScript", 17.8));
        data.add(new PieChart.Data("Go", 15));
        data.add(new PieChart.Data("TypeScript", 14.6));
        data.add(new PieChart.Data("Kotlin", 11));
        data.add(new PieChart.Data("Otros", 15.8));
        return data;
    }
}
```

Clase: PruebaGráfica

```
package gráficatorta;

import javafx.application.Application;
import javafx.collections.ObservableList;
import javafx.geometry.Side;
import javafx.scene.Scene;
import javafx.scene.chart.PieChart;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

/**
 * Esta clase denominada PruebaGráfica crea una gráfica de torta para
 * una serie de datos
 * @version 1.0/2020
```

```
 */
public class PruebaGráfica extends Application {
    PieChart gráfica; // Atributo que define un gráfico de torta

    /**
    * Método main que lanza la aplicación
    * @param args Parámetro que define los argumentos de la aplicación
    */
    public static void main(String[] args) {
        Application.launch(args);
    }

    /**
    * Método start que inicia la aplicación
    * @param stage El escenario donde se ejecutará la aplicación
    */
    @Override
    public void start(Stage stage) {
        gráfica = new PieChart(); // Crea la gráfica de torta
        gráfica.setTitle("Utilización de lenguajes de programación en
            2020"); // Establece título
        // Coloca una leyenda en el lado izquierdo
        gráfica.setLegendSide(Side.LEFT); /* Establece la ubicación de la
            leyenda */
        // Inicializa los datos de la gráfica
        ObservableList<PieChart.Data> datosGráfica = GráficaTorta.
            obtenerDatos();
        gráfica.setData(datosGráfica); // Establece los datos del PieChart
        StackPane root = new StackPane(gráfica); /* Agrega gráfica al
            contenedor */
        Scene scene = new Scene(root); // Crea escenario
        stage.setScene(scene); // Agrega escena al escenario
        stage.setTitle("Gráfica de torta"); // Establece título del escenario
        stage.show(); // Muestra el escenario (ventana)
    }
}
```

Diagrama de clases

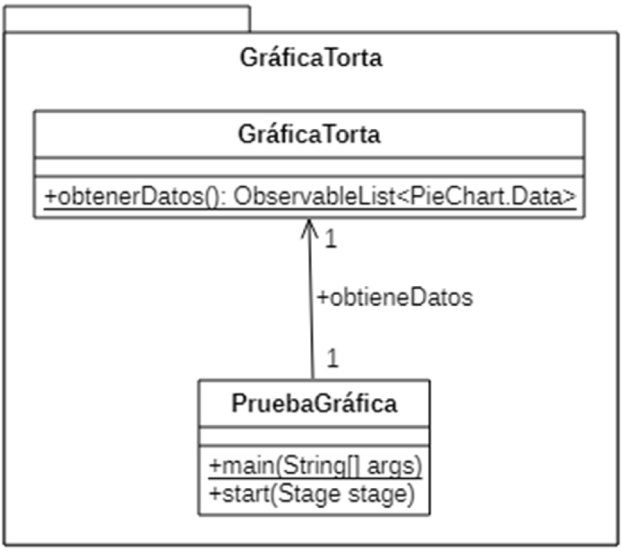


Figura 9.28. Diagrama de clases del ejercicio 9.7.

Explicación del diagrama de clases

El diagrama de clases muestra un paquete denominado "GráficaTorta", que contiene dos clases denominadas "GráficaTorta" y "PruebaGráfica".

En primer lugar, la clase GráficaTorta no tiene atributos pero cuenta con un método estático (indicado por su texto en subrayado) y público (indicado con el símbolo +) llamado obtenerDatos, el cual retorna un objeto *ObservableList* con los datos de la gráfica de torta.

En segundo lugar, la clase PruebaGráfica tampoco tiene atributos. Sin embargo, posee dos métodos: *start* que inicia la aplicación gráfica y el método *main* (punto de entrada a la aplicación). El método *main* es un método estático, por lo cual se representa con su texto subrayado. Los dos métodos son públicos, lo cual se indica con el símbolo +.

La clase PruebaGráfica utiliza la clase GráficaTorta, esto se expresa por medio de la relación de asociación entre las dos clases, denotada por una línea continua que conecta las dos clases y con el nombre de la relación que expresa que obtiene datos de esta clase. El sentido de la flecha que conecta las dos clases muestra la "navegabilidad" de la relación, que indica que

la clase PruebaGráfica conoce e invoca la clase GráficaTorta, pero la clase GráficaTorta no conoce nada sobre la clase PruebaGráfica.

Diagrama de objetos

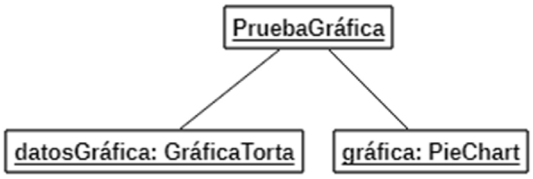


Figura 9.29. Diagrama de objetos del ejercicio 9.7.

Ejecución del programa

Figura 9.30. Ejecución del programa del ejercicio 9.7.

Ejercicios propuestos

- Desarrollar un programa para generar una ventana gráfica con un diagrama de barras (*BarChart*) con los datos de los lenguajes de programación del ejercicio anterior.
- Desarrollar una línea de tendencias (*ScatterChat*) con los datos de venta de un producto de la tabla 9.14.

Tabla 9.14. Datos de venta de un producto

Mes	Unidades vendidas
Enero	205
Febrero	304
Marzo	348
Abril	386
Mayo	404
Junio	498

Anexos

Este apartado presenta tres anexos que permiten ampliar los conocimientos sobre el lenguaje de programación Java, los diversos diagramas UML utilizados en el libro y diferentes herramientas *software* para desarrollar programas con Java.

