



# FUNDAMENTOS DE PROGRAMACIÓN ORIENTADA A OBJETOS CON JAVA



Sandra Victoria Hurtado Gil

## Contenido

|   |    |
|---|----|
| Introducción.....   | 5  |
| 1. Objetos y Clases.....  | 6  |
| 1.1. Objetos.....   | 6  |
| 1.2. Clases .....   | 9  |
| 1.3. Terminología.....  | 10 |
| 1.4. Representación.....  | 10 |
| 1.5. Implementación en Java.....  | 11 |
| 1.6. Ejercicio resuelto .....   | 13 |
| 2. Introducción a Java.....   | 15 |
| 2.1. Contexto.....  | 15 |
| 2.2. Máquina virtual de Java.....   | 15 |
| 2.3. Ambientes de desarrollo.....   | 16 |
| 2.4. Estructura de un programa sencillo .....                             | 16 |
| 2.5. Comentarios .....  | 17 |
| 2.6. Identificadores .....  | 17 |
| 2.7. Tipos de datos .....   | 18 |
| 2.8. Variables .....  | 19 |
| 2.9. Constantes.....  | 20 |
| 2.10. Expresiones y operadores .....                                      | 20 |
| 2.11. Expresiones válidas .....   | 22 |
| 2.12. Ejercicio resuelto.....   | 23 |
| 3. Atributos .....  | 24 |
| 3.1. Definición .....   | 24 |
| 3.2. Modelado de atributos .....  | 24 |
| 3.3. Representación.....  | 26 |
| 3.4. Implementación en Java.....  | 26 |
| 3.5. Ejercicio resuelto .....   | 27 |
| 4. Instrucciones Básicas en Java: Condicionales, Lectura y Escritura..... | 29 |
| 4.1. Instrucciones de escritura .....                                     | 29 |
| 4.2. Instrucciones condicionales .....                                    | 31 |
| 4.3. Selección simple ( <i>if</i> ) .....                                 | 32 |
| 4.4. Selección doble ( <i>if-else</i> ) .....                             | 33 |
| 4.5. Selección múltiple ( <i>switch</i> ) .....                           | 35 |
| 4.6. Instrucciones de lectura.....  | 37 |
| 4.7. Ejercicio resuelto .....   | 40 |
| 5. Métodos.....   | 43 |
| 5.1. Definición .....   | 43 |
| 5.2. Modelado de métodos .....  | 44 |

|       |  |     |
|-------|--|-----|
| 5.3.  | Representación.....                                  | 45  |
| 5.4.  | Implementación en Java.....                          | 46  |
| 5.5.  | Ejercicio resuelto .....                             | 51  |
| 6.    | Referencias y Operador Punto .....                   | 54  |
| 6.1.  | Tipos de variables en Java.....                      | 54  |
| 6.2.  | Referencias.....                                     | 54  |
| 6.3.  | Uso de los métodos con el operador punto .....       | 58  |
| 6.4.  | Valor <i>null</i> .....                              | 60  |
| 6.5.  | Ejercicio resuelto .....                             | 62  |
| 7.    | Estructuras Repetitivas en Java .....                | 64  |
| 7.1.  | Definición .....                                     | 64  |
| 7.2.  | Ciclo <i>while</i> .....                             | 64  |
| 7.3.  | Ciclo <i>do-while</i> .....                          | 65  |
| 7.4.  | Ciclo <i>for</i> .....                               | 67  |
| 7.5.  | Instrucción <i>break</i> .....                       | 68  |
| 7.6.  | Ejercicio resuelto .....                             | 70  |
| 8.    | Referencia <i>this</i> .....                         | 72  |
| 8.1.  | Uso de métodos y atributos propios.....              | 72  |
| 8.2.  | Diferenciar atributos y parámetros .....             | 74  |
| 8.3.  | Ejercicio resuelto .....                             | 78  |
| 9.    | API de Java.....                                     | 80  |
| 9.1.  | Definición .....                                     | 80  |
| 9.2.  | Sentencia <i>import</i> .....                        | 80  |
| 9.3.  | Búsqueda en el API.....                              | 81  |
| 9.4.  | Ejercicio resuelto .....                             | 83  |
| 10.   | Métodos <i>Set</i> , <i>Get</i> y Constructores..... | 86  |
| 10.1. | Tipos de métodos .....                               | 86  |
| 10.2. | Métodos “get”: Acceso o consulta de un atributo..... | 86  |
| 10.3. | Métodos “set”: Modificación de un Atributo .....     | 89  |
| 10.4. | Constructor.....                                     | 91  |
| 10.5. | Varios métodos constructores .....                   | 94  |
| 10.6. | Constructor por defecto .....                        | 97  |
| 10.7. | Ejercicio resuelto.....                              | 98  |
| 11.   | <i>ArrayList</i> en Java.....                        | 101 |
| 11.1. | Uso de colecciones.....                              | 101 |
| 11.2. | Definición .....                                     | 101 |
| 11.3. | Uso de <i>ArrayList</i> .....                        | 102 |
| 11.4. | Ciclo mejorado para recorrer listas.....             | 105 |
| 11.5. | Ejercicio resuelto.....                              | 106 |

|       |  |     |
|-------|--|-----|
| 12.   | Asociaciones entre Clases.....                       | 109 |
| 12.1. | Contexto y definición.....                           | 109 |
| 12.2. | Representación .....                                 | 110 |
| 12.3. | Implementación en Java .....                         | 111 |
| 12.4. | Ejercicio resuelto.....                              | 117 |
| 13.   | Paquetes y Visibilidad .....                         | 124 |
| 13.1. | Definición y representación .....                    | 124 |
| 13.2. | Nombres únicos .....                                 | 125 |
| 13.3. | Paquetes en Java .....                               | 126 |
| 13.4. | Uso de clases de otros paquetes (importar) .....     | 127 |
| 13.5. | Visibilidad .....                                    | 130 |
| 13.6. | Ejercicio resuelto.....                              | 134 |
| 14.   | Polimorfismo por Sobrecarga .....                    | 143 |
| 14.1. | Definición .....                                     | 143 |
| 14.2. | Implementación en Java .....                         | 144 |
| 14.3. | Ejercicio resuelto.....                              | 147 |
| 15.   | Herencia .....                                       | 150 |
| 15.1. | Ejemplo introductorio .....                          | 150 |
| 15.2. | Concepto de herencia y representación.....           | 151 |
| 15.3. | Implementación en Java .....                         | 155 |
| 15.4. | Palabra reservada <i>super</i> .....                 | 158 |
| 15.5. | Uso de objetos con herencia .....                    | 159 |
| 15.6. | Ejercicio resuelto.....                              | 161 |
| 16.   | Polimorfismo por Sobrescritura.....                  | 167 |
| 16.1. | Definición .....                                     | 167 |
| 16.2. | Implementación en Java .....                         | 168 |
| 16.3. | Ejercicio resuelto.....                              | 171 |
| 17.   | Conversión de variables ( <i>Cast</i> ).....         | 177 |
| 17.1. | Problemática métodos de clases hijas .....           | 177 |
| 17.2. | <i>Cast</i> .....                                    | 179 |
| 17.3. | Operador <i>instanceof</i> .....                     | 180 |
| 17.4. | Ejercicio resuelto.....                              | 181 |
| 18.   | Atributos y Métodos de Clase ( <i>Static</i> ) ..... | 186 |
| 18.1. | Contexto.....  | 186 |
| 18.2. | Atributos y métodos de clase (estáticos).....        | 186 |
| 18.3. | Método <i>main</i> .....                             | 189 |
| 18.4. | Ejercicio resuelto.....                              | 189 |
| 19.   | Clases y Métodos Abstractos .....                    | 191 |
| 19.1. | Problemática instancias clase padre .....            | 191 |

|       |   |     |
|-------|---|-----|
| 19.2. | Clases abstractas y su representación ..... | 193 |
| 19.3. | Métodos abstractos y su representación..... | 194 |
| 19.4. | Implementación en Java.....                 | 194 |
| 19.5. | Métodos y clases final.....                 | 199 |
| 19.6. | Ejercicio resuelto.....                     | 200 |
| 20.   | Interfaces.....                             | 208 |
| 20.1. | Un nuevo reto.....                          | 208 |
| 20.2. | Definición .....                            | 208 |
| 20.3. | Implementación de una interfaz .....        | 209 |
| 20.4. | Representación.....                         | 213 |
| 20.5. | Uso de las interfaces .....                 | 214 |
| 20.6. | Ejercicio resuelto.....                     | 218 |
| 21.   | Ejemplo.....                                | 223 |
| 21.1. | Enunciado .....                             | 223 |
| 21.2. | Diagrama de clases .....                    | 224 |
| 21.3. | Código Java .....                           | 225 |
| 21.4. | Ejecución del programa .....                | 233 |
|       | Bibliografía.....                           | 236 |

## Introducción

La programación es un área de trabajo muy interesante, no solo por la alta demanda que tiene en la actualidad, sino también por su continua evolución y los constantes retos y satisfacciones que presenta a quienes trabajan en ella. La programación es mucho más que solo conocer un lenguaje técnico, pues incluye –entre otros aspectos– entender muy bien el problema que se desea resolver, elaborar modelos para representar la solución planteada y realizar pruebas para garantizar la adecuada funcionalidad del software que se desarrolle. Estos aspectos han probado ser valiosos en infinidad de campos del conocimiento, por lo que saber de programación es una habilidad valiosa para cualquier profesional, no solo para los ingenieros de sistemas o informáticos.

Existen diferentes paradigmas o estilos de programación, que tienen que ver con la forma en la cual se diseña y se implementa la solución a un problema, como por ejemplo diseñar una solución pensando principalmente en los datos y sus relaciones, o pensando en la secuencia de pasos o actividades que se deben ejecutar. Cada estilo de programación tiene sus ventajas y desventajas, y es importante conocer algunos de ellos para poder aplicarlos adecuadamente.

La programación orientada a objetos – POO – es un paradigma o estilo de programación en el cual el software se construye a partir de un conjunto de objetos que colaboran entre sí (como se detallará más adelante). Este estilo de programación ha sido uno de los más utilizados en los últimos años en el mundo del desarrollo de software, y muchos de los lenguajes de programación más populares, como Java, C#, Python, Ruby, entre otros, son orientados a objetos o por lo menos incluyen algunos aspectos de programación orientada a objetos.

En este libro se presentarán los principales conceptos de la programación orientada a objetos y su aplicación en el lenguaje de programación Java, para que así el lector pueda identificar las ventajas de este estilo de programación, y además pueda desarrollar una aplicación de software sencilla con este paradigma.

Para poder aprovechar mejor este contenido, se recomienda tener conocimientos básicos de lógica de programación, tales como tipos de datos, variables, expresiones y estructuras de control (condicionales y ciclos).

# 1. Objetos y Clases

En este capítulo se mostrarán las definiciones básicas de la programación orientada a objetos (POO): la definición de objeto y la de clase. También se mostrará la forma de representar las clases de manera gráfica, usando la notación estándar que existe para ello.

A finalizar este capítulo usted debe ser capaz de:

- Diferenciar entre objetos y clases en el contexto de la programación orientada a objetos.
- Identificar objetos y clases a partir de un enunciado dado.
- Representar una clase usando una notación gráfica estándar: UML.

## 1.1. Objetos

Un objeto es una unidad que tiene **características** propias, un **comportamiento** y una **identidad**. La identidad es aquello que hace único a cada objeto y lo diferencia de otros objetos similares.

Observe la siguiente imagen, ¿cuántos objetos se pueden identificar?



Imagen 1-1 Objetos

En la imagen se observan cuatro objetos, que son:

- El libro que se encuentra en la parte izquierda. Este libro tiene como **características**, entre otras, que se titula “Ingeniería de Software: Una perspectiva orientada a objetos”, que tiene como autor una persona de apellido Braude y tiene una marca de la dueña Sandra (internamente). Este libro puede ser ojeado y puede ser leído para aumentar los conocimientos de las personas. Esto conforma el **comportamiento** del libro. Aunque pueden existir muchos otros libros parecidos, la combinación de sus características lo hace único: ningún otro libro combina ese título, autor y dueño. Esto es lo que le da su **identidad**.



- El libro que se encuentra en la parte derecha de la imagen es otro objeto. Trate de identificar sus características y comportamiento. También trate de pensar en alguna característica de un libro que le permita darle identidad para diferenciarlo de otros.
- El carro de juguete que está en la parte inferior izquierda de la imagen. Algunas de las características de este carro son: que es de color naranja, que tiene dos puestos, dos puertas y cinco llantas (contando la de repuesto). Entre su comportamiento podemos ver que puede servir como adorno, que puede ser movido hacia adelante o hacia atrás y que al moverlo puede hacerlo a diferentes velocidades, tal vez para apostar carreras con otro carro de juguete. Pero surge la pregunta: ¿Cómo diferenciar este carro de otro con las mismas características?



*Imagen 1-2 Objetos Iguales*

Si todas las características de los objetos son iguales se puede usar su posición para diferenciar uno del otro. Por ejemplo, en una tienda usted puede decir: me vende por favor el carro naranja que está a la derecha. Esto permite diferenciarlo de otro que sea igual.

- El carro que está en la parte inferior derecha es el último objeto identificado. Es otro carro, pero con características diferentes, pues tiene colores rojo y blanco, tiene ocho puestos, tres puertas y cuatro llantas. Su comportamiento es similar al del primer carro, pero hay que reconocer que su velocidad es menor a la del carro naranja y por lo tanto perdería en una carrera.
- Un aspecto importante de los objetos es que pueden relacionarse entre sí para lograr un objetivo. Por ejemplo, se tiene la siguiente descripción: “El señor Pepe Pérez se acerca al cajero electrónico del banco MuchoDinero que hay en la esquina de la calle 51. Al llegar pasa su tarjeta débito y solicita el saldo de su cuenta de ahorros; obtiene un recibo y luego se retira. Posteriormente se acerca a este mismo cajero la señora Lola López, quien también utiliza su tarjeta y luego retira \$200.000 de su cuenta corriente”.



En la anterior descripción se pueden identificar los siguientes objetos:

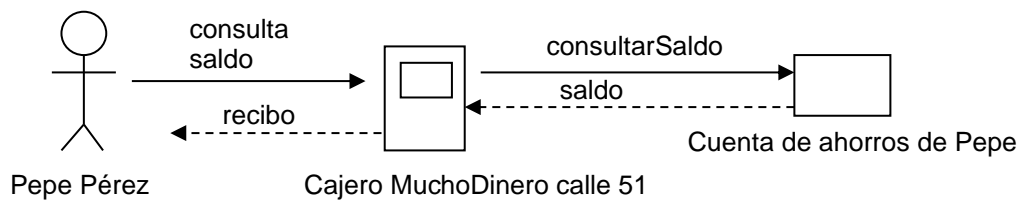
- El cajero electrónico de MuchoDinero que está en la calle 51
- El señor Pepe Pérez
- La señora Lola López
- La cuenta del señor Pepe Pérez
- La cuenta de la señora Lola López

Cada uno de estos objetos tiene características propias, un comportamiento y una identidad. Se puede decir, por ejemplo, que la cuenta del señor Pepe Pérez es una cuenta de ahorros, tiene un saldo y un número que está en la tarjeta (características). Además, se diferencia de la cuenta de la señora Lola López y de otras cuentas porque tiene un número único, lo que



le permite tener identidad. Esta cuenta presta servicios como: consultar su saldo, hacer retiros o consignaciones, lo cual es su comportamiento.

Además, estos objetos interactúan entre sí para lograr la funcionalidad que se desea.



*Imagen 1-3 Objetos interactuando para lograr una funcionalidad*

En la imagen puede verse que Pepe Pérez le envía un mensaje al cajero para indicar que desea consultar el saldo y el cajero a su vez le envía un mensaje a la cuenta para saber el saldo y luego mostrar esta información.

Es importante aclarar que, en el contexto de la POO, un objeto puede ser una cosa, pero también puede ser un animal, una persona, un objeto abstracto (como la cuenta de ahorros del ejemplo anterior) o una entidad virtual (como una página web). Lo importante es que para cada objeto se puedan definir sus características, comportamientos e identidad.

### **Ejercicio 1-1**

Determine cuáles de los siguientes elementos son objetos y cuáles no:

- El carro Mazda negro de placas NAF-234
- El número de goles que hizo la selección Colombia en el último partido
- El lapicero azul de Paola
- El color rojo de la bandera de Colombia que está en el estadio Palograndre

### **Ejercicio 1-2**

Identifique los objetos que se presentan en el siguiente enunciado, definiendo para cada uno sus características y comportamiento. Trate de definir también para cada objeto qué es lo que lo diferencia de los demás (su identidad).

En Manizales se celebró el año pasado el 33 Festival Internacional de Teatro, que contó con la participación de diferentes agrupaciones, para entretenimiento de propios y visitantes. Dentro de las obras más alabadas se encuentra "Utsushi", que es una obra de la agrupación japonesa "Sankai Juku" que representa sentimientos profundos de las personas usando solo la expresión corporal. El grupo Sankai Juku, que fue creado en 1975, está dirigido por el bailarín y coreógrafo Ushio Amagatsu. Su obra, de gran renombre internacional, fue seleccionada para ser el cierre del Festival.

