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

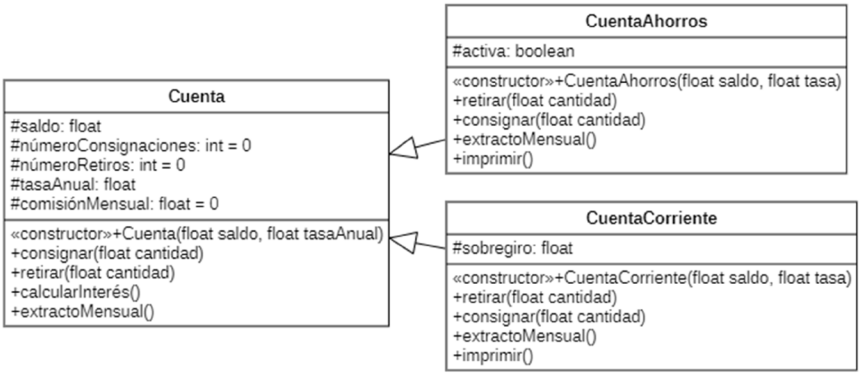

Figura 4.1. Diagrama de clases del ejercicio 4.1.

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

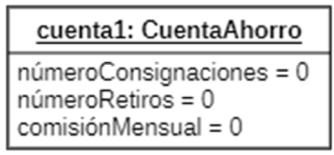

Figura 4.2. Diagrama de objetos del ejercicio 4.1.

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

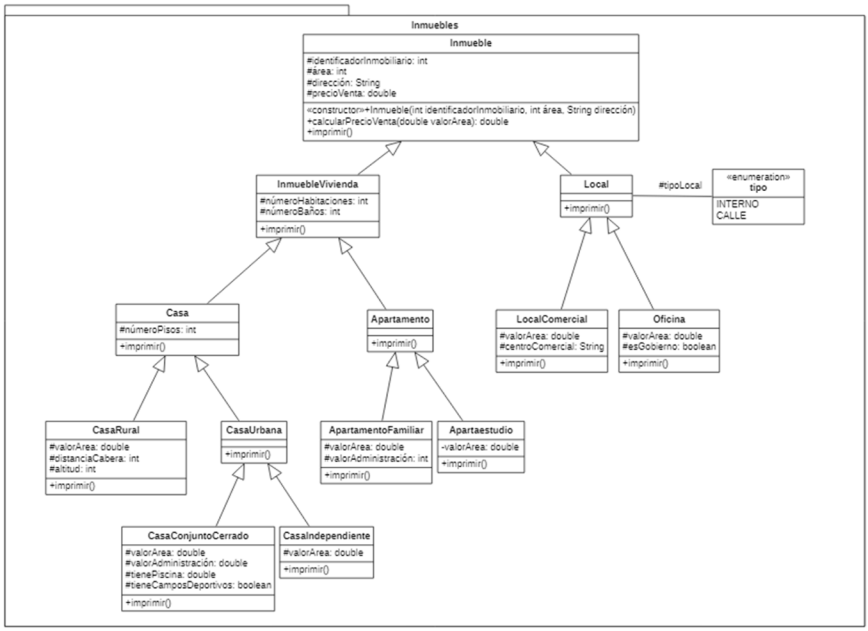

Figura 4.4. Diagrama de clases del ejercicio 4.2.

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

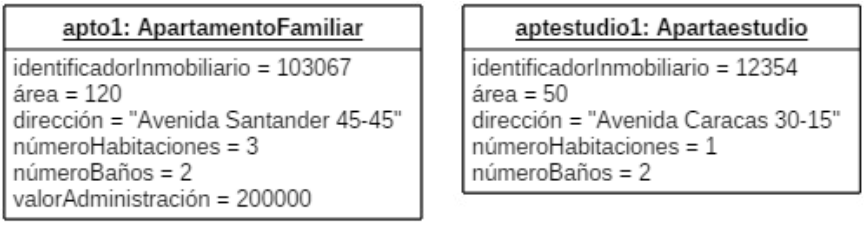

Figura 4.5. Diagrama de objetos del ejercicio 4.2.