Anexo 1. Sintaxis de Java

Este es una breve y rápida referencia del lenguaje Java. Si se desea una descripción más completa, se debe consultar la documentación oficial localizada en: https://docs.oracle.com/javase/7/docs/api/. Este anexo está adaptado de Lemay y Perkins (1996) y presenta los siguientes apartados:

- Palabras reservadas
- Comentarios
- Declaraciones de variables
- Asignación de variables
- Operadores
- Objetos
- *Arrays*
- Ciclos y condicionales
- Clases
- Métodos y constructores
- Paquetes, interfaces e importaciones
- Excepciones

Palabras reservadas

Tabla A1.1. Palabras reservadas

Palabras reservadas						
abstract	*class*	*final*	*implements*	*new*	*rest*	*throw*
boolean	*const*	*finally*	*import*	*null*	*return*	*throws*

Palabras reservadas						
break	*continue*	*float*	*inner*	*operator*	*short*	*transient*
byte	*default*	*for*	*instanceof*	*outer*	*static*	*try*
case	*do*	*future*	*int*	*package*	*sure*	*var*
cast	*double*	*generic*	*interface*	*private*	*switch*	*void*
catch	*else*	*goto*	*long*	*protected*	*synchronized*	*volatile*
char	*extends*	*if*	*native*	*public*	*this*	*while*

Comentarios

```
/* Este es un comentario multilínea */
// Este es un comentario de una sola línea
/** Comentario de Javadoc */
```

Declaraciones de variables

Tabla A1.2. Declaraciones de variables

Tipos	Formato
Enteros	*[byte | short | int | long] nombreVariable;*
Reales	*[float | double] nombreVariable;*
Caracteres	*char nombreVariable;*
Booleanos	*boolean nombreVariable;*
Clases	*NombreClase nombreVariable;*
Interfaces	*NombreInterface nombreVariable;*

Asignación de variables

Tabla A1.3. Asignación de variables

Tipo	Formato	Tipo	Formato
Asignación	variable = valor	Dividir y asignar	variable /= valor
Incremento pos	variable++	Módulo y asignar	variable %= valor
Incremento pre	++variable	AND y asignar	variable &= valor
Decremento pos	variable - -	OR y asignar	variable |= valor
Decremento pre	- -variable	XOR y asignar	variable ^= valor

Tipo	Formato	Tipo	Formato
Sumar y asignar	variable += valor	Desplazamiento a la izquierda y asignar	variable <<= valor
Restar y asignar	variable –= valor	Desplazamiento a la derecha y asignar	variable >>= valor
Multiplicar y asignar	variable *= valor	Desplazamiento a la derecha con relleno cero y asignar	variable >>>= valor

Operadores

Tabla A1.4. Operadores

Operador	Formato	Operador	Formato
Suma	argumento + argumento	OR lógico	argumento || argumento
Resta	argumento - argumento	NOT lógico	! argumento
Multiplicación	argumento * argumento	AND operador de bits	argumento y argumento
División	argumento / argumento	OR operador de bits	argumento | argumento
Módulo	argumento % argumento	XOR operador de bits	argumento ^ argumento
Menor que	argumento < argumento	NOT operador de bits	argumento ~ argumento
Mayor que	argumento > argumento	Desplazamiento a la izquierda	argumento << argumento
Menor o igual que	argumento <= argumento	Desplazamiento a la derecha	argumento >> argumento
Mayor o igual que	argumento >= argumento	Desplazamiento a la derecha, no tiene en cuenta el signo	argumento >>> argumento
Igual	argumento == argumento	Complemento	~ argumento
Diferente	argumento != argumento	Casting	(tipo) argumento
AND lógico	argumento && argumento	Instancia de	argumento *instanceof* Clase
Operador ternario	prueba ? operaciónVerdadera operaciónFalsa		

Objetos

Tabla A1.5. Objetos

Aspecto	Formato
Crear una nueva instancia	*new Class*()
Crear una nueva instancia con parámetros	*new Class*(argumento, argumento, argumento ...)
Variable de instancia	objeto.variable
Variable estática o de clase	objeto.variableClase Clase. variableClase
Método de instancia sin argumentos	objeto.método()
Método de instancia con argumentos	objeto.método(argumento, argumento, argumento, …)
Método de clase sin argumentos	objeto.métodoClase() Clase.métodoClase()
Método de clase con argumentos	objeto.métodoClase(argumento, argumento, argumento, ...) Clase.métodoClase(argumento, argumento, argumento, …)

Arrays

Tabla A1.6. *Arrays*

Elementos	Formato
Variable *array*	type nombreVariable[] type[] nombreVariable
Acceso a elemento	array[índice]
Nuevo objeto *array*	new tipo[númeroElementos]
Longitud del *array*	*array.length*

Ciclos y condicionales

Tabla A1.7. Ciclos y condicionales

Tipo	Formato	Tipo	Formato
Condicional	*if* (prueba) bloque	*continue*	*continue* [etiqueta]
Condicional con *else*	*if* (prueba) bloque *else* bloque	*do loop*	*do* bloque *while* (prueba)

Tipo	Formato	Tipo	Formato

switch	*switch* (prueba) {
	case valor : sentencia
	case value :
	sentencia
	default: sentencia
	}
		for	*for* (inicialización, prueba, actualización) bloque

while	*while* (prueba) bloque	Ciclos etiquetados	label:
break desde un ciclo o *switch*	*break* [etiqueta]		

Clases

Tabla A1.8. Clases

Elementos	Formato
Definición simple de clase	*class* nombreClase bloque
La clase no puede tener subclases	[*final*] class nombreClase bloque
La clase no puede ser instanciada	[*abstract*] class nombreClase bloque
Clase puede ser accedida por fuera del paquete	[*public*] class nombreClase bloque
Definición de subclase (herencia)	*class* nombreClase [extends nombreSuperClase] bloque

Métodos y constructores

Tabla A1.9. Métodos y constructores

Elementos	Formato
Método básico	tipoRetorno nombreMétodo() bloque
Método con parámetros	tipoRetorno nombreMétodo(parámetro, parámetro, ...) bloque
Parámetros de un método	tipo nombreParámetro
Método abstracto	[*abstract*] tipoRetorno nombreMétodo() bloque
Método estático o de clase	[*static*] tipoRetorno nombreMétodo() bloque
Método final	[*final*] tipoRetorno nombreMétodo() bloque
Control de acceso	[*public* | *private* | *protected*] tipoRetorno nombreMétodo()
Constructor básico	nombreClase() bloque
Constructor con parámetros	nombreClase (parámetro, parámetro, parámetro...) bloque

Elementos	Formato
Control de acceso	[*public* | *private* | *protected*] nombreMétodo() bloque
Referencia la objeto actual	*This*
Referencia a la superclase	*super*
Llamada al método de la superclase	super.nombreMétodo()
Llamada al constructor de la clase	*this*(...)
Llamada al constructor de la superclase	*super*(...)
Retorna un valor	*return* [valor]

Paquetes, interfaces e importación

Tabla A1.10. Paquetes, interfaces e importación

Elementos	Formato
Importación de clase específica	import nombrePaquete.nombreClase;
Importación de todas las clases públicas del paquete	import nombrePaquete.*;
Las clases del archivo pertenecen a un paquete	*package* nombrePaquete;
Interfaces	*interface* nombreInterface [extends nombreInterface] bloque [*public*] interface nombreInterface bloque [*abstract*] interface nombreInterface bloque

Excepciones

Tabla A1.11. Excepciones

Excepciones	Formato
Sentencias ejecutadas si una excepción es lanzada	*try* bloque *catch* (excepción) bloque [*finally* bloque]

Anexo 2. Lenguaje unificado de modelado (UML)

El lenguaje unificado de modelado (UML) es un lenguaje de modelado visual utilizado para visualizar, especificar, construir y documentar sistemas *software* (Booch, Rumbaugh y Jacobson, 2007). UML está conformado por

una colección de diagramas que se clasifican en diagramas estructurales y de comportamiento (Seidl *et al.*, 2015).