Aunque se presentaron varias obras de grupos asiáticos, también se encuentran otras de países latinoamericanos, como la obra "Maluco" del grupo "La Cuarta Colectivo Artístico" de Uruguay, que es una adaptación de una novela histórica pero reinterpretada con imaginación y humor. El grupo La Cuarta fue creado hace 13 años por Laura Pouso y Gustavo Zidan, y sus producciones han tenido gran impacto regional e internacional.

## 1.2. Clases

Cuando se modela un sistema usando orientación a objetos no es necesario detallar todos los objetos que hacen parte de este, sino que se identifican los diferentes grupos de objetos similares que hay. Los objetos se clasifican buscando los que representen lo mismo y compartan características y comportamiento. Esto se conoce como una clase.

Es decir, una clase es un **tipo o grupo de objetos** donde se definen las características y comportamientos que tienen en común.

Una clase también se define como el **patrón, molde o plantilla con el cual se crean objetos de un mismo tipo** para usarlos en un programa. No es posible tener objetos si no se ha definido previamente la clase para poder crear estos objetos.

En el caso de la imagen presentada en la sección anterior se identifican dos clases:



- La clase Libro, que define que los libros tienen como características: título, autor y dueño. Los libros pueden ser ojeados y leídos por las personas.
- La clase Carro de Juguete. Esta clase define que los carros tendrán color, número de puestos, número de puertas y número de llantas. También define que los carros podrán ser movidos hacia adelante y hacia atrás, o usados como adornos.

Por lo general las clases aparecen como sustantivos de los cuales se habla en alguna descripción. En el ejemplo de Pepe y Lola en el cajero, las clases son:

- Persona (o Cliente del banco)
- Cuenta bancaria
- Cajero electrónico

Para que Pepe Pérez pueda tener una cuenta de ahorros es necesario que primero el banco haya definido las características y los comportamientos que tendrán las cuentas. Es decir, **primero se define la clase y luego se crean los objetos** de esa clase.

Cada vez que se crea un objeto lo que se hace es darles **valores concretos** a las características, y así es posible tener objetos diferentes a partir de una misma clase.

Por ejemplo, se puede decir que todas las cuentas tienen como características: un número, un saldo y un tipo (si es de ahorro o corriente). Además, las cuentas permiten consultar el saldo, retirar y consignar. Cada objeto Cuenta tiene un valor diferente para estas características. Por ejemplo, el número de la cuenta de Pepe Pérez es 1234-56, tiene un saldo de \$150.000 y es de ahorros, mientras que la cuenta de Lola López es 1235-87, tiene un saldo de \$300.000 y es una cuenta corriente. En ambas cuentas se puede consultar el saldo, consignar y retirar, aunque cada una teniendo en cuenta sus valores. Por ejemplo,

Pepe solo puede retirar máximo \$150.000 mientras que Lola puede retirar más dinero de su cuenta.

|                   |                   |
|-------------------|-------------------|
| Cuenta de Pepe    | Cuenta de Lola    |
| número = 1234-56  | número = 1235-87  |
| saldo = \$150.000 | saldo = \$300.000 |
| tipo = ahorros    | tipo = corriente  |

### Ejercicio 1-3

Enumere las clases que identifica en el siguiente enunciado (que es el mismo de la sección anterior):

En Manizales se celebró el año pasado el 33 Festival Internacional de Teatro, que contó con la participación de diferentes agrupaciones, para entretención de propios y visitantes. Dentro de las obras más alabadas se encuentra “Utsushi”, que es una obra de la agrupación japonesa “Sankai Juku” que representa sentimientos profundos de las personas usando solo la expresión corporal. El grupo Sankai Juku, que fue creado en 1975, está dirigido por el bailarín y coreógrafo Ushio Amagatsu. Su obra, de gran renombre internacional, fue seleccionada para ser el cierre del Festival.

Aunque se presentaron varias obras de grupos asiáticos, también se encuentran otras de países Latinoamericanos, como la obra “Maluco” del grupo “La Cuarta Colectivo Artístico” de Uruguay, que es una adaptación de una novela histórica pero reinterpretada con imaginación y humor. El grupo La Cuarta fue creado hace 13 años por Laura Pouso y Gustavo Zidan, y sus producciones han tenido gran impacto regional e internacional.

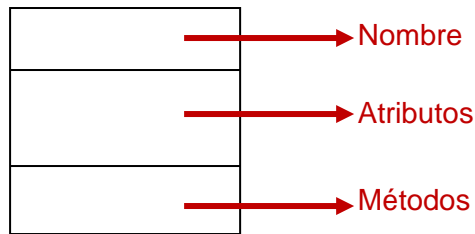
## 1.3. Terminología

En la programación orientados a objetos, los términos que se usan para definir los elementos que constituyen los objetos y las clases son:

- Los objetos son conocidos como **instancias** de las clases.
- A las características se les llama **atributos** (en inglés a veces se pueden encontrar como “*fields*” o campos). También se conocen como variables de instancia.
- Los comportamientos se definen a través de un conjunto de **métodos** (también se conocen como operaciones).

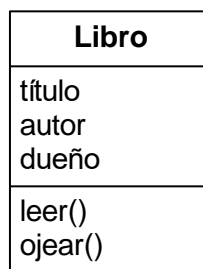
## 1.4. Representación

La forma de representar una clase de manera gráfica es mediante un rectángulo dividido en tres partes. En la parte superior va el nombre de la clase, en el recuadro del medio van los atributos y en la parte inferior van los métodos.

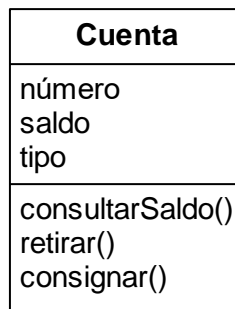


Esta notación se ha convertido en un estándar que conocen y aplican todos los desarrolladores de software, y que está definida por un lenguaje de modelado llamado **UML** (*Unified Modeling Language*).

Por ejemplo, para los carros de juguete el diagrama es:



El diagrama para una cuenta bancaria es:



## 1.5. Implementación en Java

Para definir una clase en Java se utiliza la siguiente sintaxis:

```
/**
 * Descripción de la clase
 * @author nombreAutor
 * @version númeroVersión
 */
public class NombreClase
{
}

```

Se tienen los siguientes elementos en esta definición:

- Descripción (comentario): Aquí se explica en qué consiste la clase. También se define el autor y el número de versión.
  - Para especificar el autor se usa `@author`, seguido del nombre.
  - Para especificar la versión se usa `@version`, seguido del número de versión, por ejemplo 1.0, 1.7, 2.5, etc.
- Las palabras reservadas `public class`: La palabra `public` es un modificador para indicar que la clase es pública, es decir, puede ser usada por otras clases; y la palabra `class` marca el inicio de la definición de la clase.
- El nombre de la clase. La primera letra debe ir en mayúscula.
- El bloque de código de la clase: Todo el código de la clase, es decir, los atributos y los métodos, estarán encerrados en un bloque de código que está enmarcado por llaves.

Por ejemplo, el código para definir la clase Libro (todavía sin sus atributos y métodos), es:

```
/**
 * Representa un libro de la biblioteca de una persona, de manera que
 * la persona pueda saber los datos básicos de cada libro que posee.
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Libro
{
}
```

Nota: La forma de implementar los atributos y los métodos de las clases en Java se presentará en los siguientes capítulos.

En otro ejemplo, la estructura básica para la clase Cuenta es:

```
/**
 * Una cuenta bancaria, es decir, un registro en
 * una entidad bancaria que tiene un saldo (cantidad de dinero)
 * que se puede modificar mediante retiros o consignaciones.
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Cuenta
{
}
```

### **Ejercicio 1-4**

- Elabore el diagrama de clases, indicando cuáles serían los atributos y los métodos para:
  - Un jugador de fútbol
  - Un teléfono celular
  - Una película
- Escriba la definición de las clases anteriores en Java.

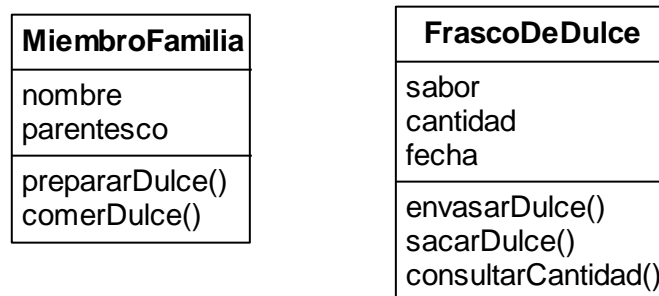
## 1.6. Ejercicio resuelto

La abuela Pata prepara deliciosos dulces en su cocina, y los nietos Hugo, Paco y Luis están siempre muy pendientes porque saben que cuando están recién hechos son deliciosos. La abuelita hace dulces de mora, guayaba y otros deliciosos sabores, pero hoy ha preparado un dulce de brevas que le ha quedado muy rico. Ella envasa el dulce en frascos de 200 gramos. En total le salen 5 frascos que va marcando con la fecha a medida que los envasa. En cuanto se va de la cocina los tres nietos entran y consumen un poco de algunos de los frascos antes de que se dé cuenta. Pero la abuelita no es tonta, y al regresar revisa la cantidad de cada frasco y los que tienen menos de 200 gramos los aparta como regalo para sus nietos, pues sabe que les encantan esos dulces. Los otros frascos los guarda para darlos a otros miembros de la familia Pata.

En el enunciado anterior se identifican los siguientes objetos:

- La abuela Pata
- El nieto Hugo
- El nieto Paco
- El nieto Luis
- El primer frasco de dulce
- El segundo frasco de dulce
- El tercer frasco de dulce
- El cuarto frasco de dulce
- El quinto frasco de dulce

Las clases correspondientes son dos, una para un miembro de la familia Pato y otra para los frascos de dulce. Cada frasco tiene como atributos su sabor, la cantidad (que al comienzo es 200 gramos) y la fecha. Uno de los métodos que debe prestar cada frasco de dulce es “envasar dulce” para que la abuela pueda guardar el dulce que elabora. Otro método importante es “sacar dulce”, para que todos puedan disfrutar los dulces de la abuela. Además, para que la abuela pueda saber que pasó, cada frasco de dulce debe tener un método que permita consultar su cantidad. El diagrama de clases correspondiente es:



El código en Java para las clases (sin atributos y métodos, por el momento), es:

```
/**
 * Persona que hace parte de una familia, por ejemplo,
 * de la familia del Pato Donald.
 * @author Juan David Correa
 * @version 0.5
 */
public class MiembroFamilia
{
}
```

```
/**
 * Un frasco con delicioso dulce, elaborado por la abuela pata.
 * Por ejemplo, un frasco de dulce de guayaba.
 * @author Juan David Correa
 * @version 0.5
 */
public class FrascoDeDulce
{
}
```

## 2. Introducción a Java

**Juan David Correa Granada**

En este capítulo se hace una breve presentación del lenguaje de programación Java, incluyendo la forma de definir variables de tipos primitivos y realizar operaciones básicas con ellas.

Al finalizar este capítulo usted debe ser capaz de:

- Definir variables de tipos primitivos en Java.
- Escribir expresiones matemáticas sencillas en Java.

### 2.1. Contexto

Java es un lenguaje de programación orientado a objetos, desarrollado por *Sun Microsystems*, y que hoy en día propiedad de *Oracle*. Su primera versión apareció en 1995, y se popularizó inicialmente como un lenguaje para internet, que podía correr en muchas plataformas.

Sin embargo, posteriores versiones de este lenguaje lo han convertido en un lenguaje de programación para aplicaciones empresariales, gracias a muchas de sus ventajas (dadas principalmente por la orientación a objetos) y el conjunto de librerías que ofrecen diferentes servicios.

### 2.2. Máquina virtual de Java

Los programas en Java no son compilados directamente al lenguaje de máquina, sino a un lenguaje intermedio llamado “bytecode”. Para poder ejecutar estos programas es necesario tener instalada la máquina virtual de java (MVJ o JVM – *Java Virtual Machine*). Existen máquinas virtuales para muchos sistemas operativos, y esto es lo que permite que los programas en Java sean portables.

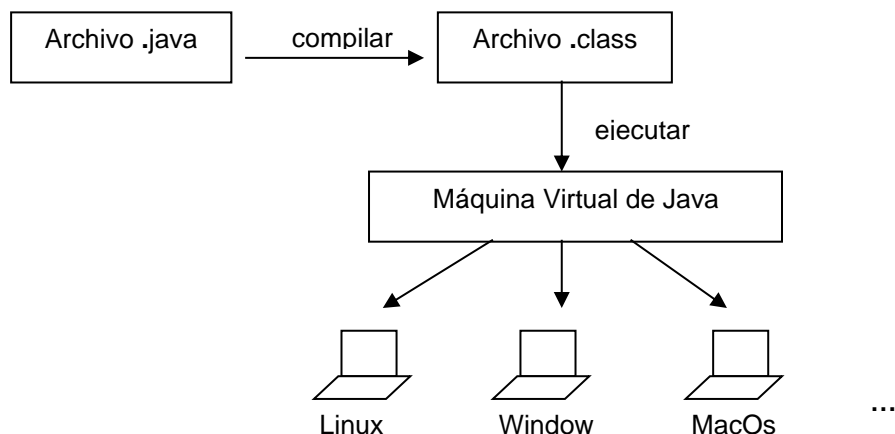


Imagen 2-1 Portabilidad gracias a la máquina virtual de Java



La máquina virtual, además de “convertir” las instrucciones que están en *bytecode* a las instrucciones acordes a cada sistema operativo, lleva a cabo otras funciones, como:

- Aislar los programas para que no puedan tener acceso directo al hardware del equipo.
- Ejecutar funciones adicionales que facilitan la programación, por ejemplo, eliminar los elementos que han dejado de ser referenciados.
- Comprobación de algunos errores al momento de ejecución, antes de ejecutar estas operaciones en el sistema operativo.

Una de las desventajas que presenta este esquema es que los programas Java pueden ser un poco más lentos que los programas que se ejecutan directamente sobre el sistema operativo.

## 2.3. Ambientes de desarrollo

Aunque es posible desarrollar aplicaciones Java utilizando sólo las herramientas básicas, generalmente el desarrollo se hace utilizando ambientes integrados de desarrollo (IDE), que incrementan la productividad del programador.

Algunos IDE (entre una gran cantidad que existen actualmente):

- Eclipse (<http://www.eclipse.org/downloads/index.php>)
- NetBeans (<http://www.netbeans.info/downloads/index.php>)
- JBuilder (<http://www.borland.com/jbuilder/>)
- JDeveloper (<http://www.oracle.com/technology/products/jdev/index.html>)

## 2.4. Estructura de un programa sencillo

Todo programa Java está formado por clases, de las cuales por lo general hay una que se “ejecuta” (para iniciar el programa), y las demás conforman toda la estructura con la funcionalidad deseada.

Las clases que se pueden ejecutar son aquellas que tiene el método **main**. Un programa muy sencillo, por lo tanto, está formado por una clase que tiene un método *main*, con el código que se desea ejecutar.

Para organizar las instrucciones de un programa, éstas se enmarcan en **bloques de código**. Un bloque de código está delimitado por llaves: { }, la llave que abre al comienzo del bloque, y la llave que cierra, al final. Para enmarcar todo lo que está en la clase o todo lo que está en un método se usan bloques de código con las llaves.

El “esqueleto” para un programa muy simple es:

```

/**
 * Comentario de la clase
 */
public class NombreClase
{
    /**
     * Comentario del método
     */
    public static void main(String[] args)
    {
        // Aquí va el código del método
    }
}

```

## 2.5. Comentarios

Para insertar comentarios en un programa se utiliza:

```
// Para comentarios de una línea
```

```

/*
    Para comentarios de varias líneas
*/

```

```

/**
    Para comentarios de varias líneas, que deseen ser incluidos en
    la documentación que se genera automáticamente
*/

```

## 2.6. Identificadores

Los identificadores son los nombres que se colocan a las clases, los métodos y las variables, entre otros. Estos nombres deben ser significativos, es decir, que al leerlos se entienda lo que significan. Por ejemplo, un nombre significativo es: “colorPantalla” y no “clrPtila”, o “numeroPersonas” y no “num”.

Para los identificadores se deben seguir las siguientes recomendaciones:

- Puede contener letras, números o el símbolo de subrayado, pero sólo pueden comenzar con una letra o el símbolo de subrayado, no con números.
- No deben contener espacios en blanco.
- Java es sensible a mayúsculas y minúsculas, es decir una variable llamada “primera” será diferente a otra llamada “Primera”.
- No pueden ser palabras propias del lenguaje de programación –o palabras reservadas– como “class” o “for”.