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

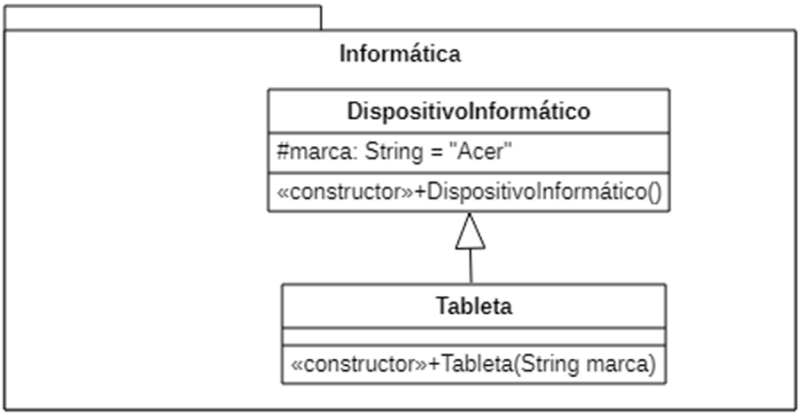

Figura 4.7. Diagrama de clases del ejercicio 4.3.

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

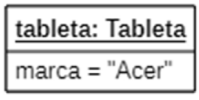

Figura 4.8. Diagrama de objetos del ejercicio 4.3.

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 .

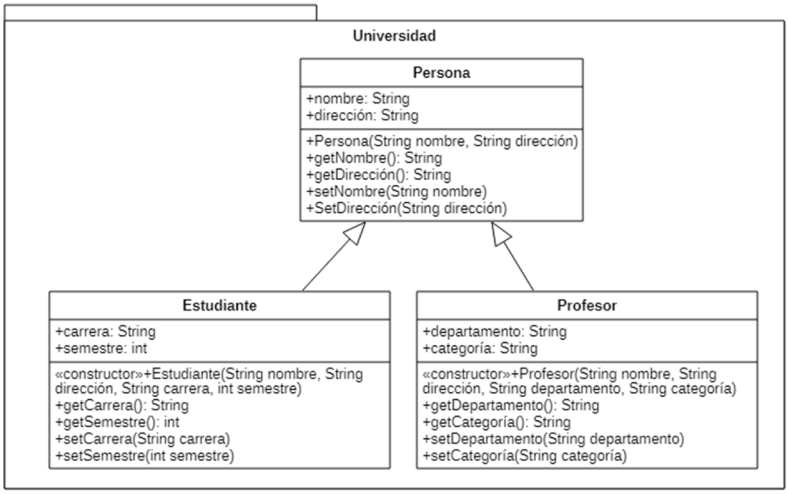

Figura 4.10. Diagrama de clases del ejercicio propuesto.

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

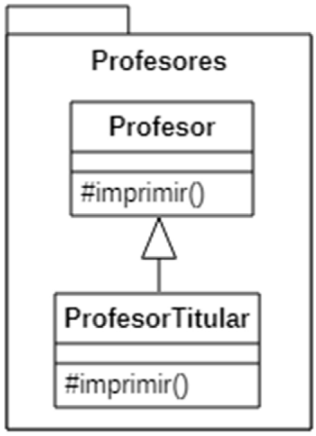

Figura 4.11. Diagrama de clases del ejercicio 4.4.

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

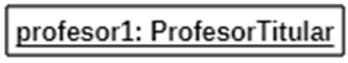

Figura 4.12. Diagrama de objetos del ejercicio 4.4.

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

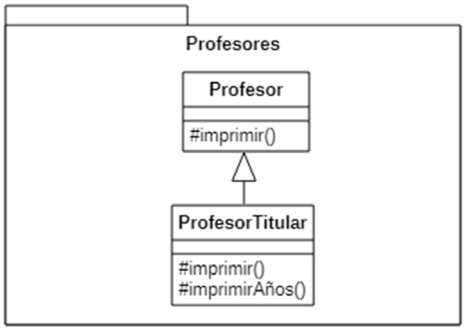

Figura 4.14. Diagrama de clases del ejercicio 4.5.

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

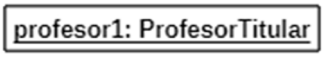

Figura 4.15. Diagrama de objetos del ejercicio 4.5.