En este libro se ha utilizado la notación de cuatro diagramas UML:

- Diagramas de clases
- Diagramas de objetos
- Diagramas de máquinas de estados
- Diagramas de actividad

A continuación, se describirá brevemente la notación gráfica de cada uno de estos diagramas.

Diagramas de clases

Una clase representa un conjunto de objetos similares que se encuentran en el sistema. Las clases poseen atributos y métodos. Un atributo permite almacenar información conocida para todas las instancias, pero que tiene valores específicos diferentes para cada instancia. Por su parte, los métodos determinan un comportamiento específico en objetos concretos (Booch, Rumbaugh y Jacobson, 2017).

Las clases se representan por medio de rectángulos con tres compartimientos (figura A2.1). El primer compartimiento es para el nombre de la clase; el segundo, para el listado de atributos de la clase y el tercero, para el listado de métodos de la clase.

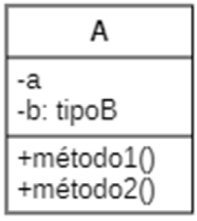
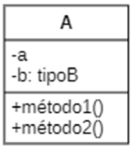
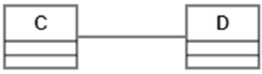
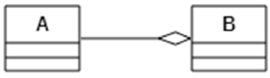
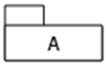


Figura A2.1. Notación de una clase en UML

Los diagramas de clase UML son diagramas estructurales. Los diagramas de clases permiten modelar la estructura estática de un sistema describiendo los elementos de este y sus relaciones entre ellos (Rumpe, 2016). Los diagramas de clase son los diagramas UML más usados.

La notación gráfica de los diagramas de clases UML es extensa. En la tabla A2.1 se presenta un resumen de los conceptos más importantes de los diagramas de clase utilizados en los ejercicios desarrollados en este libro.

Tabla A2.1. Conceptos de diagramas de clases UML

Nombre	Notación	Descripción
Clase		Descripción de la estructura y comportamiento de un conjunto de objetos.
Clase abstracta		Clases que no pueden ser instanciadas. El nombre de la clase está en cursiva.
Asociación		Relaciones entre clases.
Composición		Relaciones partes-todo dependientes (A es parte de B; si B es eliminado, las instancias relacionadas de A son eliminadas).
Agregación		Relaciones partes-todo (A es parte de B; si B es eliminado, las instancias relacionadas de A no son eliminadas).
Generalización		Relación de herencia (A hereda de B).
Interface		Elemento que declara un conjunto de métodos sin su implementación. Especifica un servicio proporcionado por una clase (A implementa B).
Paquetes		Elemento organizativo de diagramas, puede agrupar elementos de cualquier tipo si están relacionados semánticamente.
Clase genérica		Plantilla de clase que se puede parametrizar con uno o más tipos de datos.

Fuente: adaptado de Seidl *et al.* (2015).

Diagramas de objetos

En UML se describen los objetos concretos de un sistema y sus relaciones (enlaces) utilizando diagramas de objetos.

Los objetos representan formas concretas de las clases y son referidos como instancias. Un objeto tiene una identidad única y una serie de características que lo describen con más detalle. Este generalmente interactúa y se comunica con otros objetos. Las relaciones entre objetos se denominan enlaces. Las características de un objeto incluyen sus características estructurales (atributos) y su comportamiento (métodos). Valores concretos se asignan a los atributos en el diagrama de objetos; las operaciones usualmente no se indican (Schildt, 2018).

Los diagramas de objetos UML son diagramas estructurales. En un diagrama de objetos, un objeto se muestra como un rectángulo que se puede subdividir en dos compartimentos. El primer compartimento contiene información en la forma nombreObjeto: Clase (figura A2.2[a]). La segunda notación válida es el nombre del objeto sin colocar su tipo (figura A2.2[b]). La última notación es un objeto anónimo que especifica su tipo, pero no su nombre (figura A2.2[c]) (Booch, Rumbaugh y Jacobson, 2017).

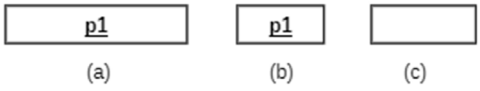


Figura A2.2. Notación de objetos

Si el rectángulo tiene un segundo compartimiento, este contiene los atributos del objeto y los valores actuales de sus atributos. Un enlace se representa como una línea continua que conecta los objetos que están relacionados entre sí (figura A2.3).

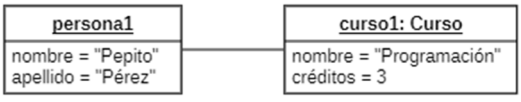


Figura A2.3. Notación de objetos relacionados y con valores de sus atributos

En la tabla A2.2 se presenta un resumen de los conceptos más importantes de los diagramas de objetos.

Tabla A2.2. Conceptos principales de los diagramas de objetos UML

Nombre	Notación	Descripción
Objeto		Instancia de una clase
Enlace		Relaciones entre objetos

Fuente: adaptado de Seidl *et al.* (2015).

Diagramas de máquinas de estado

Los objetos pasan por diferentes estados a lo largo de su ciclo de vida. Estos cambios (de un estado a otro) se representan en UML utilizando un diagrama de máquinas de estado que describen el comportamiento permitido a un objeto en forma de posibles estados y transiciones de estado iniciados por varios eventos (Booch, Rumbaugh y Jacobson, 2017).

El diagrama de máquinas de estado UML es un diagrama de comportamiento. Este es un gráfico con estados como nodos y transiciones de estado como flechas (Seidl *et al.*, 2015). Un estado se muestra como un rectángulo con esquinas redondeadas y se etiqueta con el nombre del estado (figura A2.4)

Figura A2.4. Notación gráfica de un estado en UML

El cambio de un estado a otro se denomina transición. Esta se representa con una flecha que indica la dirección de la transición. De acuerdo con Booch, Rumbaugh y Jacobson (2017), se pueden especificar varias propiedades para una transición (figura A2.5).

- **El evento** que desencadena la transición de estado (e).
- **Condición de guarda** que permite la ejecución de la transición (g).
- **Actividades** ejecutadas durante el cambio al estado objetivo (a).

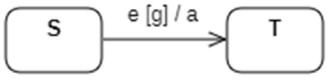
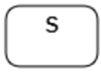
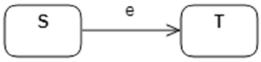
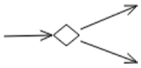
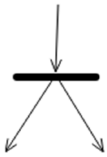
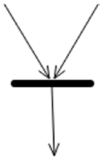


Figura A2.5. Notación gráfica para una transición

En la tabla A2.3. se presenta un resumen de los conceptos más importantes de los diagramas de máquinas de estado.