También hay unas convenciones para los identificadores en los programas en Java, entre las cuales se tienen las siguientes:

- Los nombres de las clases empiezan en mayúscula. Las demás letras son minúsculas, excepto si es el comienzo de otra palabra. Ejemplo: Casa, TeatroPrincipal
- Los nombres de los métodos, los atributos y las variables empiezan en minúscula. Las demás letras son minúsculas, excepto si es el comienzo de otra palabra. Ejemplo: contador, cuentaVeces
- Las constantes deben ir en mayúscula sostenida, y cada palabra se separa de la anterior por el símbolo de subrayado Ejemplo: IVA, VALOR\_TOTAL

### **Ejercicio 2-1**

Para cada uno de los siguientes nombres de variables, indique si es válido y si sigue las convenciones:

- 4oPuesto
- incremento%
- estatura promedio
- valorNeto
- CapitalPais

## **2.7. Tipos de datos**

En Java se habla de dos grandes grupos de tipos de datos, **los primitivos y los referenciados**. Los tipos de datos primitivos se refieren a datos que son comunes a la mayoría de los sistemas, y que se manejan como *valores*, y son los que se van a tratar en esta parte. Los tipos referenciados agrupan las clases, interfaces y los arreglos, y tienen como características que pueden ser definidos por los programadores y que no se manejan como valores sino como objetos, pero esto se estudiará en detalle más adelante.

Los tipos de datos primitivos, o básicos, en Java, son:

| Nombre  | Descripción  | Rango (o valores)   |
|---------|--|---|
| boolean | Valores lógicos o valores de verdad  | true<br>false   |
| byte    | Números enteros (valores pequeños)   | -128 a 127  |
| short   | Números enteros (valores medianos)   | -32.768 a 32.767  |
| int     | Números enteros  | $-2^{31}$ a $2^{31}-1$  |
| long    | Números enteros (valores grandes)  | $-2^{63}$ a $2^{63}-1$  |
| char    | Caracteres (aunque Java también lo considera numérico). Cada carácter se representa entre comillas simples, por ejemplo: 'f', '%', '8' | Utiliza el formato UNICODE para la representación de cada carácter ( <a href="http://www.unicode.org/">http://www.unicode.org/</a> ). |
| float   | Para números con decimales, con precisión simple (de 6 a 7 dígitos decimales significativos)   | aproximadamente $\pm 3.40282347 \times 10^{38}$   |
| double  | Para números con decimales, con precisión doble (15 dígitos decimales significativos)  | aproximadamente $\pm 1.79769313486231570 \times 10^{308}$   |

Existe otro tipo de dato muy utilizado, para las cadenas de caracteres, que es **String**. Aunque este es un tipo referenciado, se utiliza de manera similar a los tipos de datos básicos.

Las cadenas de caracteres (*String*) se representan con comillas dobles, por ejemplo:  
"Esta es una cadena"  
"texto"  
"7652"

### **Ejercicio 2-2**

Determine el tipo de dato de Java más adecuado para:

- La edad de una persona
- La nota final de un estudiante en una asignatura
- La marca de un vehículo
- El valor del dólar, en pesos

## **2.8. Variables**

Es la unidad básica de almacenamiento en Java. Toda variable debe ser de un tipo definido, y debe ser declarada antes de ser utilizada. La sintaxis para declarar las variables es:

```
tipo nombreVariable;
```

También se le puede asignar un valor inicial:

```
tipo nombreVariable = valorInicial;
```

Ejemplos:

```
// Variable llamada valor, de tipo entero, sin un valor inicial:  
int valor;  
  
// Variable precio, de tipo double, con valor inicial 35.6  
double precio = 35.6;
```

Se pueden declarar varias variables del mismo tipo en la misma línea, separadas por comas. Por ejemplo:

```
/* Declara dos variables de tipo boolean:  
   una llamada resultadoUno, sin valor inicial,  
   y otra llamada resultadoDos, con valor inicial true.  
*/  
boolean resultadoUno, resultadoDos=true;
```

Las variables sólo son conocidas dentro del bloque de código en el cual son definidas, lo cual se denomina alcance de la variable.

Ejemplo:

```

/**
 * Ejemplo de alcance de variables
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Alcance
{
    public static void main (String args [])
    {
        int valorUno = 10;
        if (valorUno == 10)
        {
            int valorDos = 20;
            System.out.print (valorUno + " " + valorDos);
        }
        valorDos = 100; // error, aquí no se conoce esta variable
    }
}

```

## 2.9. Constantes

El valor asignado a las constantes no puede ser modificado en el transcurso del programa.

Para declarar constantes se utiliza la misma sintaxis que para las variables, pero adicionando al comienzo la palabra reservada *final*, ejemplo:

```
final double IVA = 15.3;
```

### Ejercicio 2-3

Escriba cómo declarar, en Java, valor de la raíz cuadrada del número 2 como una constante (el valor es 1.41421356237309).

## 2.10. Expresiones y operadores

Una expresión es una combinación de operadores y operandos, que dan como resultado un valor determinado. Los operadores en Java pueden ser variables, valores literales, constantes, llamadas a métodos u otras expresiones. Ejemplos de expresiones:

```

(temperatura / 2) + 3
(edad > 18) & (edad < 60)

```

A continuación, se presentan los principales operadores que se pueden usar en Java para escribir expresiones.

- **Asignación:**

Para asignar un valor a una variable se usa el operando =, por ejemplo:

```
int cantidadAlumnos;  
cantidadAlumnos = 34;
```

- **Aritméticos:**

Cuando se realiza una expresión que incluye operaciones aritméticas el resultado será del tipo de dato con mayor precisión involucrado en la expresión. Por ejemplo, si todos los operandos son enteros el resultado será entero (*¡incluso en una división!*), pero si algún operando es un número real con decimales (como un *double*) el resultado será también un número con decimales.

Operaciones básicas entre dos operandos:

|   |                                 |
|---|---------------------------------|
| + | Suma                            |
| - | Resta                           |
| * | Multiplicación                  |
| / | División                        |
| % | Módulo (residuo de la división) |

Por ejemplo:

```
int numeroImpar = 6 - 1;  
int valorEntero = numeroImpar / 2;           // En valorEntero queda 2  
double valorDecimal = numeroImpar / 2.0;     // En valorDecimal queda 2.5  
int resultado = valorEntero * 10 + (7 % 3);
```

Estas operaciones se pueden combinar con la asignación si uno de los operandos es el mismo donde se desea guardar el resultado. Es decir, si se tiene una operación de la forma  $a = a + b$ , se puede cambiar por:  $a += b$ . Ejemplo:

```
double gramos = 34.5;  
gramos*=3;      // Equivale a gramos = gramos * 3;
```

Operaciones que aplican sobre un operando:

|    |  |
|----|--|
| ++ | Incrementa en uno el valor de la variable. |
| -- | Decrementa en uno el valor de la variable. |

Ejemplo:

```
short peso = 60;  
peso++;           // peso queda con valor 61
```

### **Ejercicio 2-4**

Escriba la siguiente expresión matemática como expresión válida de Java:

$$\frac{7.5y - (12.3 + w)}{6w}$$

- **Relacionales:**

Comparan dos operandos, y el resultado es un valor booleano.

> Mayor que  
 < Menor que  
 == Igual  
 != Diferente  
 >= Mayor o igual  
 <= Menor o igual

Ejemplo:

```
int edad = 20;
boolean mayorEdad = edad > 18;           //true
boolean igualEdad = edad == 21;          // false
```

- **Lógicos:**

Permiten realizar operaciones entre operandos de tipo booleano.

! Es la negación (es un operador unitario)  
 & AND  
 && AND cortocircuito (si el primer operador es false no evalúa el segundo)  
 | OR  
 || OR cortocircuito (si el primer operador es true no evalúa el segundo)  
 ^ OR excluyente

Ejemplos:

```
boolean ganoCarrera = false;
boolean juegaLimpio = true;
boolean recibePremio = ganoCarrera | juegaLimpio;           //true
boolean doblePremio = ganoCarrera && juegaLimpio;            //false
```

### **Ejercicio 2-5**

Escriba la siguiente expresión lógica usando sintaxis de Java:  
 “Iremos de viaje si no estoy enfermo y si sales a vacaciones”.

## **2.11. Expresiones válidas**

Para que una expresión sea válida debe combinar operadores y operandos que sean compatibles y siguiendo la sintaxis de los operadores. Por ejemplo, si se desea usar el

operador suma, deben tenerse dos operandos, como por ejemplo en 5+3. Si falta uno de los dos sería una expresión no válida:

```
int suma = 8 + ;           // expresión no válida: falta un operador
```

Tampoco es válido asignar valores a variables incompatibles. Por ejemplo:

```
boolean pudoCalcular = 2 * 6;           // tipos no compatibles:  
                                         // la expresión es int y la variable es boolean
```

### **Ejercicio 2-6**

Defina si la siguiente expresión es válida o no. Justifique su respuesta:

```
int resultado = 8.5 - 1.5;
```

## **2.12. Ejercicio resuelto**

Se desea calcular el valor de una venta, que se calcula multiplicando el número de productos que lleva la persona, por el valor del producto. El cliente ha llevado 3 productos, y el producto tiene un valor de \$20.000.

El programa en Java para hacer esto se muestra a continuación.

```
/**  
 * Determina el valor de una venta, basado en la cantidad de productos  
 * y el valor del producto.  
 * @author Sandra V. Hurtado  
 * @version 1.0  
 */  
public class Venta  
{  
    public static void main (String args[])  
    {  
        int cantidadProductos = 3;  
        int valorProducto = 20000;  
        double valorVenta = cantidadProductos * valorProducto;  
    }  
}
```



### 3. Atributos

En este capítulo se extiende la definición de atributo y se muestra su representación gráfica en UML. También se explica la forma de implementar los atributos en el lenguaje de programación Java.

Al finalizar este capítulo usted debe ser capaz de:

- Identificar atributos de las clases en un enunciado dado.
- Representar los atributos de una clase usando una notación gráfica estándar.
- Implementar una clase en Java incluyendo sus atributos.

#### 3.1. Definición

En el capítulo anterior se estableció que los objetos tienen características, comportamiento e identidad. Los objetos se modelan utilizando clases que se constituyen en los elementos base de la programación orientada a objetos.

Un atributo (o variable de instancia) es la definición de una característica de los objetos de una clase. Es decir, **un atributo representa una característica** que tendrán los objetos. Por ejemplo, algunos atributos de una clase carro pueden ser el color, la placa, la marca, etc.

Para definir un atributo en una clase son indispensables dos elementos:

- El nombre: Es la denominación dada a cada atributo, normalmente es un sustantivo. Debe ser un nombre significativo.
- El tipo: Cada atributo debe ser de un tipo de dato definido. Estos tipos de datos dependen del lenguaje de programación seleccionado. Por ejemplo, en Java se tienen tipos de datos como *int* (número entero), *double* (número real), *boolean* (valor de verdad o booleano), *String* (cadena de caracteres), entre otros.

#### 3.2. Modelado de atributos

Retomando el ejemplo del capítulo anterior: “El señor Pepe Pérez se acerca al cajero electrónico del banco MuchoDinero que hay en la esquina de la calle 51. Al llegar pasa su tarjeta débito y solicita el saldo de su cuenta de ahorros; obtiene un recibo y luego se retira. Posteriormente se acerca a este mismo cajero la señora Lola López, quien también utiliza su tarjeta y luego retira \$200.000 de su cuenta corriente”. Se identificaron 3 clases:

- Persona (o Cliente del Banco)
- Cuenta bancaria
- Cajero electrónico

Para establecer los atributos de las clases se tienen en cuenta todas las fuentes de información disponibles en el proyecto, ya sea un enunciado, entrevista, documento, diagrama, cuestionario u otro que describa lo que el sistema debe permitir. El sentido común y la experiencia también son factores para tener en cuenta a la hora de establecer los atributos de una clase.

Al definir la clase Persona, por ejemplo, es posible identificar muchas características: *número de identificación*, *nombres*, *apellidos*, *edad*, *sexo*, *grupo sanguíneo*, *etnia* y *número*

de hijos. Aunque existen muchas otras características que podrían ser enunciadas, al modelar una clase el analista o desarrollador debe identificar aquellas que son **relevantes** para el **contexto** en el que se construye el sistema. Estas características relevantes serán los atributos de la clase.

Para el enunciado anterior el contexto está relacionado con el retiro de dinero de un banco en medios electrónicos, de allí que los atributos *identificación*, *nombres* y *apellidos* sean muy importantes, mientras que la *edad*, *grupo sanguíneo*, *etnia* y *número de hijos* no tengan la misma relevancia. Estas últimas serían muy importantes en otro contexto, por ejemplo, para un sistema médico.

Una vez identificados los atributos para modelar, se establece el tipo de dato más adecuado. Por ejemplo, para los atributos de Persona:

| Nombre del atributo | Tipo de dato (en Java)     |
|---------------------|----------------------------|
| Identificación      | <i>String</i> o <i>int</i> |
| Nombres             | <i>String</i>              |
| Apellidos           | <i>String</i>              |

### **Ejercicio 3-1**

A partir del siguiente enunciado, identifique los atributos y sus respectivos tipos de datos, para las clases CarroRecolector y Aseador:

La empresa de Aseo “Mundo Limpio” tiene varios carros de recolección de basura y también varios aseadores que se encargan de la limpieza de la ciudad. Los carros de recolección de basura pueden ser de diferentes tipos: están los carros generales, de 15 toneladas, donde se recoge la basura de los hogares; los carros industriales, de 20 toneladas, donde se recoge la basura de las industrias, y los carros especiales, de 5 toneladas, donde se recolectan los residuos peligrosos (como los residuos hospitalarios o los electrónicos).

Don Pepe, uno de los aseadores de la empresa, nos cuenta cómo últimamente se ha incrementado el uso de los carros especiales:

*“Sí, señores, imagínese que debe ser por la cultura de las personas, por toda la publicidad sobre la importancia de conservar el planeta y esas cosas, que ahora hay cada vez más empresas que nos solicitan el servicio de recolección de residuos peligrosos. Quién iba a pensar, por ejemplo, que en un colegio lo necesitaran, pero resulta que sí, porque en los laboratorios de química y hasta los mismos señores del aseo manejan este tipo de residuos. Entonces la empresa ha comprado 5 carros más para este tipo de recolección, y nos parece muy bien porque así tenemos una ciudad más limpia y nosotros también más trabajito.”*

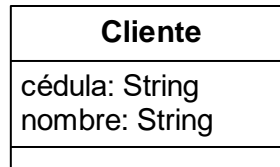
Don Pepe lleva 5 años trabajando en la empresa y ha recibido varias capacitaciones para manejar los residuos peligrosos, de lo cual se siente muy orgulloso. Los directivos de la compañía nos confirman esto, y nos dicen también que cada carro de recolección tiene todas las medidas de seguridad y sanitarias requeridas. La vida útil promedio de estos carros es de 10 años, y los que tengan más de este tiempo de uso son vendidos como chatarra.

### 3.3. Representación

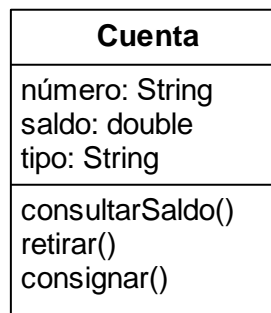
Recordemos que las clases se representan con un rectángulo dividido en tres partes. La parte intermedia es el lugar dónde se definen los atributos. La forma de representar cada atributo en el diagrama es:

```
nombre: tipo
```

Por ejemplo, para un cliente del banco, el diagrama (mostrando solo los atributos) queda:



Para una cuenta bancaria el diagrama es:



De acuerdo con el diagrama anterior, cada objeto que se cree de la clase Cuenta tendrá los siguientes atributos, a los que podrá darle valores:

- El número, que es de tipo *String* (puede tener caracteres como el guion).
- El saldo, que es de tipo *double*.
- El tipo, que es de tipo *String* (para indicar si es corriente o de ahorros).

### 3.4. Implementación en Java

Los atributos se definen como variables, es decir:

```
tipoDeDato nombre;
```

Se recomienda que el nombre del atributo esté en minúscula, con mayúscula intermedia cuando está formado por varias palabras. Los atributos deben ir **DENTRO** del bloque de código de la clase.