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

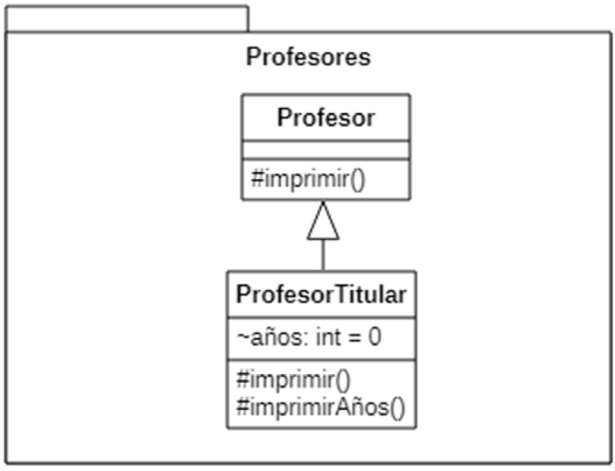

Figura 4.16. Diagrama de clases del ejercicio 4.6.

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

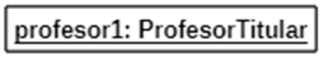

Figura 4.17. Diagrama de objetos del ejercicio 4.6.

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

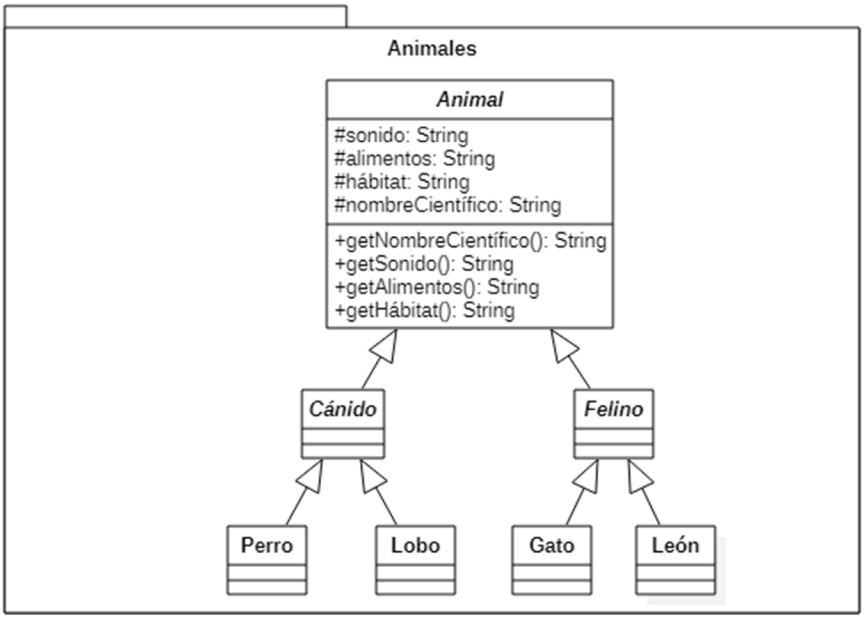

Figura 4.18. Diagrama de clases del ejercicio 4.7.

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

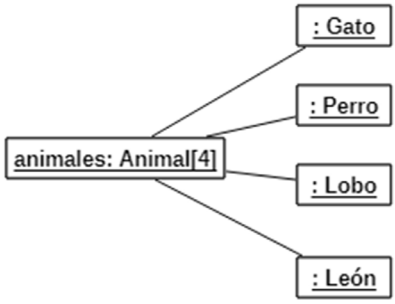

Figura 4.19. Diagrama de objetos del ejercicio 4.7.