Tabla A2.3. Conceptos en diagramas de máquinas de estado UML

Nombre	Notación	Descripción
Estado		Descripción de un tiempo específico en el que un objeto se encuentra durante su ciclo de vida.
Transición		Transición de estado desde un estado fuente S a un estado objetivo T.
Estado inicial		Inicio de un diagrama de máquinas de estados.
Estado final		Fin de un diagrama de máquinas de estados
Nodo decisión		Nodo desde el que pueden proceder múltiples transiciones alternativas.
Nodo de paralelización		División de una transición en múltiples transiciones paralelas.
Nodo de sincronización		Fusión de múltiples transiciones paralelas en una sola transición.

Fuente: adaptado de Seidl *et al.* (2015).

Diagramas de actividad

Los diagramas de actividad UML se enfocan en modelar aspectos del procesamiento procedimental de un sistema especificando el flujo de control y de datos entre diferentes pasos requeridos para completar la actividad

(Seidl *et al.*, 2015). Su origen se remonta a diagramas para describir procesos de negocio.

Un diagrama de actividad permite especificar el comportamiento definido por el usuario en forma de actividades. Un diagrama de actividad puede describir la implementación de un caso de uso o de un método específico (Booch, Rumbaugh y Jacobson, 2017). Sin embargo, en este libro se utilizan para definir el comportamiento de un algoritmo donde las actividades toman la forma de instrucciones del lenguaje de programación.

Una actividad se denota como un rectángulo con los bordes redondeados. Las actividades se conectan por medio de flechas dirigidas, que expresan el orden en que las actividades son ejecutadas. Los diagramas de actividad deben poseer un nodo inicial y un nodo final. El primero se representa con un círculo negro e indica que la ejecución del flujo de actividades comienza y el segundo, con un círculo negro que tiene un círculo blanco concéntrico exterior e indica que el flujo de actividades ha finalizado (figura A2.6).

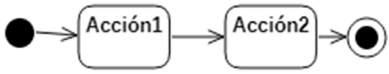


Figura A2.6. Notación gráfica para actividades con un nodo inicial y un nodo final

Las flechas que conectan actividades pueden contener condiciones de guarda que estipulan que, si cierta condición se cumple, la actividad de origen finaliza y la siguiente actividad comienza a ejecutarse (figura A2.7).

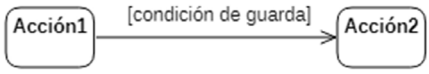


Figura A2.7. Notación gráfica para una condición de guarda entre actividades

En un diagrama de actividad se pueden modelar ramas alternativas de ejecución por medio de nodos de decisión que se representan por medio de rombos blancos con una flecha de entrada y una o varias flechas de salida (figura A2.8). Si se desea volver a unir ramas alternativas se pueden utilizar los nodos de fusión. Este nodo también se representa como un rombo, pero con múltiples flechas entrantes y solo una flecha de salida.

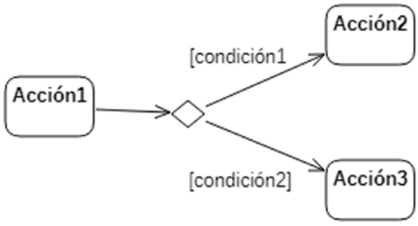
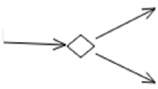
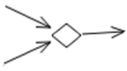


Figura A2.8. Notación gráfica para un nodo de decisión

En la tabla A2.4 se presenta un resumen de los conceptos más importantes de los diagramas de actividad utilizados en este libro.

Tabla A2.4. Conceptos de diagramas de actividad UML

Nombre	Notación	Descripción
Nodo actividad/ acción		Las actividades se descomponen. Las acciones son atómicas (no se pueden descomponer).
Nodo inicial		Inicio de la ejecución de una actividad.
Nodo final		Fin de la ejecución de todas las rutas de una actividad.
Nodo de decisión		Divide la ruta de ejecución en dos o más rutas alternativas de ejecución.
Nodo de fusión		Fusión de dos o más rutas alternativas de ejecución.
Flecha		Conexión entre los nodos de una actividad.

Fuente: adaptado de Seidl *et al.* (2015).

Anexo 3. Herramientas

Este anexo revisa algunas herramientas que pueden utilizarse para desarrollar *software* utilizando el lenguaje de programación Java.

En primer lugar, se presentará la instalación del *Java Development Kit* JDK y las principales herramientas de línea de comandos. Después se describen

brevemente herramientas basadas en GUI. Por último, se presenta un listado de otras herramientas que pueden utilizarse y que apoyan el desarrollo de proyectos *software*.

Instalación del JDK

Para desarrollar en el lenguaje de programación Java es un requisito instalar el *Java Development Kit (JDK)*. Para instalarlo se deben seguir los siguientes pasos:

1. Por medio de un navegador web dirigirse a la siguiente URL: http://www.oracle.com/technetwork/java/javase/downloads/index.html.
2. Identificar el último *release* disponible para Java SE y seleccionar la opción JDK.
3. Aceptar el acuerdo de licencia. Seleccionar el instalador más adecuado según la arquitectura del sistema operativo.
4. Una vez descargado el instalador, se debe proceder a su ejecución. Se visualizará un asistente que ayudará en dicho proceso.
5. Cuando finalice el proceso de instalación, en el equipo se debe crear un directorio *Java* (C:\Program Files\Java), el cual a su vez contendrá los directorios *jreVersion y jdkVersión*, donde versión corresponde al *release* descargado.
6. Crear la variable de entorno JAVA_HOME. Ir a "Equipo", clic secundario, "Propiedades" -> "Configuración avanzada del sistema" -> Pestaña "Opciones avanzadas" -> "Variables de entorno". En la sección "Variables del Sistema", seleccionar "Nueva" (figura A3.1).

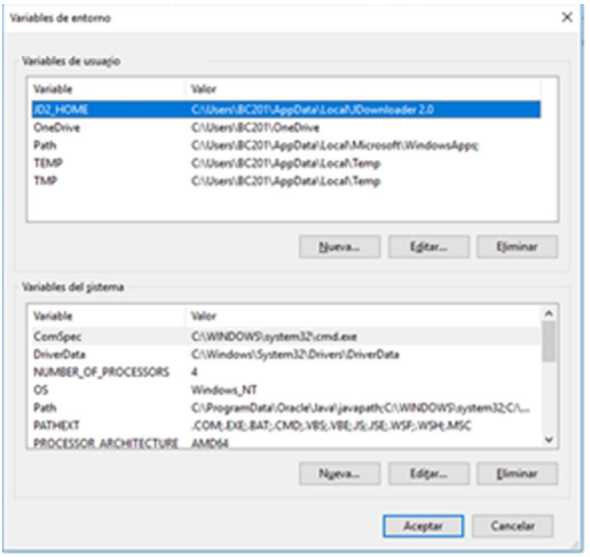


Figura A3.1. Variables de entorno

7. En la venta emergente, ingresar en el campo "Nombre" el valor JAVA_HOME. En el campo "Valor de la variable", la localización del directorio *jdk*. Por último, seleccionar la opción "Aceptar" (figura A3.2).