Por ejemplo, el código de la clase Cliente, con sus atributos es:

```

/**
 * Una persona que tiene una cuenta en el banco, y por lo tanto
 * se considera cliente del banco.
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Cliente
{
    String cedula;
    String nombre;
}

```

También se muestra el código de la clase Cuenta, incluyendo sus atributos:

```

/**
 * Una cuenta bancaria, es decir, un registro en
 * una entidad bancaria que tiene un saldo (cantidad de dinero)
 * que se puede modificar mediante retiros o consignaciones.
 * @author Sandra V. Hurtado
 * @version 1.2
 */
public class Cuenta
{
    String numero;
    double saldo;
    String tipo;
}

```

### **Ejercicio 3-2**

Escriba el código Java para crear las clases Revista (en un almacén) y Paciente (en un consultorio médico), que se muestran a continuación:

| Revista   | Paciente  |
|---|---|
| título: String<br>páginas: int<br>precio: double<br>enOferta: boolean | nombre: String<br>edad: int<br>sexo: char<br>peso: double |

## **3.5. Ejercicio resuelto**

Del ejercicio del capítulo anterior (el de la abuela Pata que hace deliciosos dulces), se identificaron dos clases: MiembroFamilia y FrascoDeDulce. Al establecer el tipo de dato adecuado para cada atributo, el diagrama queda:

| <b>MiembroFamilia</b>                |
|--------------------------------------|
| nombre: String<br>parentesco: String |
| prepararDulce()<br>comerDulce()      |

| <b>FrascoDeDulce</b>                                  |
|---|
| sabor: String<br>cantidad: int<br>fecha: String       |
| envasarDulce()<br>sacarDulce()<br>consultarCantidad() |

El código en Java para estas clases – sin tener en cuenta los métodos – sería:

```
/**
 * Persona que hace parte de una familia, por ejemplo,
 * de la familia del Pato Donald.
 * @author Juan David Correa
 * @version 1.0
 */
public class MiembroFamilia
{
    String nombre;
    String parentesco;
}
```

```
/**
 * Un frasco con delicioso dulce, elaborado por la abuela pata.
 * Por ejemplo, un frasco de dulce de guayaba.
 * @author Juan David Correa
 * @version 1.0
 */
public class FrascoDeDulce
{
    String sabor;
    int cantidad;
    String fecha;
}
```

## 4. Instrucciones Básicas en Java: Condicionales, Lectura y Escritura

En este capítulo se continúa con la introducción al lenguaje de programación Java, presentando las estructuras condicionales y unas instrucciones básicas que permiten solicitar y mostrar datos al usuario, para permitir construir programas un poco más elaborados.

Al finalizar este capítulo usted debe ser capaz de:

- Escribir un programa sencillo en Java, donde se puedan pedir y mostrar datos al usuario (enteros, reales y cadenas de caracteres).
- Describir tres instrucciones condicionales en Java: *if*, *if-else* y *switch*.
- Incluir estructuras condicionales en un método en Java.

### 4.1. Instrucciones de escritura


Haciendo uso de expresiones es posible escribir programas en Java que realicen una serie de operaciones para llegar a un resultado. Sin embargo, después de hacer la operación es importante **mostrar** el resultado, pues si no se muestra ninguna información el usuario no podrá saber si el programa funcionó o no. Para esto se usan las instrucciones de escritura o de salida.

En Java existen varias formas de escribir o mostrar datos de salida al usuario, y en esta sección se mostrarán dos de ellas: una para mostrar datos por consola (solo texto) y otra para mostrar datos en pequeña ventanas llamadas ventanas de diálogo.

- **Escritura por consola**

Para mostrar algún mensaje por pantalla, la instrucción que se utiliza en Java es:

```
System.out.println (MENSAJE);
```



MENSAJE es un *String* (puede ser una variable o una cadena de texto entre comillas).

Por ejemplo, se desea mostrar por pantalla el mensaje “¡Un saludo!”, la instrucción es:

```

/**
 * Programa para mostrar un saludo en pantalla - por consola
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class SaludoConsola
{
    public static void main(String args[])
    {
        System.out.println("¡Un saludo!");
    }
}

```

El resultado de ejecutar este programa es:

```
¡Un saludo!
```

Si se desea mostrar el resultado de una variable de un tipo diferente a *String* se puede usar el operador de concatenación de cadenas, que es el **+** (como el de suma, pero para cadenas es unir una cadena con otra).

Por ejemplo, retomando un ejercicio de un capítulo anterior: un programa para calcular y mostrar el valor de una venta, conociendo la cantidad de productos (que es 3) y el valor del producto (que es \$20.000).

```

/**
 * Determina el valor de una venta, basado en la cantidad de productos
 * y el valor del producto.
 * @author Sandra V. Hurtado
 * @version 1.5
 */
public class Venta
{
    public static void main (String args [])
    {
        int cantidadProductos = 3;
        int valorProducto = 20000;
        double valorVenta = cantidadProductos * valorProducto;
        System.out.println("Valor de la venta: " + valorVenta);
    }
}

```

El resultado que se muestra es:

```
Valor de la venta: 60000.0
```

- **Escritura por ventana de diálogo**

De manera similar hay otro método para mostrar un mensaje, pero usando una ventana de diálogo. Este método es:

```
JOptionPane.showMessageDialog(ventanaMayor, MENSAJE);
```

Como por el momento no se tienen más ventanas el primer valor será *null*.

Sin embargo, para poder usar este método, se debe escribir, **antes de la clase**, la siguiente instrucción:

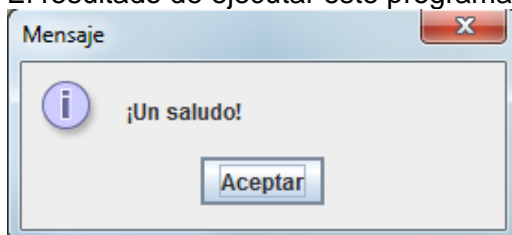
```
import javax.swing.JOptionPane;
```

Por ejemplo, un código en Java para mostrar el mensaje “¡Un saludo!” en una ventana de diálogo, es:

```
import javax.swing.JOptionPane;

/**
 * Programa para mostrar un saludo en una ventana de diálogo
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class SaludoDialogo
{
    public static void main (String args [])
    {
        JOptionPane.showMessageDialog(null, "¡Un saludo!");
    }
}
```

El resultado de ejecutar este programa es:



### **Ejercicio 4-1**

Escriba un programa en Java que calcule y muestre el área de un triángulo rectángulo, conociendo que su base es de 7.5 centímetros y la altura es de 6 centímetros.

## **4.2. Instrucciones condicionales**

Muchas veces durante un programa es importante verificar una condición para determinar si se ejecutan o no un conjunto de instrucciones. Por ejemplo: “Mostrar un mensaje de bienvenida si la persona es mayor de edad, o un mensaje de prohibido si es menor de edad”. Para esto se debe verificar si la edad de la persona es igual o mayor a 18 años, y dependiendo del resultado se ejecutará la instrucción correspondiente.



Este tipo de instrucciones, que permiten **analizar una condición** para saber cuáles acciones ejecutar, se denominan estructuras de decisión o condicionales.

A continuación, se presentan brevemente tres instrucciones de decisión en el lenguaje de programación Java: selección simple (*if*), selección doble (*if-else*) y selección múltiple (*switch*).

### 4.3. Selección simple (*if*)

Esta estructura permite tomar una decisión para saber si se ejecutan un conjunto de instrucciones o no. La sintaxis es:

```
if (condición)
{
    // instrucciones
}
```

La condición puede ser una variable *boolean* o una expresión que dé como resultado un valor booleano.

Por ejemplo, calcular el valor de una venta, dada la cantidad de productos y considerando que el valor del producto es \$20.000. Se aplica un descuento del 10 %, pero solo si el valor de la venta es superior a \$50.000.

```
/**
 * Determina el valor de una venta, basado en la cantidad de productos,
 * el valor del producto y un posible descuento.
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class VentaConDescuento
{
    /**
     * Calcula el valor de una venta, a partir de la cantidad de productos.
     * El valor de un producto es 20.000,
     * y se hace un descuento del 10 % si el valor es mayor a 50.000
     * @param cantidadProductos la cantidad de productos vendidos
     * @return el valor de la venta, en pesos
     */
    double calcularValorVenta (int cantidadProductos)
    {
        final double DESCUENTO = 0.10;
        int valorProducto = 20000;
        double valorVenta = cantidadProductos * valorProducto;
```

//continúa

```

        if (valorVenta > 50000)
        {
            valorVenta = valorVenta - (valorVenta * DESCUENTO);
        }
        return valorVenta;
    }
} // fin clase VentaConDescuento

```

Para probar esta función es necesario escribir otra clase con el método *main*. Se presenta un código simplificado para esta otra clase, solo como ilustración, pues ese tema se tratará en otros capítulos.

```

/**
 * Clase para usar la funcionalidad del valor de la venta
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class PruebaValorVenta
{
    public static void main (String[] args)
    {
        VentaConDescuento venta = new VentaConDescuento();
        double valorObtenido = venta.calcularValorVenta(4);
        System.out.println("Valor de la venta: " + valorObtenido);
    }
}

```

### **Ejercicio 4-2**

Complete el siguiente fragmento de código, para que diga que los menores de 12 años y los mayores de 70 años deben ingresar con una persona responsable. Tenga en cuenta que la edad de la persona está en una variable llamada “edad”:

```

if (  )
{
    System.out.println("Debe ingresar con un responsable");
}

```

## **4.4. Selección doble (*if-else*)**

Esta estructura permite tomar una decisión para seleccionar entre dos conjuntos de instrucciones, es decir, saber si se ejecutan un conjunto de instrucciones u otro. Si se cumple la condición se ejecutan las instrucciones dentro del *if*, y si no se cumple la condición se ejecutan las instrucciones dentro del *else*.

La sintaxis es:

```

if (condición)
{
    // Instrucciones si la condición se cumple
}
else
{
    // Instrucciones si la condición no se cumple
}

```

Note que las instrucciones dentro del *if* y del *else* van enmarcadas por llaves: {}

Por ejemplo, calcular el valor de una venta, dada la cantidad de productos vendidos y considerando que el valor del producto es \$20.000. Se aplica un descuento del 10 %, si el valor de la venta es superior a 50.000, y en caso contrario el descuento es del 5 %.

```

/**
 * Determina el valor de una venta, basado en la cantidad de productos,
 * el valor del producto y descuentos por promociones.
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class VentaConPromocion
{
    /**
     * Calcula el valor de una venta, a partir de la cantidad de productos.
     * El valor de un producto es 20.000.
     * Se hace un descuento del 10 % si el valor es mayor a 50.000,
     * y un descuento del 5 % si el valor es igual o inferior a 50.000
     * @param cantidadProductos la cantidad de productos vendidos
     * @return el valor de la venta, en pesos
     */
    double calcularValorVenta (int cantidadProductos)
    {
        final double DESCUENTO_MAYOR = 0.10;
        final double DESCUENTO_MENOR = 0.05;
        int valorProducto = 20000;
        double valorVenta = cantidadProductos * valorProducto;
        if (valorVenta > 50000)
        {
            valorVenta = valorVenta - (valorVenta * DESCUENTO_MAYOR);
        }
        else
        {
            valorVenta = valorVenta - (valorVenta * DESCUENTO_MENOR);
        }
        return valorVenta;
    }
}

```

### Ejercicio 4-3

Determine cuál mensaje se muestra por pantalla si se ejecutan las siguientes instrucciones:

```
int velocidad = 60;
if (velocidad > 60 | velocidad < 20)
{
    System.out.println("Velocidad no permitida");
}
else
{
    System.out.println("Velocidad adecuada");
}
```

## 4.5. Selección múltiple (*switch*)

Esta estructura permite ejecutar un conjunto de instrucciones dependiendo del valor de una variable.

En este caso no se trabaja con condiciones, sino con valores específicos de una variable. Los tipos de variables que pueden usarse en el *switch* son: *byte*, *short*, *int*, *char* y *String*.

La sintaxis es:

```
switch (variable)
{
    case valorUno :
        // instrucciones
        break;
    case valorDos :
        // instrucciones
        break;
    default : // instrucciones
}
```

Dependiendo del valor de la variable el programa "entra" por una de las opciones.

Si no coincide con ningún valor en la lista "entra" a las instrucciones que están en *default*. Este valor es opcional.

Por ejemplo, calcular el valor de una venta, dada la cantidad de productos comprados y el tipo de cliente. El valor del producto es \$20.000 y los descuentos dependen del tipo de cliente, así: para el tipo 1 no hay descuento, para el tipo 2 es un 5 %, para el 3 el descuento es 8 % y para los demás tipos es el 12 %.

```

/**
 * Determina el valor de una venta, basado en la cantidad de productos,
 * el valor del producto y descuentos por tipo de cliente.
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class VentaPorTipoCliente
{
    /**
     * Calcula el valor de una venta, a partir de la cantidad de productos
     * y del tipo de cliente. El valor de un producto es 20.000.
     * Para clientes tipo 1 no hay descuento, para tipo 2 el descuento
     * es 5 %, para tipo 3 es 8 % y para los demás es 12 %.
     * @param cantidadProductos la cantidad de productos vendidos
     * @param tipoCliente el tipo de cliente determina el descuento,
     * va de 1 a 5.
     * @return el valor de la venta, en pesos
     */
    double calcularValorVenta (int cantidadProductos, int tipoCliente)
    {
        final double DESCUENTO_TIPO2 = 0.05;
        final double DESCUENTO_TIPO3 = 0.08;
        final double DESCUENTO_OTROS = 0.12;
        int valorProducto = 20000;
        double valorVenta = cantidadProductos * valorProducto;
        switch (tipoCliente)
        {
            case 1:
                break;
            case 2:
                valorVenta = valorVenta - (valorVenta * DESCUENTO_TIPO2);
                break;
            case 3:
                valorVenta = valorVenta - (valorVenta * DESCUENTO_TIPO3);
                break;
            default:
                valorVenta = valorVenta - (valorVenta * DESCUENTO_OTROS);
        }
        return valorVenta;
    }
}

```

**Observe que en cada uno de los casos las instrucciones terminan con break. Si no se escribe esta instrucción se continuarían ejecutando todas las demás instrucciones que hay en el switch.**

#### **Ejercicio 4-4**

En un jardín infantil el cobro de la matrícula depende de la edad del niño, pues esto es lo que determina en cuál área se asigna y qué materiales utilizará. Los niños de 1 año pagan un valor de \$400.000, los de 2 y 3 años pagan \$500.000 y los de 4 años pagan \$600.000.

Sin embargo, si el niño tiene un hermano que también estudia en el jardín se hace un descuento del 15 %.

Dado el anterior enunciado, escriba las instrucciones Java para determinar el valor que debe pagar un niño por su matrícula, dada su edad y la información de si tiene o no un hermano en el jardín.

## 4.6. Instrucciones de lectura

Los programas que se escriben en cualquier lenguaje de programación pueden ser mucho más flexibles si trabajan con información que ingrese el usuario. Esto permite, por ejemplo, que un programa que suma dos números no tenga que ser cambiado cada vez que el usuario desee sumar dos valores diferentes, sino que cada vez que se ejecuta lea los dos números que el usuario desea sumar y muestre el resultado.

En Java existen varias formas de leer o solicitar datos de entrada al usuario, haciendo uso de clases y métodos especializados. En esta sección se mostrarán dos formas para leer datos, uno por consola y otro mediante ventanas de diálogo.

- **Lectura por consola**

Una de las formas de recibir datos del usuario es usando la clase *Scanner*. Los pasos para usar esta clase son:

1. Se debe colocar, antes de la clase, la instrucción:

```
import java.util.Scanner;
```

2. En el código se escribe, antes de hacer cualquier lectura:

```
Scanner lector = new Scanner(System.in);
```

3. Cuando se desee leer un dato se usan los siguientes métodos del objeto lector, y cada uno se recibe en una variable del tipo correspondiente:
  - *nextLine()* para leer un *String*
  - *nextInt()* para leer un valor *int*
  - *nextDouble()* para leer un valor *double*

Antes de usar estos métodos es importante mostrar un mensaje al usuario para explicarle qué información es la que debe ingresar.  
Por ejemplo:

```
System.out.print("Precio del producto: ");  
double precio = lector.nextDouble();
```

A continuación, se muestra un código en Java para mostrar si una persona puede participar en una votación o no, dependiendo de su edad. Se tiene en cuenta que una persona puede votar en Colombia a partir de los 18 años.

```

import java.util.Scanner;

/**
 * Determina si una persona puede votar o no, dependiendo de su edad
 * (En Colombia la edad para poder votar son 18 años).
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class EdadVotarConsola
{
    public static void main (String args [])
    {
        Scanner lector = new Scanner(System.in);

        final int EDAD_VOTAR = 18;

        System.out.print("Escriba su edad: ");
        int edad = lector.nextInt();
        if (edad >= EDAD_VOTAR)
        {
            System.out.println("Sí puede votar");
        }
        else
        {
            System.out.println("No puede votar");
        }
    }
}

```

Un ejemplo de ejecución de este código, donde aparece en verde lo que escribe el usuario:

```

Escriba su edad: 60
Sí puede votar

```

Otro ejemplo:

```

Escriba su edad: 12
No puede votar

```

- **Lectura con ventana de diálogo**

Otra de las formas de solicitar datos al usuario es usando ventanas de diálogo, para lo cual los pasos son:

1. Se debe colocar, antes de la clase, la instrucción:

```
import javax.swing.JOptionPane;
```

2. Usar la siguiente instrucción cuando se desee leer un dato:

```
JOptionPane.showInputDialog(MENSAJE)
```

Mensaje que se muestra al usuario para explicarle qué dato debe ingresar.