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

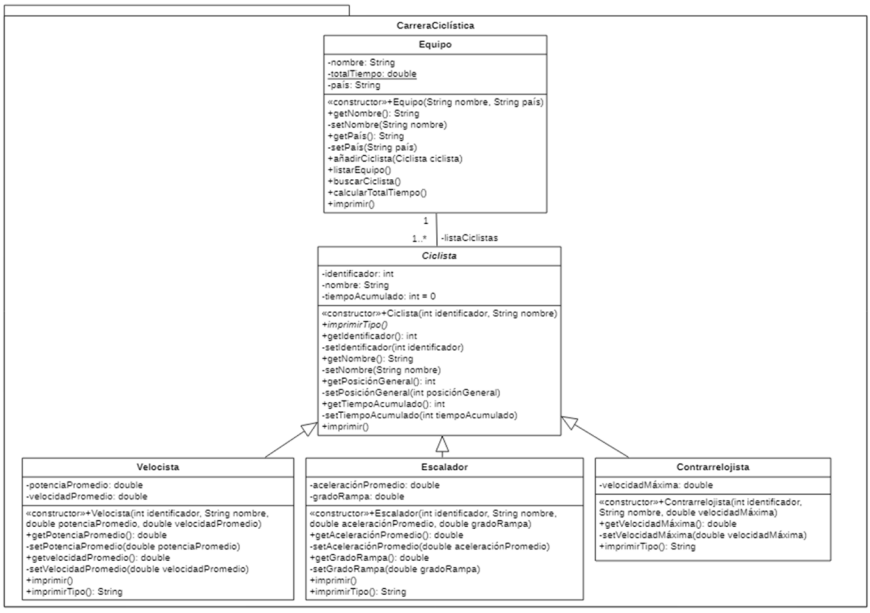

Figura 4.21. Diagrama de clases del ejercicio 4.8.

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

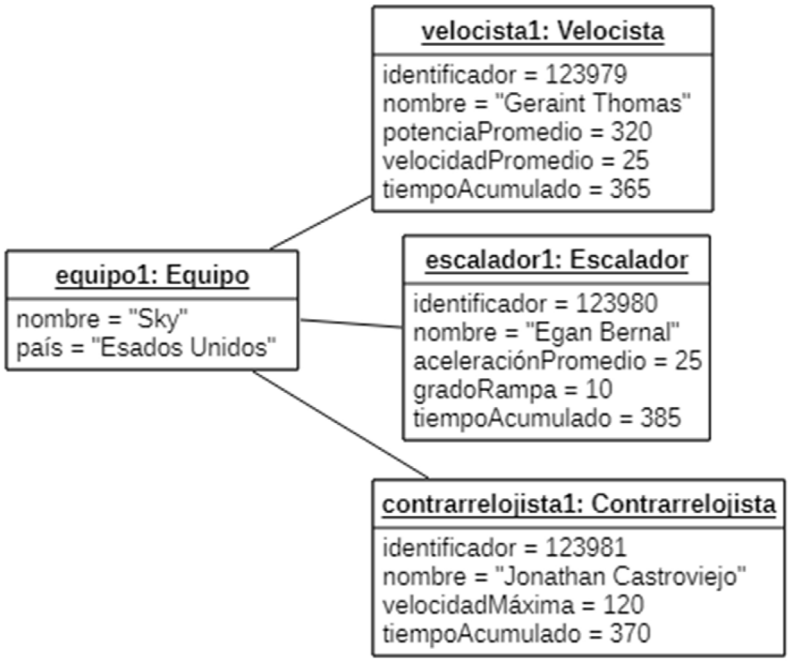

Figura 4.22. Diagrama de objetos del ejercicio 4.8.

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

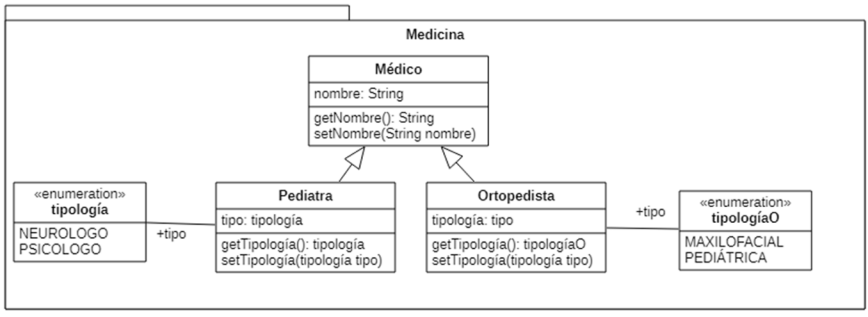

Figura 4.24. Diagrama de clases del ejercicio 4.9.

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

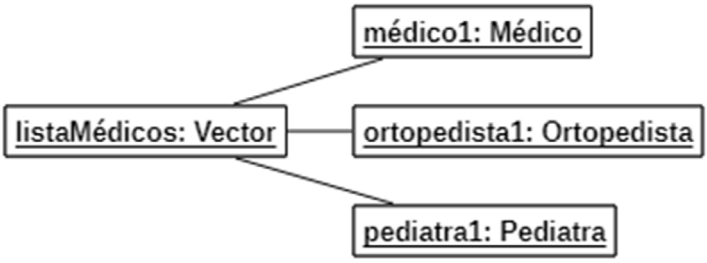

Figura 4.25. Diagrama de objetos del ejercicio 4.9.