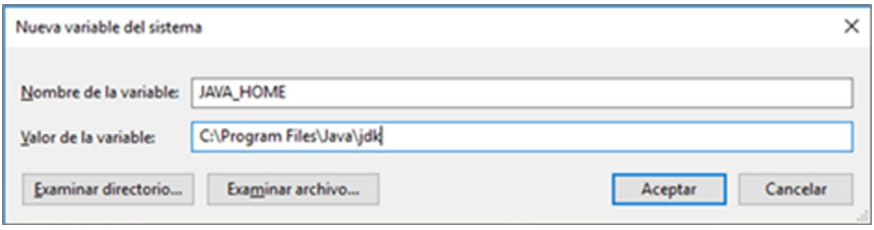


Figura A3.2. Nueva variable de entorno

8. En la sección "Variables del sistema", localizar la variable Path, seleccionar la opción "Modificar". Al final del valor del campo "Valor de la variable", ingresar la siguiente cadena ";%JAVA_HOME%\bin" (sin comillas). Seleccionar la opción "Aceptar" (figura A3.3).

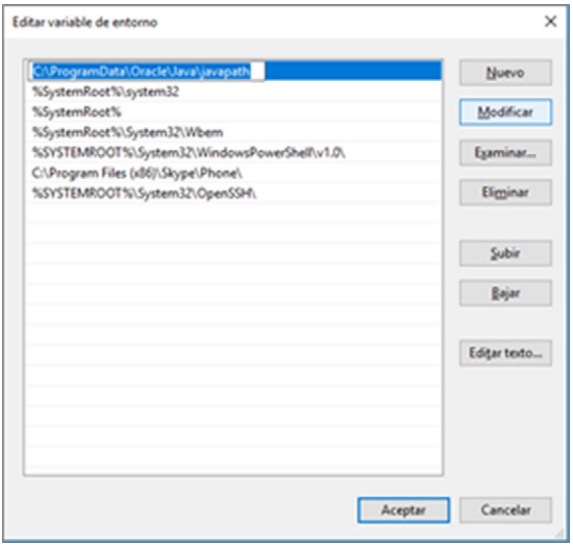


Figura A3.3. Editar variable de entorno

9. Verificar la instalación. Para ello, abrir una consola de comandos e ingresar el siguiente comando: "java –versión". En pantalla deberá aparecer un reporte con la versión instalada.

Herramientas de línea de comandos

Existen muchas herramientas basadas en líneas de comando en Java. En este anexo se describirán brevemente las más utilizadas.

Comando javac

javac es el compilador de código fuente de Java. Este comando produce *bytecode* (archivos .class) a partir de archivos fuente .java.

En los actuales proyectos de desarrollo basado en Java, este comando no se usa directamente, ya que es de bajo nivel y difícil de manejar en códigos que están conformados por muchas clases. Los entornos de desarrollo integrados (IDE) trabajan *javac* automáticamente o tienen compiladores incorporados que se ejecutan mientras se escribe el código.

El formato del comando javac es:

javac nombreArchivo.java [opciones]

si se desea compilar todos los archivos de una carpeta:

*javac *.java*

Algunas de las opciones permitidas son:

Tabla A3.1. Opciones comando javac

Opción	Descripción
classpath <path>	Específica donde encontrar los archivos .class. Se utiliza si el programa usa archivos .class que está almacenados en otra carpeta.
-d <directorio>	Especifica dónde colocar los archivos .class generados.
-g	Genera información de depuración.
-help	Imprime un resumen de las opciones estándar.
-nowarn	No genera avisos de advertencia de errores.
-sourcepath <path>	Especifica dónde encontrar los archivos fuente de java.
-target <release>	Genera archivos .class para una versión específica de máquina virtual.
-version	Proporciona información de la versión.

▸ Comando java

Una vez se haya generado el archivo .class se puede ejecutar el programa utilizando el comando *java*. Este comando inicia la ejecución de la máquina virtual de Java y tiene el siguiente formato:

java nombreArchivo

▸ Comando jar

El comando *jar* empaqueta toda la información del programa (directorios, clases, archivos de imágenes, etc.) en un único archivo con extensión .jar.

Para crear un archivo .jar, todas las clases deben estar compiladas. Se debe ubicar la línea de comandos en el directorio que tiene los archivos .class e ingresar el siguiente comando:

*jar -cf nombreArchivo.jar *.class*

El comando "c" indica que se va a crear un archivo con el nombre indicado en el siguiente parámetro y la opción "f" que inmediatamente direcciona a los nombres de los archivos .class o con el "*" se indican todos.

Para ejecutar un archivo .jar se debe estar ubicado en el directorio donde se encuentra el archivo e ingresar el siguiente comando:

java -jar nombreArchivo.jar

Herramienta javadoc

Esta herramienta permite generar la documentación de un programa Java. Para ello, el código debe estar comentado siguiendo los siguientes formatos:

```
/**
* Comentario en Javadoc
*/
```

La documentación de clases incorpora etiquetas predefinidas como el autor y la versión. Por ejemplo:

```
/**
* Esta clase denominada Persona define una.
* @author nombre completo del autor
* @version 2.1
*/
```

La documentación de métodos utiliza el mismo formato, pero tiene etiquetas adicionales, tales como:

Tabla A3.2. Etiquetas herramienta javadoc

Etiqueta	Descripción
@param	Definición del parámetro del método. Es requerido para todos los parámetros del método. Uno por cada línea con su nombre y descripción.
@return	Informa acerca de lo que devuelve el método. No aplica a constructores y métodos que no devuelven nada (*void*).
@see	Relación con otros métodos o clases.

Etiqueta	Descripción
@deprecated	Indica que el método es obsoleto, no se recomienda utilizar el método ya que es propio de versiones anteriores.
@throw	Indica el tipo de excepción que se puede generar durante la ejecución del método.

Una vez todo el código esté documentado utilizando estas etiquetas, se procede a generar la documentación utilizando el siguiente comando:

*javadoc -d nombreDirectorio/-use *.java*

El parámetro "d" indica el nombre del directorio donde se generará la documentación y el parámetro "use" indica los directorios donde se localizan los archivos .java.

Herramientas basadas en GUI

Con las herramientas de líneas de comandos se puede compilar programas, ejecutarlos y depurarlos. Solo se necesita un editor de texto para escribir el código fuente de los programas. Para facilitar este proceso, se utilizan herramientas basadas en GUI denominadas entornos de desarrollo integrado (IDE, por sus siglas en inglés, *Integrated Development Environment*). A continuación, se presentan algunos de los IDE más utilizados en la industria.