Se debe asignar el valor que retorna a una variable de tipo *String*, por ejemplo:

```
String nombreUsuario = JOptionPane.showInputDialog("Nombre:");
```

3. Como el método anterior solo permite pedir una cadena (*String*), se debe usar otro método adicional para realizar la conversión en caso de necesitar leer un *int* o un *double*:
- `Integer.parseInt(CADENA)` para convertir una cadena en un *int*
  - `Double.parseDouble(CADENA)` para convertir una cadena en un *double*

Por ejemplo:

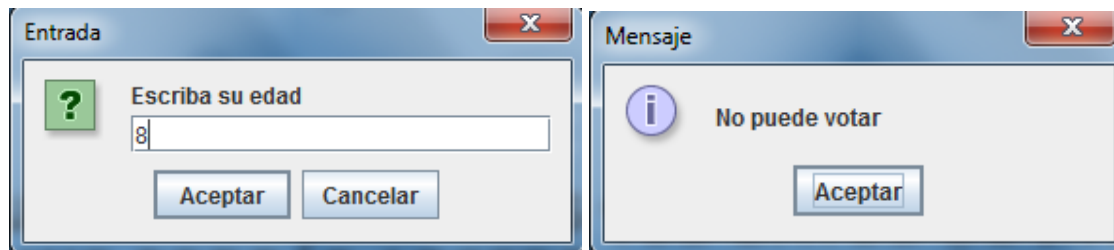
```
String valorCadena = JOptionPane.showInputDialog("Valor producto:");  
double precio = Double.parseDouble(valorCadena);
```

De nuevo se mostrará el código en Java para pedir la edad de una persona y decir si puede votar o no, pero usando la ventana de diálogo para pedir los datos de entrada:

```
import javax.swing.JOptionPane;  
  
/**  
 * Determina si una persona puede votar o no, dependiendo de su edad.  
 * (En Colombia la edad para poder votar son 18 años).  
 * Muestra la información y solicita los datos por ventanas de diálogo  
 * @author Sandra V. Hurtado  
 * @version 1.0  
 */  
public class EdadVotarDialogo  
{  
    public static void main (String args [])  
    {  
        String cadenaEdad = JOptionPane.showInputDialog(  
            "Escriba su edad");  
  
        final int EDAD_VOTAR = 18;  
  
        int edad = Integer.parseInt(cadenaEdad);  
  
        if (edad >= EDAD_VOTAR)  
        {  
            JOptionPane.showMessageDialog(null, "Sí puede votar");  
        }  
        else  
        {  
            JOptionPane.showMessageDialog(null, "No puede votar");  
        }  
    }  
}
```

Un ejemplo al ejecutar este código:





En caso de querer solicitar un *char*, tanto por consola como con ventanas de diálogo, se debe leer una cadena (*String*), y luego obtener el primer carácter de la cadena, con la instrucción:

```
CADENA.charAt(0)
```

Por ejemplo:

```
String texto = "verdadero";  
char primerCaracter = texto.charAt(0);
```

En este caso, en la variable “primerCaracter” queda la letra v.

#### **Ejercicio 4-5**

Modifique la clase Venta, mostrada en este capítulo, para que **la cantidad de productos** vendidos y **el valor del producto** se pidan al usuario como entradas, y que al final se muestre el valor de la venta. Realice la lectura de datos primero por consola y luego mediante ventanas de diálogo.

### **4.7. Ejercicio resuelto**

Una persona llega a una entidad financiera para enviar una cantidad de dinero a otra persona. Para poder hacer esto le preguntan si desea realizar una consignación, una transferencia o un giro. Si realiza una consignación cobran el 2 % de la cantidad que desea enviar, si realiza una transferencia cobran el 4 % y si realiza un giro cobran el 3 %. El programa debe solicitar la cantidad de dinero que se desea enviar y la forma de envío, y debe mostrar cuál es el valor que se le cobra a la persona para hacer este envío.

Usando instrucciones de lectura y escritura por consola:

```

import java.util.Scanner;

/**
 * Cálculo del cobro a una persona para enviar una cantidad de dinero,
 * dependiendo de la forma de envío.
 * Consignación (c) es 2%, transferencia (t) es 4% y giro (g) es 3%
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class EnvioDineroConsola
{
    public static void main(String[] args)
    {
        Scanner lector = new Scanner(System.in);
        System.out.print("Cantidad de dinero que desea enviar: ");
        double cantidadDinero = lector.nextDouble();
        lector.nextLine(); //Para "limpiar" el lector
        System.out.print(
            "Forma de envío: consignación, transferencia o giro ");
        String cadenaForma = lector.nextLine();
        char formaEnvio = cadenaForma.charAt(0);

        double valorCobro = 0;

        switch (formaEnvio)
        {
            case 'c':    valorCobro = cantidadDinero * 0.02;
                        break;
            case 't':    valorCobro = cantidadDinero * 0.04;
                        break;
            case 'g':    valorCobro = cantidadDinero * 0.03;
                        break;
        }

        System.out.println("El valor que cobrará es: "+ valorCobro);

        lector.close(); //Para "cerrar" apropiadamente la consola
    }
}

```

Usando ventanas de diálogo:

```

import javax.swing.JOptionPane;

/**
 * Cálculo del cobro a una persona para enviar una cantidad de dinero,
 * dependiendo de la forma de envío.
 * Consignación (c) es 2%, transferencia (t) es 4% y giro (g) es 3%
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class EnvioDineroDialogo
{
    public static void main(String[] args)
    {
        String cadenaCantidad = JOptionPane.showInputDialog(
            "Cantidad de dinero que desea enviar ");
        double cantidadDinero = Double.parseDouble(cadenaCantidad);

        String cadenaForma = JOptionPane.showInputDialog(
            "Forma de envío: consignación, transferencia o giro ");
        char formaEnvio = cadenaForma.charAt(0);

        double valorCobro = 0;

        switch (formaEnvio)
        {
            case 'c':    valorCobro = cantidadDinero * 0.02;
                        break;
            case 't':    valorCobro = cantidadDinero * 0.04;
                        break;
            case 'g':    valorCobro = cantidadDinero * 0.03;
                        break;
        }

        JOptionPane.showMessageDialog(null,
            "El valor que cobrará es: "+ valorCobro);
    }
}

```

## 5. Métodos

En este capítulo se extiende la definición de método y se muestra su representación gráfica en UML y la forma de implementarlo en el lenguaje de programación Java.

Al finalizar este capítulo usted debe ser capaz de:

- Representar los métodos de una clase usando una notación gráfica estándar.
- Implementar una clase sencilla en Java incluyendo sus métodos.

### 5.1. Definición

Como se ha explicado previamente, los objetos, además de tener características e identidad, tienen comportamiento. El comportamiento se refiere a las acciones que puede hacer un objeto, o las acciones que se pueden hacer con él, incluyendo responder “preguntas” sobre sus características.

Cuando se modela un grupo de objetos mediante una clase, el comportamiento común que se desee representar de estos objetos se establece en los métodos. Por ejemplo, en un carro (automóvil) sus métodos son: encender, acelerar, frenar, cambiar una llanta, venderlo, etc.

Un método, por lo tanto, representa **una acción, comportamiento o servicio de los objetos**. También puede entenderse un método como una pequeña función o procedimiento que actúa sobre el objeto y sus atributos.

Es importante tener claro que un método no es un programa completo, solo una parte pequeña de un programa, y que está relacionado con el objeto sobre el cual se ejecuta. Un programa completo está formado por varios objetos que hacen uso de sus métodos para cumplir alguna funcionalidad.

Para definir un método, los elementos son:

- El nombre: Por lo general es un verbo que explica la acción que se puede realizar.
- Los parámetros: Corresponden a información adicional que se envía al objeto para que pueda realizar la acción. No siempre es necesario enviar información adicional al objeto para usar un método, depende de lo que se desee pedir. Por ejemplo, si a una persona se le pregunta el nombre, no es necesario darle información adicional, porque cada persona tiene como atributo su nombre. Pero si a una persona se le pide que pague una factura, se le debe dar la factura y el dinero, porque estos no son atributos de la persona.  
Un método puede tener cero, uno o más parámetros.
- El tipo de retorno: Corresponde al tipo de información que se espera que devuelva (o “retorne”) un objeto cuando se llame al método. No siempre los métodos retornan algo, por ejemplo, si a una persona se le pregunta el nombre, se espera que retorne una cadena de caracteres con el nombre, pero si a una persona se le pide que entre a una oficina solo se espera que realice la acción y no que retorne ninguna información.

Un método solo puede tener **un** valor de retorno (o ninguno). No es posible que un método retorne varias cosas. Si se necesitan varios resultados, se debe dividir el método y hacer un método diferente para cada resultado.

## 5.2. Modelado de métodos

Para establecer los métodos de las clases se tienen en cuenta todas las fuentes de información disponibles en el proyecto para determinar qué acciones realiza cada objeto o qué acciones se pueden realizar con ese objeto en el programa. Los métodos por lo general se relacionan con los atributos que tiene cada objeto, permitiendo consultarlos, modificarlos o hacer operaciones con ellos.

Es importante tener en cuenta que en la programación orientada a objetos cada método no debe realizar muchas acciones, solo debe realizar una función concreta.

Al igual que con los atributos, solo se deben establecer los métodos que sean **relevantes** para el sistema que se esté modelando. Por ejemplo, en un almacén donde vendan carros, no son importantes los métodos acelerar o frenar del carro; sino otros métodos como definir el precio del carro, venderlo o asignarle la placa (si es nuevo).

En un ejemplo previo: “El señor Pepe Pérez se acerca al cajero electrónico del banco MuchoDinero que hay en la esquina de la calle 51. Al llegar pasa su tarjeta débito y *solicita el saldo de su cuenta* de ahorros; obtiene un recibo y luego se retira. Posteriormente se acerca a este mismo cajero la señora Lola López, quien también utiliza su tarjeta y luego *retira \$200.000 de su cuenta* corriente”.

Se han dejado en cursiva las acciones que se realizan con cada cuenta, que son: consultar saldo y retirar dinero. Por lo tanto, estas dos acciones o servicios se identifican como métodos. También, considerando el contexto, que es una entidad bancaria, se definirá el método *consignar*.

Para cada método se definirá si necesita que le envíen información adicional para poder realizar su función (los parámetros) y si devolverá o retornará algún valor, y de qué tipo sería este valor. Por ejemplo, para la cuenta bancaria:

| Nombre del Método | Parámetros y su tipo de dato (en Java)                      | Lo que retorna y su tipo de dato (en Java)                |
|-------------------|---|---|
| Consultar saldo   | No requiere (pues el saldo es un atributo de la clase).     | El saldo: <i>double</i>                                   |
| Retirar           | La cantidad de dinero que se desea retirar: <i>double</i> . | Una indicación de si se pudo retirar o no: <i>boolean</i> |
| Consignar         | La cantidad de dinero que se consignará: <i>double</i> .    | No retorna nada.  |

### Ejercicio 5-1

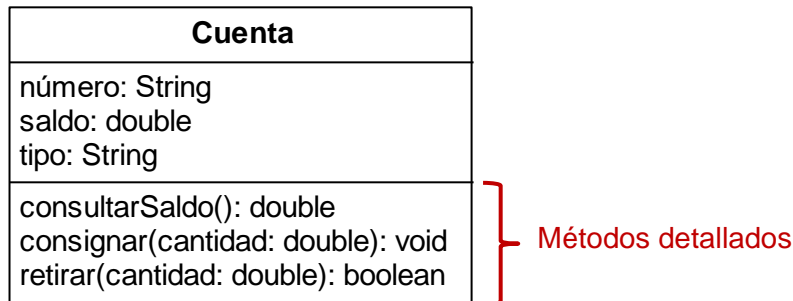
A partir del siguiente enunciado, identifique los métodos – con sus parámetros y retorno – que tendría la clase Colectivo.

En la ciudad de Lejanías tienen un servicio de transporte muy exclusivo, que funciona solo con colectivos. Cada colectivo tiene capacidad para 8 personas únicamente, y durante todo el recorrido se suben y bajan personas. Por supuesto, al final del recorrido se deben bajar todas las personas que vayan en el colectivo. Se debe tener en cuenta que no es posible que se suban más personas que la capacidad dada por el colectivo. En algunos puntos del recorrido se tienen puntos de control donde un supervisor verifica la cantidad de personas que tiene el colectivo, garantizando así que se cumplan con las normas establecidas.

### 5.3. Representación

En el tercer compartimiento del rectángulo que representa a una clase van los métodos, es decir, los servicios que tendrán los objetos de esa clase.

Por ejemplo, para la clase Cuenta, el diagrama es:



La forma de representar cada método es:

```
nombre (parámetros): retorno
```

Un **parámetro** es una variable que contiene información adicional (diferente a la que ya tiene el objeto), que necesita recibir el método para poder cumplir con el servicio que le están pidiendo. Un parámetro es un valor de “entrada” para el método.

Cada parámetro tiene un nombre y un tipo de dato, así:

```
nombre: tipo
```

Por ejemplo, en la clase Cuenta los parámetros de cada método son:

- consultarSaldo():double → no tiene parámetros. Para una cuenta poder consultar su saldo no necesita nada, pues esa información ya la tiene cada objeto internamente como atributo y por eso no recibe parámetros. Se colocan los paréntesis sin nada dentro.
- retirar(cantidad:double):boolean → tiene el parámetro “cantidad”. Para poder retirar de una cuenta se debe saber cuánta cantidad de dinero se desea retirar. Esta cantidad puede ser un número con decimales, por lo que se selecciona el tipo de dato *double*.
- consignar(cantidad:double):void → tiene el parámetro “cantidad”. Para poder consignar se necesita saber qué cantidad de dinero se desea consignar. Esa cantidad es un valor de tipo *double*.

Además, dependiendo del servicio que se le pida al objeto, él **retornará** algún valor o no. Por ejemplo, cuando usted dice “por favor abran la ventana”, no espera que le den nada, solo que se realice la acción. Por otra parte, cuando dice “por favor me presta una regla”, usted espera que le den algo (una regla).

Con los métodos sucede algo similar: todos realizan acciones, pero algunos retornarán algo y otros no retornarán nada. En Java, si el método retorna algún valor se debe definir de qué tipo es, y si no retorna nada se escribe **void**.

En el caso de la clase Cuenta:

- consultarSaldo():**double** → Se espera que indique cuál es el saldo, y por lo tanto hay un valor de retorno. Como el saldo es de tipo *double*, el tipo de retorno de este método también es *double*.
- retirar(cantidad:double):**boolean** → Cada cuenta realiza el retiro de dinero internamente, pero algunas veces no podrá hacerlo porque no hay saldo suficiente. Por ese motivo se espera saber si se pudo realizar el retiro o no. Es decir, además de cambiar internamente el saldo del objeto cuenta, se espera un valor de verdad como respuesta (true o false), lo cual es el tipo *boolean*.
- consignar(cantidad:double):**void** → No se espera que retorne nada. Simplemente se desea que se consigne el valor en la cuenta, es decir, que se aumente su saldo, pero esto es algo que se hará internamente, no hay que esperar respuesta. En este caso se define como tipo de retorno "void".

### **Ejercicio 5-2**

- a) Defina para los siguientes métodos si necesitan parámetros o no, y de qué tipo serían:
- El método "ladrar" de un perro
  - El método "sumar" de una calculadora
  - El método "cocinar" de un horno microondas
  - El método "encender" de un computador
- b) Defina para los siguientes métodos si retornan algo o no, y de qué tipo sería lo que retorna:
- El método "sumar" de una calculadora
  - El método "encender" de un computador
  - El método "cambiarTamaño" de un cuadrado
  - El método "consultarNombre" de una persona

## **5.4. Implementación en Java**

Después de tener identificados todos los métodos en el diagrama de clases, se puede elaborar el código correspondiente en un lenguaje de programación como Java.

Los métodos en el lenguaje Java tienen dos partes:

- Un encabezado: tiene la información general del método, que es igual a la que está en el diagrama, solo que escrito con la notación del lenguaje de programación.
- Un cuerpo: es donde se escribe el código, que se ejecutará cada vez que llamen al método. Es decir, es el "corazón" del método, donde se define su comportamiento, las acciones que realiza con el objeto y sus atributos.

A continuación, se presentará la forma de definir los métodos de una clase usando el lenguaje de programación Java.

- **El encabezado**

El encabezado del método es donde se indica su nombre, los parámetros que recibe y el tipo de dato que retorna. La forma de definirlo es:

```
/**
 * Descripción del método
 * @param nombreParámetro explicación
 * @return explicación
 */
tipoRetorno nombreMétodo(parámetros)
{
}
```

- Lo primero que debe ir es el comentario que explica en qué consiste el método, los parámetros que recibe y lo que retorna.
  - Para cada parámetro se usa un *@param*, seguido del nombre del parámetro y luego una explicación del mismo.
  - Para lo que retorna se usa *@return*, seguido de una explicación de lo que retorna.
- Luego va la línea de encabezado del método, también llamada la interfaz o la firma del método, así:
  - Primero va el tipo de retorno (como *int*, *double* o *void*).
  - Luego sigue el nombre del método, con la primera letra en minúscula.
  - Luego van los parámetros entre paréntesis.
    - Si no recibe parámetros se colocan los paréntesis vacíos.
    - Si van varios parámetros se separan por comas.
    - Para cada parámetro va primero el tipo de dato y luego el nombre.