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

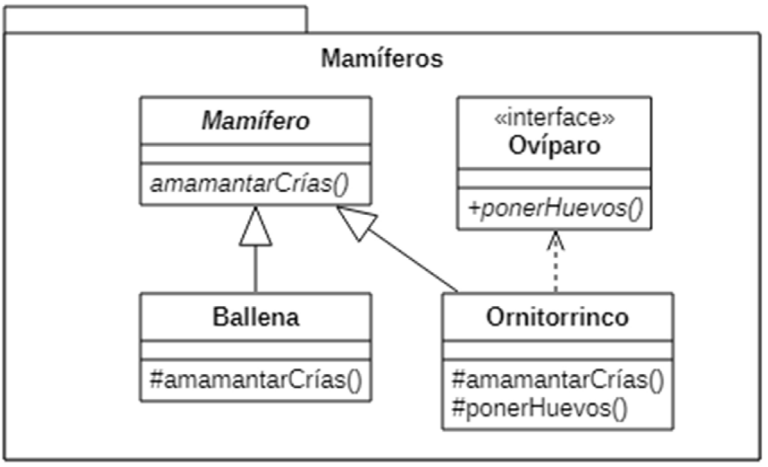

Figura 4.27. Diagrama de clases del ejercicio 4.10.

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

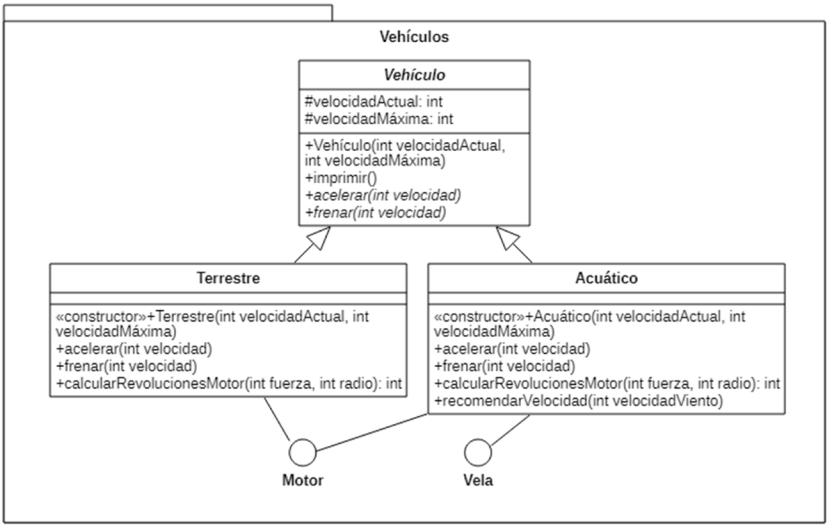

Figura 4.30. Diagrama de clases del ejercicio 4.11.

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

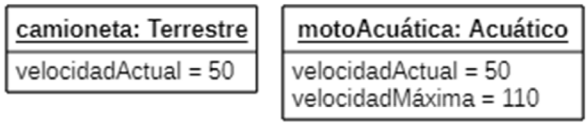

Figura 4.31. Diagrama de objetos del ejercicio 4.11.

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

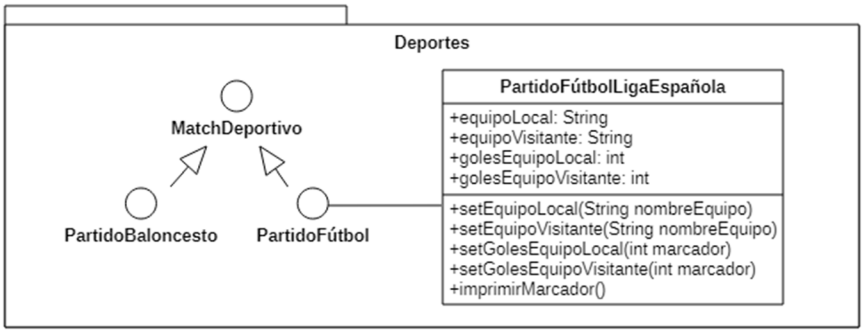

Figura 4.33. Diagrama de clases del ejercicio 4.12.

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.