- **Eclipse**

Este IDE libre y *open source* es ampliamente utilizado para desarrollar aplicaciones en Java y está disponible en varios idiomas. Sus usuarios pueden extender su funcionalidad, ya que posee una gran cantidad de *plugins*. Inicialmente fue desarrollado por IBM y, actualmente, es desarrollado por una organización independiente y sin ánimo de lucro, la Fundación Eclipse. Posee un editor de texto con analizador sintáctico y la compilación es automática. Eclipse tiene una vasta comunidad de usuarios alrededor del mundo. El sitio oficial de Eclipse es: https://www.eclipse.org/. En la figura A3.4 se muestra una pantalla del entorno de desarrollo Eclipse.

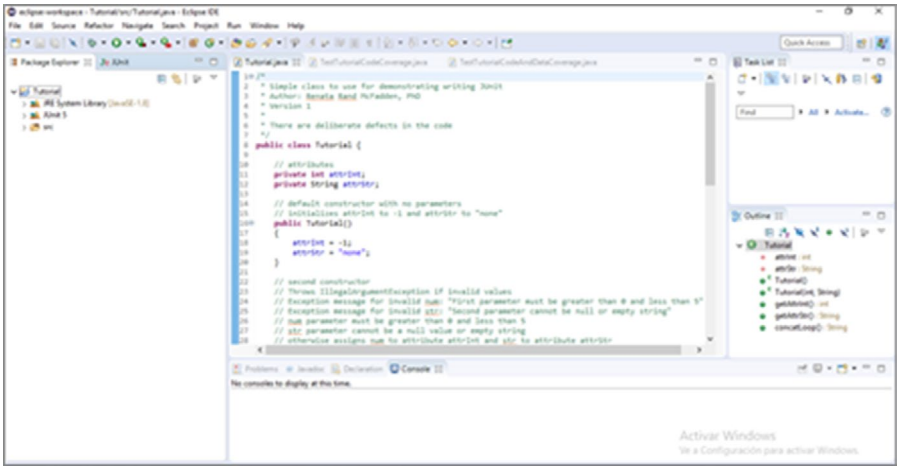


Figura A3.4. Herramienta Eclipse

Netbeans

El entorno de desarrollo integrado Netbeans es un producto libre y gratuito utilizado principalmente para el lenguaje de programación Java. Su creador y patrocinador es Sun MicroSystems, aunque esta empresa pasó luego a formar parte de Oracle Corporation. Tiene una estructura modular ampliable mediante *plugins*, que tiene configuraciones predefinidas para diferentes lenguajes de programación. El sitio web oficial de Netbeans es: https://netbeans.apache.org/. En la figura A3.5 se muestra una pantalla del entorno de desarrollo Eclipse.

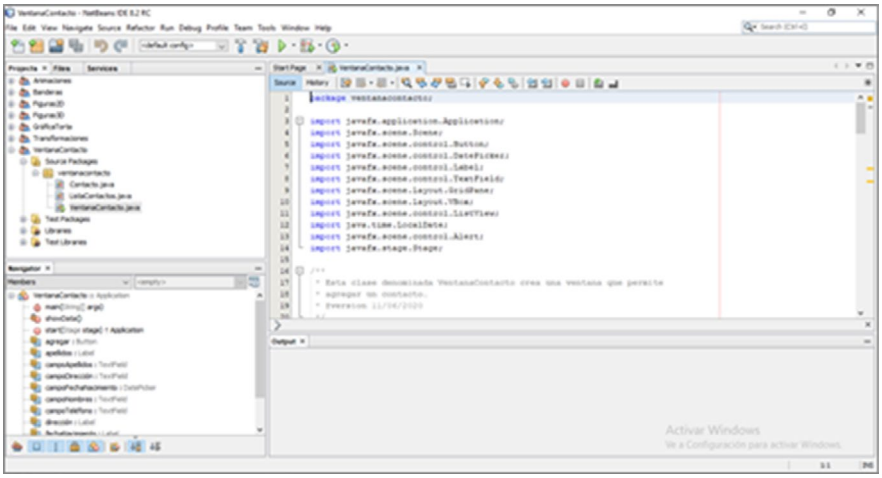


Figura A3.5. Herramienta Netbeans

IntelliJ IDEA

Este entorno de desarrollo para Java se caracteriza por no ser completamente un proyecto de *software* libre, es decir, es un producto comercial. Sin embargo, una versión reducida denominada *Community* se puede obtener gratuitamente. Fue desarrollado por la empresa JetBrains y puede incorporar soporte para otros lenguajes de programación. El sitio web oficial de IntelliJ IDEA es: https://www.jetbrains.com/idea/. En la figura A3.6 se muestra una pantalla del entorno de desarrollo.

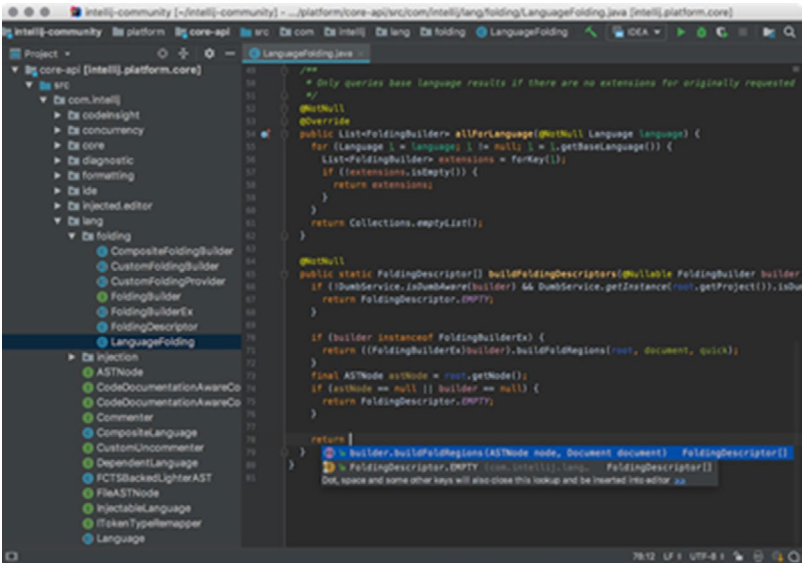


Figura A3.6. Herramienta IntelliJ IDEA

▶ JGrasp

Este entorno de desarrollo es más ligero que los tres anteriores y, por lo tanto, tiene una funcionalidad intermedia. Fue desarrollado por la Universidad de Auburn (Alabama, EE. UU.) para proporcionar la generación automática de visualizaciones de *software* y así mejorar la comprensión del mismo. Produce diagramas de estructura de control (CSD), gráficos de perfil de complejidad (CPG) y diagramas de clase UML para Java. El sitio web oficial de JGrasp es: https://www.jgrasp.org/. En la figura A3.7 se muestra una pantalla del entorno de desarrollo.