Para el ejemplo:

- a) El encabezado del método `consultarSaldo` de la clase `Cuenta`, que aparece en el diagrama como “`consultarSaldo(): double`”, es:

```
/**
 * Consulta el saldo actual de la cuenta (el dinero que tiene).
 * @return el valor del saldo, en pesos.
 */
double consultarSaldo()
```

- b) El encabezado del método “`retirar(cantidad:double): boolean`” es:

```
/**
 * Retira o saca una cantidad de dinero de la cuenta,
 * si hay saldo suficiente.
 * @param cantidad la cantidad que se desea retirar, en pesos.
 * @return si se pudo hacer el retiro porque tenía dinero
 * suficiente o no (true o false)
 */
boolean retirar(double cantidad)
```



c) El encabezado del método “consignar(cantidad:double): void” es:

```
/**
 * Consigna o adiciona una cantidad de dinero en la cuenta,
 * lo cual incrementa el saldo.
 * @param cantidad    la cantidad de dinero que se desea
 *                    consignar, en pesos
 */
void consignar(double cantidad)
```

- **El cuerpo**


El cuerpo de cada método tiene las instrucciones Java para hacer las acciones que se requieren. El cuerpo de los métodos siempre va enmarcado o encerrado por llaves; esto permite diferenciar los diferentes métodos que tenga una clase.

Por ejemplo, para el método “consignar”, lo que se hace es sumar la cantidad que se recibe al saldo de la cuenta, así:

```
/**
 * Consigna o adiciona una cantidad de dinero en la cuenta,
 * lo cual incrementa el saldo.
 * @param cantidad    la cantidad de dinero que se desea
 *                    consignar, en pesos
 */
void consignar(double cantidad)
{
    saldo = saldo + cantidad;
}
```

Cuando un método retorna algún valor, la instrucción para hacerlo es:

```
return loQueRetorna;
```



Puede ser un valor, una variable o una expresión, que corresponda al tipo de dato que se indicó como retorno del método.

Por ejemplo, para el método “retirar”, debe mirar si el saldo alcanza para retirar la cantidad solicitada. Si es así se resta del saldo y se retorna un valor verdadero (*true*), y si no alcanza el saldo se retorna un valor falso (*false*).

```

/**
 * Retira o saca una cantidad de dinero de la cuenta,
 * si hay saldo suficiente.
 * @param cantidad la cantidad que se desea retirar, en pesos.
 * @return si se pudo hacer el retiro porque tenía dinero
 *         suficiente o no (true o false)
 */
boolean retirar(double cantidad)
{
    if (saldo >= cantidad)
    {
        saldo = saldo - cantidad;
        return true;
    }
    else
    {
        return false;
    }
}

```

Es importante tener en cuenta:

- Si un método tiene en su encabezado algún tipo de retorno, siempre debe tener una instrucción *return*.
- La instrucción *return* termina la ejecución del método, incluso si está dentro de un ciclo.

Por ejemplo, para el método “consultarSaldo” no es necesario hacer ninguna operación, solo se debe retornar el saldo actual, por lo tanto, el método queda:

```

/**
 * Consulta el saldo actual de la cuenta (el dinero que tiene).
 * @return el valor del saldo, en pesos.
 */
double consultarSaldo()
{
    return saldo;
}

```

A continuación, se presenta el código completo de la clase Cuenta, como referencia:

```

/**
 * Una cuenta bancaria, es decir, un registro en
 * una entidad bancaria que tiene un saldo (cantidad de dinero)
 * que se puede modificar mediante retiros o consignaciones.
 * @author Sandra V. Hurtado
 * @version 1.5
 */
public class Cuenta
{
    String numero;
    double saldo;
    String tipo;

    /**
     * Consulta el saldo actual de la cuenta (el dinero que tiene).
     * @return el valor del saldo, en pesos.
     */
    double consultarSaldo()
    {
        return saldo;
    }

    /**
     * Retira o saca una cantidad de dinero de la cuenta,
     * si hay saldo suficiente.
     * @param cantidad la cantidad que se desea retirar, en pesos.
     * @return si se pudo hacer el retiro porque tenía dinero
     * suficiente o no (true o false)
     */
    boolean retirar(double cantidad)
    {
        if (saldo >= cantidad)
        {
            saldo = saldo - cantidad;
            return true;
        }
        else
        {
            return false;
        }
    }

    /**
     * Consigna o adiciona una cantidad de dinero en la cuenta,
     * lo cual incrementa el saldo.
     * @param cantidad la cantidad de dinero que se desea
     * consignar, en pesos
     */
    void consignar(double cantidad)
    {
        saldo = saldo + cantidad;
    }
} // fin clase Cuenta

```

### Ejercicio 5-3

Escriba el código Java para las clases que se presentan en el siguiente diagrama, incluyendo el código de los métodos. Tenga en cuenta que el IMC (índice de masa corporal) es el peso dividido el cuadrado de la estatura (peso / estatura<sup>2</sup>).

| Rectángulo  |
|---|
| base: double<br>altura: double                        |
| calcularÁrea(): double<br>calcularPerímetro(): double |

| Báscula   |
|---|
| calcularIMC(peso: double, estatura: double): double |

## 5.5. Ejercicio resuelto

La abuela Pata prepara deliciosos dulces en su cocina, de diferentes sabores. Ella envasa cada dulce en frascos de 200 gramos, que marca con la fecha para garantizar siempre su frescura. Su familia disfruta mucho estos dulces y por supuesto cuando pueden sacan dulce de los frascos para disfrutarlos.

El diagrama de clases, sin detallar los métodos, es:

| FrascoDeDulce   |
|---|
| sabor: String<br>cantidad: int<br>fecha: String       |
| envasarDulce()<br>sacarDulce()<br>consultarCantidad() |

Se analiza cada método para determinar si necesita o no parámetros y si retorna o no algún valor.

- envasarDulce:
  - Parámetros: Necesita saber la cantidad de dulce que se desea envasar y también la fecha en la cual se envasó, por lo tanto, necesita dos parámetros. El primer parámetro se llamará "cantidadEnvasar" y será un valor entero (*int*), que representa los gramos de dulce que se desean envasar en el frasco. El segundo parámetro se llamará "fechaEnvasado" y será una cadena de texto (*String*).
  - Retorno: Como los frascos tienen un límite de 200 gramos este método devolverá una indicación de si se pudo envasar la cantidad deseada o no (en caso de ser superior a 200 gramos). Esto corresponde a un valor de verdad (*true* o *false*), que es de tipo *boolean*.
- sacarDulce:
  - Parámetros: Necesita saber la cantidad de dulce que se desea sacar, para así disminuir la cantidad que hay en el frasco. Por lo tanto, necesita un

- parámetro que sea la cantidad de dulce que se desea sacar, en gramos. Este parámetro se llamará “cantidadSacar” y será un valor entero (*int*).
  - Retorno: Hay que tener en cuenta que no siempre se alcanza a sacar todo lo que se desea porque a veces hay menos cantidad de lo que se cree. Por eso este método debe indicar cuánta cantidad efectivamente se pudo sacar. Es decir, este método retornará un valor entero (*int*).
- consultarCantidad:
  - Parámetros: No necesita recibir ningún valor como parámetro, porque la información que necesita, que es la cantidad que hay en el frasco, ya está como atributo.
  - Retorno: Este método debe retornar un valor, porque lo que se necesita es precisamente saber la cantidad de dulce que hay en el frasco. Por lo tanto, debe retornar un valor entero (*int*).

Después de este análisis el diagrama de clases con la información detallada de los métodos queda:

| FrascoDeDulce   |
|---|
| sabor: String<br>cantidad: int<br>fecha: String   |
| envasarDulce(cantidadEnvasar: int, fechaEnvasado: String): boolean<br>sacarDulce(cantidadSacar: int): int<br>consultarCantidad(): int |

El código correspondiente a esta clase, incluyendo los métodos, es:

```
/**
 * Un frasco con delicioso dulce, elaborado por la abuela pata.
 * Por ejemplo, un frasco de dulce de guayaba.
 * @author Sandra V. Hurtado
 * @version 1.5
 */
public class FrascoDeDulce
{
    String sabor;
    int cantidad;
    String fecha;

    /**
     * Envasar o insertar dulce en el frasco, hasta máximo 200 gramos.
     * @param cantidadEnvasar la cantidad, en gramos,
     *                        que se desea envasar de dulce
     * @param fechaEnvasado fecha en la cual se envasa el dulce
     * @return una indicación (true/false) de si se pudo envasar o no
     *        toda la cantidad deseada o solo 200 gramos
     */
    //continúa
```

```

boolean envasarDulce(int cantidadEnvasar, String fechaEnvasado)
{
    fecha = fechaEnvasado;

    if (cantidadEnvasar > 200)
    {
        cantidad = 200;
        return false;
    }
    else
    {
        cantidad = cantidadEnvasar;
        return true;
    }
}

/**
 * Saca cierta cantidad del dulce del frasco, si hay suficiente
 * en el frasco; pero si hay menos solo se puede sacar lo que queda.
 * @param cantidadSacar    la cantidad, en gramos,
 *                          que se desea sacar de dulce
 * @return    la cantidad de dulce que se pudo sacar, en gramos
 */
int sacarDulce(int cantidadSacar)
{
    int cantidadSacada = 0;
    if (cantidad >= cantidadSacar)
    {
        cantidad = cantidad - cantidadSacar;
        cantidadSacada = cantidadSacar;
    }
    else
    {
        cantidadSacada = cantidad;
        cantidad = 0;
    }
    return cantidadSacada;
}

/**
 * Consulta la cantidad de dulce del frasco
 * @return    la cantidad de dulce que hay en el frasco, en gramos
 */
int consultarCantidad()
{
    return cantidad;
}
} // fin clase FrascoDeDulce

```

## 6. Referencias y Operador Punto

En este capítulo se explicará el concepto de referencia en el lenguaje de programación Java, que es la forma en la cual se trabaja con los objetos. Se mostrará la diferencia entre los valores y las referencias; se explicará el uso de los métodos en los objetos con el operador punto y se presentará el valor especial *null*.

A finalizar este capítulo usted debe ser capaz de:

- Explicar la diferencia entre un valor y una referencia en Java.
- Usar el operador punto para usar un servicio de un objeto, es decir, llamar uno de sus métodos.

### 6.1. Tipos de variables en Java

En un programa orientado a objetos lo primero que debe hacerse es definir las clases que se necesitarán. Sin embargo, después de crear las clases, y para que el programa logre los objetivos deseados, es muy importante crear objetos de esas clases y trabajar con sus valores y métodos.

Para poder crear y utilizar los objetos en un lenguaje de programación como Java, es necesario tener variables. Java tiene dos grandes grupos de tipos de datos para poder crear variables:

- Los primitivos (o básicos): corresponden a valores simples y son: *boolean*, *byte*, *short*, *int*, *long*, *float*, *double* y *char*.
- Los referenciados (u objeto): corresponden a las clases, y es el tema que se presentará en este capítulo.

### 6.2. Referencias

Cuando se trabaja con tipos primitivos las variables **contienen** un dato o valor, pero cuando se trabaja con objetos las variables **referencian** el objeto, es decir, no lo contienen, sino que permiten llegar al objeto para usarlo; es como un alias o un apodo con el cual conocemos a una persona.

Por ejemplo, se definirán dos variables: una primitiva y otra referenciada, cada una con un valor inicial.

```
int ancho = 20;  
String frase = new String("¡Viva aprender!");
```

Una representación de cómo queda la memoria puede ser:



Las variables de tipo primitivo se usan directamente en las expresiones, mientras que las variables referenciadas por lo general no se usan directamente, sino que a través de ellas se pueden llamar los métodos del objeto. Por ejemplo:

```
int dobleAncho = ancho * 2;
```

→ Se usa directamente

```
String palabraInicial = frase.substring(1,5);
```

→ Permite usar el método "substring"

Al igual que las variables de tipo primitivo, las variables referenciadas pueden declararse antes de usarlas, pero en ese caso todavía no están referenciando ningún objeto, de la misma forma que una variable primitiva que no se ha inicializado no tiene ningún valor. Por ejemplo:

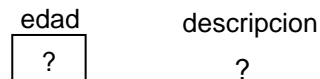
```
int edad;  
String descripcion;
```

La variable "edad":

- Es de tipo *int*, es decir, un tipo primitivo.
- Por el momento no tiene ningún valor.

La variable "descripción":

- Es de tipo *String*, es decir, un tipo referenciado (*String* es una clase).
- Por el momento no referencia a ningún objeto.

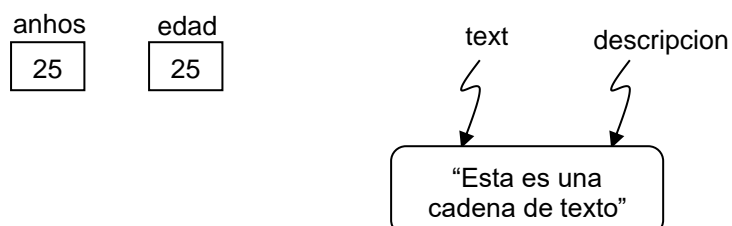


Para poder usar estas variables es necesario asignarles algún valor o referencia. A una variable referenciada se le puede asignar un objeto que ya existe, o uno nuevo, usando la instrucción *new*.

Cuando a una variable primitiva se le asigna otra variable, lo que se hace es copiar el valor, mientras que cuando a una variable referenciada se le asigna otra variable, lo que se hace es que referencia **el mismo objeto** (NO crea una copia).

Por ejemplo:

```
int anhos = 25;  
edad = anhos;  
String texto = new String("Esta es una cadena de texto");  
descripcion = texto;
```





En otro ejemplo, se tiene la clase Cuenta:

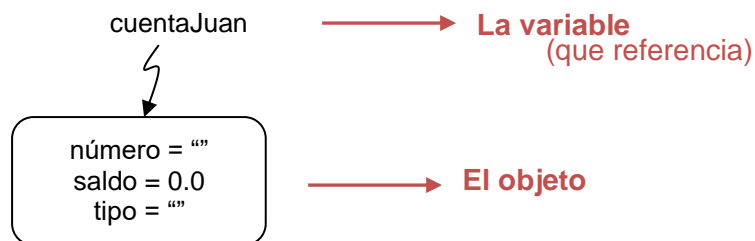
| Cuenta  |
|---|
| número: String<br>saldo: double<br>tipo: String   |
| consultarSaldo(): double<br>consignar(cantidad: double): void<br>retirar(cantidad: double): boolean |

Para crear un objeto de esta clase en Java, la instrucción es:

```
Cuenta cuentaJuan = new Cuenta();
```

Lo que se está haciendo es: creando un objeto de la clase Cuenta, que es **referenciado** por la variable “cuentaJuan”.

Una forma de representar esto es:



Esto también se puede escribir en dos pasos, así:

```
Cuenta cuentaJuan;
```

En este primer paso se declara la variable. Es una variable llamada “cuentaJuan” de tipo Cuenta, es decir, para trabajar con objetos de esa clase, pero actualmente no está referenciando nada.

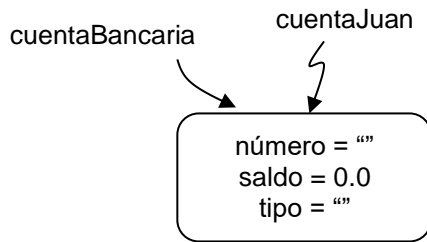
```
cuentaJuan = new Cuenta();
```

En este segundo paso se crea el objeto Cuenta (con **new**) y se asigna para que sea referenciado por la variable “cuentaJuan”.

Ahora, se define otra variable de tipo Cuenta, y se le asigna el objeto que se acabó de crear, así:

```
Cuenta cuentaBancaria = cuentaJuan;
```

Una representación de cómo queda la memoria puede ser:



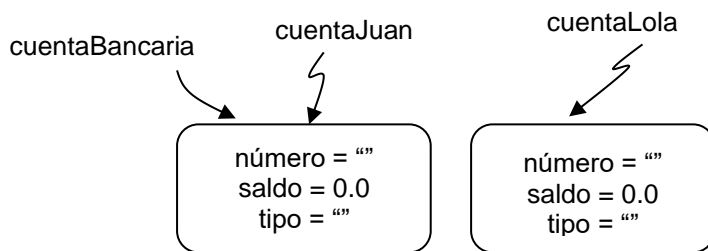
NO se ha creado un nuevo objeto.  
La variable "cuentaBancaria"  
referencia **el mismo objeto** anterior.

Las dos variables están referenciando al mismo objeto.

Ahora se crea un nuevo objeto:

```
Cuenta cuentaLola = new Cuenta();
```

Lo cual se representa en memoria:



Se tienen **dos** objetos  
Cuenta.

### **Ejercicio 6-1**

Se tiene la siguiente clase y unas instrucciones en Java:

| Gato                                       |
|--|
| maullar(veces: int, volumen: String): void |

```
Gato gatito;  
Gato consentido = new Gato();
```

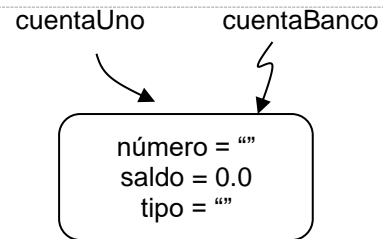
A partir del código anterior responda las siguientes preguntas:

- ¿Cuántos objetos de la clase Gato se crean?
- ¿Cuántas variables se están definiendo? ¿De qué tipo son?
- Elabore el dibujo que muestre cómo quedarían las variables y los objetos en memoria.
- Escriba la instrucción en Java para crear un objeto Gato y que sea referenciado por la variable gatito.