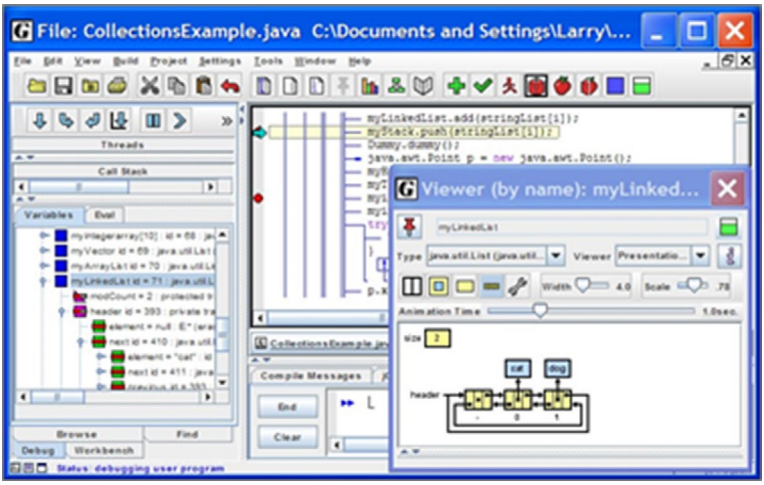


Figura A3.7. Herramienta JGrasp

- **BlueJ**

Este entorno de desarrollo elaborado específicamente para Java es ampliamente utilizado por estudiantes de todo el mundo para aprender Java. Fue desarrollado por la Universidad de Kent (Reino Unido). Su escritorio permite instanciar los objetos (de una manera visual) a partir de las clases definidas, invocar en forma explícita los métodos definidos, inspeccionar el estado del objeto, etc. El autor lo ha utilizado con éxito en sus clases de programación orientada a objetos y recomienda su uso como herramienta inicial para introducir conceptos básicos del paradigma OO. El sitio web oficial de Bluej es: https://www.bluej.org/. La figura A3.8 muestra una pantalla del entorno de desarrollo.

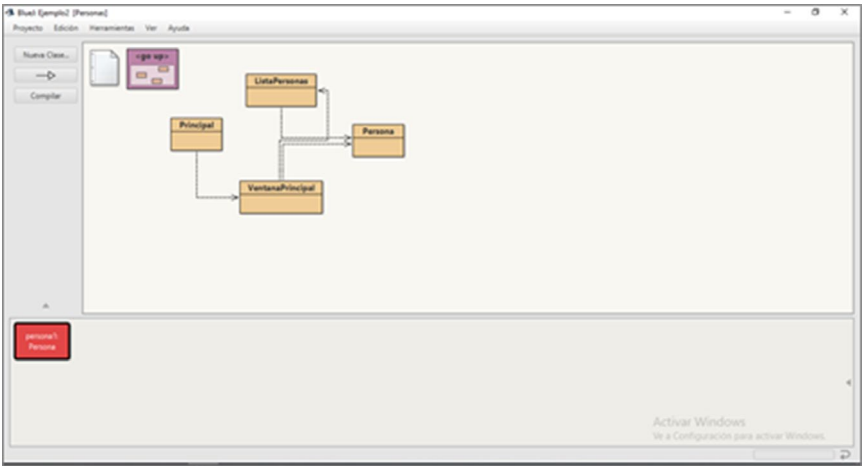


Figura A3.8. Herramienta Bluej

Otras herramientas

Existe un espectro diverso de herramientas que necesitan los desarrolladores para culminar con éxito un proyecto *software*. A continuación, se presenta un pequeño listado de herramientas que contribuyen a desarrollar un proyecto. No se describirán detalladamente porque son temas relacionados concretamente con la ingeniería de *software* y están fuera del alcance de este anexo.

- Maven: para la gestión y construcción de proyectos que utilizan un repositorio de artefactos *software*. Disponible en https://maven.apache.org/
- Jira: para gestionar el flujo de trabajo de proyectos ágiles de desarrollo de *software*. Disponible en https://www.atlassian.com/es/software/jira
- Ant: para automatizar el proceso de compilación de programas Java. Disponible en http://ant.apache.org/index.html
- JUnit: para automatizar las pruebas unitarias de código fuente realizado con Java. Disponible en https://junit.org/junit5/
- Git: para establecer un sistema de control de versiones del código fuente de los programas. Disponible en https://git-scm.com/
- Jenkins: para establecer un sistema de integración continua y periódica del código fuente. Disponible en https://www.jenkins.io/

Enlaces web de interés

API Java:
https://docs.oracle.com/javase/7/docs/api/
Ejercicios de Java:
http://www.sarmaroof.com/
https://www.ntu.edu.sg/home/ehchua/programming/java/J3f_OOPExercises.html
Geeks for Geeks:
https://www.geeksforgeeks.org/java/
Guía de Java:
https://www.javaguides.net/
Java World:
https://www.javaworld.com/
Java Concept of the Day. Java Tutorial Site for Beginners:
https://javaconceptoftheday.com/
Tutoriales de Java:
https://docs.oracle.com/javase/tutorial/uiswing/index.html
W3Schools:
https://www.w3schools.com/java/default.asp

Referencias

Altadill-Izurra, P.X., y Pérez-Martínez, E. (2017). *Java limpio: programación Java y buenas prácticas de desarrollo*. Estados Unidos: Createspace Independent Publishing Platform.

API Java (2020). *Java™ Platform, Standard Edition 7 API Specification*. Recuperado de https://docs.oracle.com/javase/7/docs/api/

Arroyo-Díaz, C. (2019a). *Programación en Java I: aplicaciones robustas y confiables. El entorno de programación-sintaxis-elementos-estructuras de control* [EPub]. Buenos Aires, Argentina: Creative Andina Corp.

Arroyo-Díaz, C. (2019b). *Programación en Java II: aplicaciones robustas y confiables. Clases-construcción de objetos-encapsulamiento-herencia* [EPub]. Buenos Aires, Argentina: Creative Andina Corp.

Arroyo-Díaz, C. (2019c). *Programación en Java III: aplicaciones robustas y confiables. Clases abstractas-interfaces-manejo de excepciones-rRecursividad* [EPub]. Buenos Aires, Argentina: Creative Andina Corp.

Bajracharya, K. (2019). *Polymorphism in Java. Geek for Geeks*. Recuperado de https://www.geeksforgeeks.org/polymorphism-in-java/

Black, A. P. (2013). Object-oriented programming: Some history, and challenges for the next fifty years. *Information and Computation*, 231, 3-20.

Bloch, J. (2017). *Effective Java*. Estados Unidos: Addison-Wesley Professional.

Booch, G., Rumbaugh, J., y Jacobson, I. (2017). *The Unified Modeling Language User Guide (Addison-Wesley Object Technology Series)*. Estados Unidos: Addison-Wesley Professional.

Cadenhead, R. (2017). *Java in 24 Hours, Sams Teach Yourself*. Indianápolis, Estados Unidos: Sams Publishing.