Volviendo al ejemplo con la clase Cuenta:

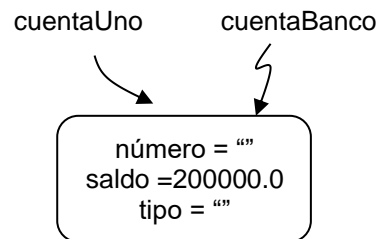
```
/* Código donde se crea un objeto Cuenta,  
   que es referenciado por dos variables  
   */
```

```
Cuenta cuentaUno = new Cuenta();  
Cuenta cuentaBanco = cuentaUno;
```



```
/* Ahora se consigna en la cuenta,  
   usando la variable cuentaUno  
   */
```

```
cuentaUno.consignar(200000);
```



```
/* Se muestra el saldo,  
   usando la variable cuentaBanco  
   */
```

```
System.out.println("Saldo: $" + cuentaBanco.consultarSaldo());
```

El resultado es:

```
Saldo: $200000.0
```

Aunque se usen diferentes variables se está trabajando con el mismo objeto.

### 6.3. Uso de los métodos con el operador punto

Después de crear los objetos, se pueden usar los servicios que ofrecen, es decir, **llamar** a los métodos para realizar alguna operación.

Siguiendo con el ejemplo de la clase Cuenta, se observa que tiene el método "consultarSaldo". Usando este método es posible saber el saldo o cantidad de dinero de una cuenta, pero antes por lo general se consigna o se retira algún valor, para lo cual se pueden usar los métodos "consignar" y "retirar".

Por ejemplo, para obtener y mostrar el saldo de una cuenta en la cual se consigna \$150.000 y se retiran \$50.000 y \$20.000, el código en Java es:

```
Cuenta cuenta = new Cuenta();
cuenta.consignar(150000);
boolean pudoPrimerRetiro = cuenta.retirar(50000);
boolean pudoSegundoRetiro = cuenta.retirar(20000);
double saldoFinal = cuenta.consultarSaldo();
System.out.println("El saldo es: " + saldoFinal);
```

Como se puede ver en el ejemplo anterior, la forma de usar (o llamar) los métodos de un objeto es con el operador punto (.).

Primero va la variable, sigue el punto y luego el método que se desee utilizar.

La sintaxis es:



Se debe tener en cuenta:

- Si el método retorna algo es adecuado guardar (asignar) lo que retorna en una variable, como en el ejemplo del saldo:

```
double saldoFinal = cuenta.consultarSaldo();
```

- Si el método no retorna nada (*void*) no se necesita ninguna variable. Como en el ejemplo de consignar:

```
cuenta.consignar(150000);
```

- Si el método recibe parámetros, se deben enviar **los valores** para cada parámetro, en el orden adecuado y de acuerdo con los tipos de datos definidos.

Por ejemplo, se desea usar el método “maullar” que está definido en la clase Gato.

El método está definido así:

```
maullar(veces:int, volumen:String):void
```

Es decir, para usar el método “maullar” se deben enviar dos parámetros: un entero (las veces que maullará) y una cadena (el volumen del maullido).

Por lo tanto, las instrucciones en Java para crear un objeto Gato y llamar a su método maullar, indicándole que maúlle 3 veces a volumen bajo serían:

```
Gato mascota = new Gato();
mascota.maullar(3, "bajo");
```

## **Ejercicio 6-2**

Escriba las instrucciones en Java para crear un objeto Gato y decirle que maúlle una vez muy alto.

### Ejercicio 6-3

Elabore un dibujo que represente cómo quedarían las variables y los objetos en el siguiente código:

```
String mensaje1 = "Buenos días";  
String mensaje2 = mensaje1;  
String mensaje3 = "¿Cómo está?";
```

## 6.4. Valor *null*

Cuando se definen variables dentro de un método es recomendable asignarles un valor inicial. Esto ayuda a evitar errores en la ejecución del programa.

Por ejemplo, el siguiente código es erróneo:

```
int suma;  
suma = suma + 1; //error
```

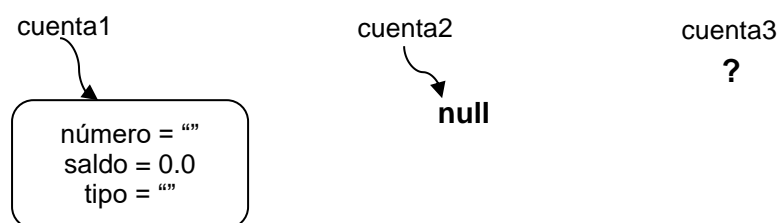
Al tratar de ejecutarlo, como la variable “suma” no tiene un valor inicial, no se puede ejecutar la segunda instrucción.

Lo mismo sucede para las **variables de tipo objeto o referenciadas**: Si no se le asigna ningún valor queda con “basura”, lo cual puede llevar a errores. Por este motivo, en Java se definió un valor especial para indicar que una variable de este tipo no está referenciando a ningún objeto: este valor es ***null***.

Por ejemplo, se tiene el siguiente código:

```
Cuenta cuenta1 = new Cuenta();  
Cuenta cuenta2 = null;  
Cuenta cuenta3;
```

En el anterior código se tienen tres variables de tipo Cuenta, de las cuales la primera (cuenta1) está referenciando un objeto, la segunda (cuenta2) tiene valor *null* y la última (cuenta3) no tiene ningún valor definido, así:



Posteriormente se desea saber la cantidad de dinero (o el saldo) que tiene cada cuenta – lo cual solo funcionará para la variable que está referenciando un objeto.

Es decir, en el siguiente código solo es correcta la primera instrucción, las dos siguientes generan errores al compilar o al ejecutar el programa, porque las variables no están referenciando ningún objeto:

```
System.out.println(cuenta1.consultarSaldo());  
System.out.println(cuenta2.consultarSaldo());    //ERROR  
System.out.println(cuenta3.consultarSaldo());    //ERROR
```

Para evitar estos errores se modificará el código. De esta manera, antes de preguntar por el saldo, se validará si la variable está referenciando un objeto, lo cual se hace preguntando si es diferente de **null**, así:

a) Para cuenta1:

```
if (cuenta1 != null)  
{  
    System.out.println(cuenta1.consultarSaldo());  
}
```

Este código funciona apropiadamente y muestra el saldo que tiene la cuenta.

b) Para cuenta2:

```
if (cuenta2 != null)  
{  
    System.out.println(cuenta2.consultarSaldo());  
}
```

Este código también funciona apropiadamente, y como la variable está con valor **null**, no entra al *if*. No se muestra el mensaje, pero no sale error y el programa continúa normalmente.

c) Para cuenta3:

```
if (cuenta3 != null)  
{  
    System.out.println(cuenta3.consultarSaldo());  
}
```

Este código **genera un error**. Como la variable **cuenta3** no tiene un valor definido –ni siquiera **null**– no se puede usar.

En resumen, con respecto al valor **null**:

- Se usa el valor **null** para inicializar una variable de tipo objeto (referenciada) cuando todavía no tiene un objeto asociado.
- Al comparar una variable de tipo objeto con el valor **null** se puede saber si tiene asociado un objeto o no.
- Si trata de usar una variable referenciada que no tiene ningún valor –ni siquiera **null**– se generará un error.
- Se pueden usar las variables que tengan valor **null**, pero solo para validar si están referenciando o no a un objeto.
- **Solo se pueden usar los métodos en variables que referencien un objeto.** No se pueden usar en variables que no tengan valor o que sean **null** porque se generaría un error.

## 6.5. Ejercicio resuelto

Considerando que se tiene la clase `FrascoDeDulce` (el diagrama se muestra a continuación), escriba el código en Java para crear un frasco, tratar de envasar 250 gramos de dulce, y mostrar si fue posible envasar los 250 gramos o solo 200. Luego intentar sacar 150 gramos y 60 gramos y mostrar en cada caso la cantidad de dulce que efectivamente se sacó y la cantidad de dulce que quedó en el frasco.

| <b>FrascoDeDulce</b>  |
|---|
| sabor: String<br>cantidad: int<br>fecha: String   |
| envasarDulce(cantidadEnvasar: int, fechaEnvasado: String): boolean<br>sacarDulce(cantidadSacar: int): int<br>consultarCantidad(): int |

```
/**
 * Creación de un objeto FrascoDeDulce, usando sus métodos
 * para envasar, sacar dulce y consultar la cantidad.
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class PruebaDulces
{
    public static void main (String args [])
    {
        // Crear el frasco de dulce
        FrascoDeDulce frasco = new FrascoDeDulce();

        // Tratar de envasar 250 gramos
        boolean pudoEnvasar = frasco.envasarDulce(250,"10/09/2017");
        if (pudoEnvasar)
        {
            System.out.println("Se envasaron 250 gramos");
        }
        else
        {
            System.out.println("Se envasaron solo 200 gramos");
        }

        // Sacar 150 gramos
        int cantidadSacada = frasco.sacarDulce(150);
        int cantidadFinal = frasco.consultarCantidad();
        System.out.println("Del frasco se sacaron "+
            cantidadSacada + " gramos, y quedan: "+ cantidadFinal);

        //continúa
    }
}
```

```
// Sacar 60 gramos  
cantidadSacada = frasco.sacarDulce(60);  
cantidadFinal = frasco.consultarCantidad();  
System.out.println("Del frasco se sacaron "+  
    cantidadSacada + " gramos, y quedan: "+ cantidadFinal);  
}  
} // fin clase PruebaDulces
```

El resultado de ejecutar este programa es:

```
Se envasaron solo 200 gramos  
Del frasco se sacaron 150 gramos y quedan 50  
Del frasco se sacaron 50 gramos y quedan 0
```



## 7. Estructuras Repetitivas en Java

En este capítulo se presenta la última parte de la introducción al lenguaje de programación Java, presentando las estructuras repetitivas o ciclos.

Al finalizar este capítulo usted debe ser capaz de:

- Describir tres instrucciones repetitivas en Java: *while*, *do-while* y *for*.
- Incluir estructuras repetitivas en un método en Java.

### 7.1. Definición

Cuando se elaboran programas a veces se identifica un conjunto de instrucciones que se repiten. Por ejemplo, si se desea calcular el peso promedio de un grupo de 20 estudiantes, hay que preguntar 20 veces por el peso (por cada estudiante) e ir sumando para poder hacer al final el cálculo del promedio, que es la suma de los pesos dividido 20. Las instrucciones que se repiten en este caso son preguntar el peso e ir sumando. No es necesario escribirlas 20 veces, sino que se puede escribir una sola vez y enmarcarlas en una estructura repetitiva para indicar al programa que se ejecutan varias veces.

Las estructuras repetitivas, por lo tanto, permiten repetir un conjunto de instrucciones dependiendo de una condición. Mientras la condición sea verdadera las instrucciones se seguirán ejecutando. El ciclo se detiene cuando la condición se vuelva falsa.

En Java se tienen tres estructuras para repeticiones: *while*, *do-while* y *for*, las cuales se presentarán a continuación.

### 7.2. Ciclo *while*

La sintaxis es:

```
while (condición)
{
    // instrucciones
}
```

Si la condición es verdadera **entra** (o repite) el ciclo, para ejecutar las instrucciones.

Por ejemplo, se tiene la clase Sopa:

| Sopa  |
|---|
| cantidad: double  |
| sacar(cantidadSacar: double): void<br>getCantidad(): double |

El método “sacar” es para sacar una cantidad de sopa, en onzas.

El método “getCantidad” es para consultar la cantidad de sopa que hay en el momento.

Se desea repartir sopa para un almuerzo. Cada plato es de 20 onzas y se desea saber cuántos platos se alcanzan a repartir con una olla de sopa.

Para esto se tendrá otra clase, llamada ReparticiónSopa:

| ReparticiónSopa           |
|---------------------------|
| calcularPlatosSopa(): int |

En el método “calcularPlatosSopa” se creará un objeto Sopa y se sacarán 20 onzas cada vez. Se continúa sacando sopa mientras aún alcance para un plato, es decir, mientras haya más de 20 onzas de sopa en el frasco.

Cuando haya menos de 20 onzas de sopa se termina de repartir, y se debe decir cuántas veces se pudo sacar sopa, es decir, cuántos platos se sacaron.

```
/**
 * Reparten varios platos de sopa, usando un ciclo.
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class ReparticionSopa
{
    /**
     * Calcula cuántos platos se pueden sacar de una sopa
     * @return    los platos de 20 onzas que se sacan de una sopa
     */
    int calcularPlatosSopa()
    {
        Sopa ollaSopa = new Sopa();
        int platos = 0;

        // Sigue mientras haya cantidad suficiente para otro plato
        while (ollaSopa.getCantidad() >= 20)
        {
            ollaSopa.sacar(20);
            platos++;
        }

        return platos;
    }
}
```

### 7.3. Ciclo *do-while*

Este ciclo es muy parecido al ciclo *while*, y su única diferencia es que la condición no se evalúa al principio sino al final del ciclo.

Como la condición se evalúa al final, este ciclo garantiza que las instrucciones se ejecutan por lo menos una vez.

La sintaxis es:

```
do
{
    // instrucciones
} while (condición);
```

Si la condición es verdadera **regresa** al ciclo ejecutando las instrucciones.

A diferencia de otras estructuras de control en Java – condicionales y repetitivas-, en este caso la instrucción termina con punto y coma. Esto se debe a que la condición va al final del bloque de instrucciones, y no al comienzo como todas las demás.

Por ejemplo, se tiene la clase Cajero, que representa cajeros electrónicos:

| Cajero  |
|---|
| cantidadDinero: int   |
| getCantidadDinero(): int<br>retirarDinero(cantidad: int): boolean |

El método “getCantidadDinero” es para saber la cantidad de dinero que tiene el cajero. El método “retirarDinero” es para retirar alguna cantidad del cajero. Retorna true si tenía dinero suficiente para retirar, y false si no pudo retirar porque no tenía dinero suficiente.

Se desea hacer una prueba con un cajero electrónico, para ir retirando de a 200.000 cada vez, mientras se pueda retirar, y al final debe mostrar cuánto dinero quedó en el cajero. Para esto se tendrá la clase DiagnósticoCajero, con el método *main*.

| DiagnósticoCajero                  |
|------------------------------------|
|                                    |
| <u>+main(args: String[]): void</u> |

El código de esta clase es:

```
/**
 * Realiza diagnósticos o pruebas a un cajero electrónico,
 * retirando dinero varias veces, mientras todavía tenga suficiente.
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class DiagnosticoCajero
{
    public static void main(String[] args)
    {
        Cajero cajeroPrueba = new Cajero();
        boolean pudoRetirar;

        //continúa
```

```

        // retira 200000 del cajero y sigue si pudo retirar
do
{
    pudoRetirar = cajeroPrueba.retirarDinero(200000);

} while (pudoRetirar);

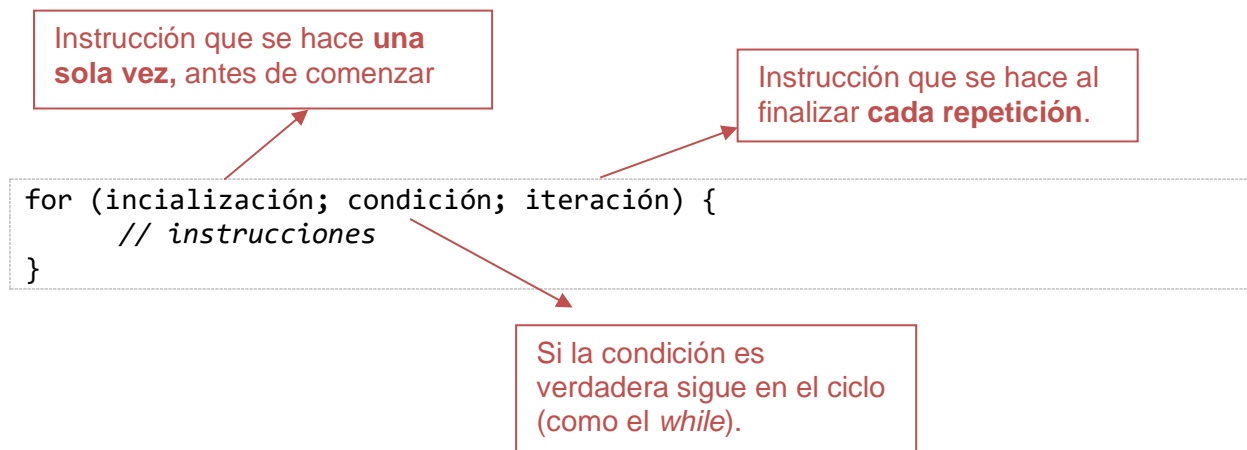
int dineroRestante = cajeroPrueba.getCantidadDinero();
System.out.println("Dinero restante: "+ dineroRestante);
}
} // fin clase DiagnosticoCajero

```

## 7.4. Ciclo for

Este ciclo permite incluir, en el encabezado, una instrucción de inicialización y una de iteración. Es decir, en el *for* –además de la condición– se puede incluir una instrucción que se hace **antes** de comenzar el ciclo (una sola vez) y una que se hace al final de **cada repetición**.