Cha, J. (2016). *Java: Learn Java in One Day and Learn It Well*. Estados Unidos: Learn Coding Fast Publishing.

Clark, N. (2017). *Java: Programming Basics for Absolute Beginners*. Estados Unidos: Createspace Independent Publishing Platform.

Deitel, H. M., y Deitel, P. J. (2017). *Java How to program, Early objects*. Estados Unido: Pearson Education.

Dea, C., Heckler, M., Grunwald, G., Pereda, J., y Phillips, S. M. (2014). *JavaFX 8: Introduction by example*. Nueva York, Estados Unidos: Apress.

Flanagan, D., y Evans, B. (2019). *Java in a Nutshell: A Desktop Quick Reference*, California, Estados Unidos: O'Really Media.

Horstmann, C. S. (2018). *Core Java Volume I. Fundamentals*. Michigan, Estados Unidos: Prentice Hall.

Hunt, J. (2013). *The Unified Process for Practitioners: Object-oriented Design, the UML and Java*. Estados Unidos: Springer.

Jain, D., y Mangal, K. (2019). *Constructos in Java. Geek for Geeks*. Recuperado de https://www.geeksforgeeks.org/constructors-in-java/

Joy, B., Bracha, G., Steele, G., Buckley, A., y Gosling, L. (2013). *The Java Language Specification*, Estados Unidos: Addison-Wesley Professional.

Kumar, J. (2019). *Inheritance in Java. Geek for Geeks*. Recuperado de https://www.geeksforgeeks.org/inheritance-in-java/

Köllinjg, M., y Barnes, D. (2013) *Programación orientada a objetos con Java usando Bluej*. Madrid, España: Pearson Universidad.

Lemay, L., y Perkins, C. L. (1996) *Teach Yourself SunSoft Java Workshop in 21 Days; With Cdrom*. Indianápolis, Estados Unidos: Macmillan Publishing Co., Inc.

Liang, Y. D. (2017). *Introduction to Java Programming and Data Structures, Comprehensive Version*. India: Pearson Education India.

Morelli, R., y Walde, R. (2012). *Java, Java, Java: Object-Oriented Problem Solving*. Connecticut, Estados Unidos: Pearson Education, Inc.

Murach, J., Boehm, A., y Delameter, M. (2017). *Murach's Java Programming*. Estados Unidos: Mike Murach & Associates.

Pressman, R. S., y Maxim. B. R. (2014). *Software engineering: a practitioner's approach*. Boston, Estados Unidos: McGrawHill Education.

Reyes, S., y Stepp, M. (2016). *Building Java Programs: A Back to Basics Approach*. Estados Unidos: Pearson.

Rumpe, B. (2016). *Modeling with UML. Language, Concepts, Methods*. Berlín, Alemania: Springer.

Seidl, M., Scholz, M., Huemer, C., y Kappel, G. (2015). *UML @ Classroom: An Introduction to Object-Oriented Modeling*. Heidelberg: Alemania: Springer.

Schildt, H. (2018). *Java: The Complete Reference*. Nueva York, Estados Unidos: McGraw-Hill Education.

Schildt, H., y Skrien, D. (2013). *Java Programming: A Comprehensive Introduction Core Java Volume I. Fundamentals*. Nueva York, Estados Unidos: Mac Graw Hill Education.

Sharan, K. (2015). *Learn JavaFX 8: building user experience and interfaces with Java 8*. NuevaYork, Estados Unidos: Apress.

Sharan, K. (2018) *Java APIs, Extensions and Libraries. With JavaFX, JDBC, jmod, jlink, Networking, and the Process API*. NuevaYork, EE. UU.: Apress.

Samoylov, N. (2019). *Learn Java 12 Programming: A step-by-step guide to learning essential concepts in Java*. Birmingham, Reino Unido: Packt Publishing.

Vos, J., Chin, S., Gao, W., Weaver, J. L., y Iverson, D. (2012). *Pro JavaFX 2*. Nueva York, Estados Unidos: Apress.

Vozmediano, A. M. (2017). *Java para novatos: cómo aprender programación orientada a objetos con Java sin desesperarse en el intento*. Volumen 3. Madrid, España: CreateSpace Independent Publishing Platform.

Yang, H. (2018). *Java Swing Tutorial. Herong's Tutorial Examples*. Estados Unidos: Independently published.

▶ Índice temático

A

B

D

E

Q

R

S

T

U

V

Ciencias de Gestión

Ejercicios de programación orientada a objetos con Java y UML

Hace parte de la Colección Ciencias de Gestión.
Se editó y diagramó en la Editorial
Universidad Nacional de Colombia,
en septiembre de 2021.
Bogotá, D. C., Colombia.
En su composición se utilizaron caracteres
ITC Berkeley Oldstyle Std 11/14 puntos.

Otros títulos de esta colección

- ***El lenguaje de las emociones en el aprendizaje. Análisis de la interactividad en un entorno educativo híbrido***
  Victoria Eugenia Valencia Maya

- ***Turismo transformador. Gestión del conocimiento y tecnologías digitales en el turismo***
  Marcelo López Trujillo,
  Carlos Eduardo Marulanda Echeverry y
  Carlos Hernán Gómez Gómez

- ***Análisis financiero corporativo***
  Alberto Antonio Agudelo Aguirre

- ***Sistema de gestión de seguridad de la información basado en la familia de normas ISO/IEC 27000***
  Francisco Javier Valencia Duque

- ***Los costos bajo NIIF y su implementación en las pymes***
  Ricardo Alfredo Rojas Medina

Ciencias de Gestión

El paradigma orientado a objetos es una forma de analizar, diseñar y construir sistemas *software* para afrontar la complejidad de los programas, entender los requisitos de estos y proponer soluciones a las problemáticas planteadas en el ámbito informático.
Este libro ofrece un conjunto de ejercicios para apoyar el aprendizaje de la programación orientada a objetos. Además, incorpora diagramas UML que muestran el diseño de las soluciones propuestas. Estas se presentan en el lenguaje de programación Java, que se ha consolidado como uno de los lenguajes de programación más populares a nivel mundial.
El conjunto de ejercicios propuesto contiene temas como: estructuras básicas de programación, clases y objetos, objetos especializados, herencia y polimorfismo, diferentes relaciones entre clases, excepciones, lectura y escritura de archivos e incluye, en su apartado final, el desarrollo de interfaces gráficas de usuario utilizando *swing* y JavaFX.
Cada ejercicio contiene apartados en los que se contempla: conceptos teóricos, objetivos de aprendizaje, enunciado del ejercicio, instrucciones Java utilizadas, código de la solución propuesta, diagrama de clases UML junto con su explicación, diagrama de objetos UML y varios ejercicios propuestos para reforzar los conceptos presentados. Todo ello orientado para que al lector se le facilite comprender el problema y entender la solución presentada.

9 789587 945751

editUNAB