La sintaxis es:



Por ejemplo, se tiene la clase Televisor:

| Televisor   |
|---|
| pulgadas: int   |
| setPulgadas(pulgadasTV: int): void<br>calcularConsumo(): double |

El método “setPulgadas” es para definir el tamaño, en pulgadas, que tendrá el televisor. El método “calcularConsumo” es para saber cuál es el consumo, en vatios, del televisor.

Se desea un programa en Java para crear 10 televisores de diferentes tamaños, empezando en 31 pulgadas, e incrementando en una pulgada cada vez hasta 40 pulgadas. De cada

televisor se debe mostrar cuál es el consumo, y al final se debe mostrar cuál es el consumo total de todos los televisores.

| FábricaTelevisores                 |
|------------------------------------|
| <u>+main(args: String[]): void</u> |

El código de esta clase queda:

```
/**
 * Fábrica que construyen televisores de diferentes pulgadas
 * y calcula y muestra su consumo, en vatios.
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class FabricaTelevisores
{
    public static void main(String[] args)
    {
        double consumoTotal = 0;

        // comienza con pulgadas en 31, y
        // mientras sea menor o igual a 40 sigue construyendo televisores.
        // En cada iteración incrementa en 1 las pulgadas
        for (int pulgadas=31; pulgadas <= 40; pulgadas++)
        {
            Televisor tele = new Televisor();
            tele.setPulgadas(pulgadas);
            double consumoTv = tele.calcularConsumo();
            consumoTotal += consumoTv;
            System.out.println("Consumo televisor es:"+consumoTv);
        }
        System.out.println("Total consumo: "+ consumoTotal);
    }
}
```

## 7.5. Instrucción **break**

Esta instrucción permite terminar un ciclo sin importar si se cumple la condición o no. Cuando se encuentra esta instrucción el programa sale del ciclo y continúa con las instrucciones que hay después del ciclo.

Por ejemplo, se desea modificar el programa de los televisores para que el ciclo se detenga si algún televisor tiene un consumo mayor a 50. En este caso se debe mostrar de cuántas pulgadas es el televisor con este alto consumo.

```

/**
 * Fábrica que construyen televisores de diferentes pulgadas
 * y calcula y muestra su consumo, en vatios.
 * Se desea un máximo consumo de 50 vatios, si algún televisor
 * supera esta cantidad se detiene la producción.
 * @author Sandra V. Hurtado
 * @version 2.0
 */
public class FabricaTelevisores
{
    public static void main(String[] args)
    {
        double consumoTotal = 0;

        for (int pulgadas=30; pulgadas <= 40; pulgadas++)
        {
            Televisor tele = new Televisor();
            tele.setTamaño(pulgadas);
            double consumoTv = tele.calcularConsumo();
            consumoTotal += consumoTv;
            System.out.println("Consumo televisor es:"+consumoTv);

            if (consumoTv > 50)
            {
                System.out.println("Pulgadas alto consumo: "+pulgadas);
                // Termina el ciclo.
                break;
            }
        }
        System.out.println("Total consumo: "+ consumoTotal);
    }
}

```

### **Ejercicio 7-1**

Implemente lo siguiente con un ciclo *for*:

```

int edad = 0;
while (edad < 5)
{
    System.out.println("Feliz cumpleaños # "+ edad);
    edad = edad + 1;
}

```

### **Ejercicio 7-2**

Se tiene la clase Cuadrado:

| <b>Cuadrado</b>                                      |
|--|
| lado: int  |
| setLado(valorLado: int): void<br>calcularÁrea(): int |

El método “setLado” es para definir el tamaño, en centímetros, que tendrá el cuadrado.  
El método “calcularÁrea” es para calcular el área del cuadrado.

Escriba un programa en Java que permita crear 11 cuadrados de diferentes lados, comenzado en un cuadrado de 25 centímetros de lado e incrementando cada vez en cinco centímetros. Al final del programa se debe mostrar cuál es el área total, es decir, la suma de las áreas de todos los cuadrados.

## 7.6. Ejercicio resuelto

Considerando que se tiene la clase FrascoDeDulce (el diagrama se muestra a continuación), escriba el código en Java para pedir al usuario cuántos frascos de dulce desea envasar, y luego se debe pedir la cantidad (en gramos) de dulce que se envasará en cada frasco. Al finalizar de envasar los frascos se debe decir cuántos gramos totales de dulce se envasaron.

| <b>FrascoDeDulce</b>  |
|---|
| sabor: String<br>cantidad: int<br>fecha: String   |
| envasarDulce(cantidadEnvasar: int, fechaEnvasado: String): boolean<br>sacarDulce(cantidadSacar: int): int<br>consultarCantidad(): int |

Para resolver el ejercicio, se creará otra clase, llamada EnvasaFracosDulce, que tendrá el método *main*.

| <b>EnvasaFracosDulce</b>           |
|------------------------------------|
| <u>+main(args: String[]): void</u> |

Dentro de este método se realizará lo propuesto en el enunciado, como se observa a continuación.

```

import javax.swing.JOptionPane;
/**
 * Programa para envasar varios dulces, solicitando de cada uno
 * los gramos que se envasarán y mostrando la cantidad total al final.
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class EnvasaFrascosDulce
{
    public static void main (String args [])
    {
        // Variable para guardar el total envasado (en gramos)
        int cantidadTotalEnvasada = 0;

        // Pregunta cuántos frascos desea envasar
        String cadenaFrascos = JOptionPane.showInputDialog(
            "Cantidad de frascos que desea envasar ");
        int cantidadFrascos = Integer.parseInt(cadenaFrascos);

        // Ciclo para envasar los frascos
        for (int i=0; i<cantidadFrascos; i++)
        {
            FrascoDeDulce frascoDulce = new FrascoDeDulce();
            String cadenaGramos = JOptionPane.showInputDialog(
                "Cantidad de dulce (en gramos) que desea envasar: ");
            int cantidadGramos = Integer.parseInt(cadenaGramos);
            frascoDulce.envasarDulce(cantidadGramos, "17/07/2017");

            //Va sumando la cantidad de cada frasco
            cantidadTotalEnvasada += frascoDulce.consultarCantidad();
        }

        JOptionPane.showMessageDialog(null, "Se envasaron en total "+
            cantidadTotalEnvasada + " gramos de dulce");
    }
} // fin clase EnvasaFrascosDulce

```



## 8. Referencia *this*

En este capítulo se presentará la palabra reservada *this* en Java, para poder hacer referencia a los atributos y métodos del mismo objeto con el cual se está trabajando. Esta referencia ayuda a evitar confusiones cuando se tienen algunas variables con el mismo nombre, y por lo tanto ayuda a que los programas sean más fáciles de entender.

Al finalizar este capítulo usted debe ser capaz de:

- Usar la referencia *this* para hacer uso de atributos y métodos del objeto dentro de la misma clase.

### 8.1. Uso de métodos y atributos propios

Recordemos:

- Después de crear los objetos, se pueden usar los servicios que ofrecen, tanto los atributos como los métodos.
- La forma de usar o llamar los atributos o métodos de un objeto es con el operador punto (.). Primero va la variable, sigue el punto y luego el atributo o método que se desee utilizar.

Lo anterior aplica para métodos que son de otros objetos, pero ¿qué pasa si se desea usar un método del mismo objeto?

Por ejemplo, en una empresa que vende teléfonos celulares se tiene la clase Celular:

| <b>Celular</b>   |
|--|
| marca: String<br>tipo: char  |
| setMarca(marca: String): void<br>setTipo(tipo: char): void<br>calcularValor(): double<br>calcularIva(): double |

El método “calcularValor” está definido de la siguiente forma:

```
/**
 * Calcula el valor del celular, así:
 * Los de tipo 'c' valen 50.000, los de tipo 'b' 130.0000
 * y los de de tipo 'a' 250.000
 * @return el valor del celular
 */
```

*//continúa*

```
double calcularValor()
{
    double valor = 0;
    switch (tipo)
    {
        case 'a':    valor = 250.000;
                     break;
        case 'b':    valor = 130.000;
                     break;
        case 'c':    valor = 50.000;
                     break;
    }
    return valor;
}
```

El IVA es el 16 % del valor del celular, es decir, para poder calcular el IVA se necesita saber cuál es el valor del celular.

Por lo tanto, para no repetir código, dentro el método “calcularIva” se llamará al método “calcularValor” y así se obtendrá el valor del celular. Es decir, se usará un método del mismo objeto.

Cuando se usa un **método que es del mismo objeto**, se tienen dos opciones para usarlo:

- a) Usar la palabra **this** para representar el objeto (como diciendo “**este** objeto”, en el que se está trabajando”), así:

**this**

•

elMétodo( )

O,

- b) Llamar el método directamente, ya que se está dentro de la misma clase, sin usar un objeto ni el punto. Es decir:

elMétodo( )

Por lo tanto, el método “calcularIva” puede quedar de dos formas:

- a) Usando *this*:

```
/**
 * Calcula el IVA, que es el 16% del valor del celular.
 * @return el valor del IVA para el celular
 */
double calcularIva ()
{
    double valorCelular = this.calcularValor();
    double iva = valorCelular * 0.16;
    return iva;
}
```

b) Sin usar *this*:

```
/**
 * Calcula el IVA, que es el 16% del valor del celular.
 * @return el valor del IVA para el celular
 */
double calcularIva ()
{
    double valorCelular = calcularValor();
    double iva = valorCelular * 0.16;
    return iva;
}
```

En general, siempre que dentro de una clase se desee hacer referencia a sus propios elementos (atributos y métodos) se puede usar la variable de referencia *this*. El uso de la variable de referencia *this* da mayor claridad en los programas.

Esta variable representa siempre la instancia o el objeto “actual” dentro de la misma clase.

Por ejemplo, en el primer código que se presentó en este capítulo, del método “calcularValor”, se puede escribir de la siguiente forma:

```
/**
 * Calcular el valor del celular, así:
 * Los de tipo 'c' valen 50.000, los de tipo 'b' 130.0000
 * y los de de tipo 'a' 250.000
 * @return el valor del celular
 */
double calcularValor()
{
    double valor = 0;
    switch (this.tipo)
    {
        case 'a':    valor = 250.000;
                     break;
        case 'b':    valor = 130.000;
                     break;
        case 'c':    valor = 50.000;
                     break;
    }
    return valor;
}
```

## 8.2. Diferenciar atributos y parámetros

Una de las principales utilidades de la variable de referencia *this* es que permite diferenciar entre atributos de la clase y parámetros de un método.

En Java es válido que un parámetro se llame igual que un atributo, y precisamente para evitar confusiones se usa *this*.

Por ejemplo, continuando con el ejemplo de la clase Celular, el método “setMarca” recibe un parámetro llamado “marca”, que es igual al nombre del atributo. El código, aparentemente, sería:

```
/**
 * Establecer o modificar la marca del celular
 * @param marca la marca del celular
 */
void setMarca(String marca)
{
    marca = marca;
}
```

Este código es incorrecto.

El código anterior es incorrecto, porque Java identifica la variable como la de menor alcance, es decir, que AMBAS las toma como el parámetro. Por lo tanto, el código anterior no hace nada, solo asigna el valor del parámetro a él mismo, y NO se cambia la marca del celular.

Para evitar ese error, y lograr que el método efectivamente asigne el nuevo valor de la marca al celular, se usa *this* en el atributo, como se observa a continuación:

```
/**
 * Establecer o modificar la marca del celular
 * @param marca la marca del celular
 */
void setMarca(String marca)
{
    this.marca = marca;
}
```

De esta manera, el valor o la referencia del parámetro es asignado al atributo, es decir, se cambia la marca del celular, que es lo que se desea en este método.

De manera similar, en el método “setTipo” de la clase Celular se recibe un parámetro llamado tipo, que es el mismo nombre del atributo. Por lo tanto, para evitar confusiones, se debe usar la referencia *this*. El código queda:

```

/**
 * Definir el tipo del celular, el cual puede ser: a, b o c
 * Si no se ingresa un tipo válido se define como tipo c.
 * @param tipo el nuevo tipo del celular
 */
void setTipo(char tipo)
{
    if (tipo == 'a' || tipo == 'b' || tipo == 'c')
    {
        this.tipo = tipo;
    }
    else
    {
        this.tipo = 'c';
    }
}

```

Este es el parámetro

Este es el atributo

De esta forma, en los métodos se pueden usar parámetros que tengan el mismo nombre que los atributos de la clase, pero se deben diferenciar internamente con el *this*.

El código completo de la clase Celular es:

```

/**
 * Aparato de comunicación, entretenimiento y productividad, conocido
 * como celular o móvil.
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Celular
{
    String marca;
    char tipo;

    /**
     * Establecer o modificar la marca del celular
     * @param marca la marca del celular
     */
    void setMarca(String marca)
    {
        this.marca = marca;
    }
}

```

//continúa

```

/**
 * Definir el tipo del celular, el cual puede ser: a, b o c
 * Si no se ingresa un tipo válido se define como tipo c.
 * @param tipo    el nuevo tipo del celular
 */
void setTipo(char tipo)
{
    if (tipo == 'a' || tipo == 'b' || tipo == 'c')
    {
        this.tipo = tipo;
    }
    else
    {
        this.tipo = 'c';
    }
}

/**
 * Calcular el valor del celular, así:
 * Los de tipo 'c' valen 50.000, los de tipo 'b' 130.0000
 * y los de de tipo 'a' 250.000
 * @return    el valor del celular
 */
double calcularValor()
{
    double valor = 0;
    switch (this.tipo)
    {
        case 'a':    valor = 250.000;
                     break;
        case 'b':    valor = 130.000;
                     break;
        case 'c':    valor = 50.000;
                     break;
    }
    return valor;
}

/**
 * Calcula el IVA, que es el 16% del valor del celular.
 * @return    el valor del IVA para el celular
 */
double calcularIva ()
{
    double valorCelular = calcularValor();
    double iva = valorCelular * 0.16;
    return iva;
}
} // fin clase Celular

```

### **Ejercicio 8-1**

Escriba el código Java para la siguiente clase, teniendo en cuenta que el valor de un lote es de \$30.000 por metro cuadrado. Use la referencia *this* donde sea pertinente.

| <b>Lote</b>   |
|---|
| largo: double<br>ancho: double  |
| setLargo(largo: double): void<br>setAncho(ancho: double): void<br>calcularÁrea(): double<br>calcularValor(): double |

### **8.3. Ejercicio resuelto**

Se tiene la clase FrascoDeDulce, donde se han cambiado el nombre de un parámetro:

| <b>FrascoDeDulce</b>  |
|---|
| sabor: String<br>cantidad: int<br>fecha: String   |
| envasarDulce(cantidadEnvasar: int, fecha: String): boolean<br>sacarDulce(cantidadSacar: int): int<br>consultarCantidad(): int |

Se muestra el código de esta clase FrascoDeDulce, haciendo uso de la referencia *this* en los métodos.

```
/**
 * Un frasco con delicioso dulce, elaborado por la abuela pata.
 * Por ejemplo, un frasco de dulce de guayaba.
 * @author Sandra V. Hurtado
 * @version 2.0
 */
public class FrascoDeDulce
{
    String sabor;
    int cantidad;
    String fecha;
```

*//continúa*

```

/**
 * Envasar o insertar dulce en el frasco, hasta máximo 200 gramos.
 * @param cantidadEnvasar la cantidad, en gramos, que se desea envasar
 * @param fecha          la fecha en la cual se envasa el dulce
 * @return una indicación (true/false) de si se pudo envasar o no
 *         toda la cantidad deseada o solo 200 gramos
 */
boolean envasarDulce(int cantidadEnvasar, String fecha)
{
    this.fecha = fecha;
    if (cantidadEnvasar > 200)
    {
        this.cantidad = 200;
        return false;
    }
    else
    {
        this.cantidad = cantidadEnvasar;
        return true;
    }
}

/**
 * Saca cierta cantidad del dulce del frasco, si hay suficiente
 * en el frasco; pero si hay menos solo se puede sacar lo que queda.
 * @param cantidadSacar la cantidad, en gramos, que se desea sacar
 * @return la cantidad de dulce que se pudo sacar, en gramos
 */
int sacarDulce(int cantidadSacar)
{
    int cantidadSacada = 0;
    if (this.cantidad >= cantidadSacar)
    {
        this.cantidad = this.cantidad - cantidadSacar;
        cantidadSacada = cantidadSacar;
    }
    else
    {
        cantidadSacada = this.cantidad;
        this.cantidad = 0;
    }
    return cantidadSacada;
}

/**
 * Consulta la cantidad de dulce del frasco
 * @return la cantidad de dulce que hay en el frasco, en gramos
 */
int consultarCantidad()
{
    return this.cantidad;
}
} // fin clase FrascoDeDulce

```



## 9. API de Java

En este capítulo se dará una introducción corta al denominado API de Java, que presenta una gran cantidad de funciones para los programadores, facilitando así el desarrollo de los programas en este lenguaje.

A finalizar este capítulo usted debe ser capaz de:

- Definir qué es el API de Java.
- Hacer una búsqueda básica en la documentación del API de Java.

### 9.1. Definición

La mayoría de los lenguajes de programación vienen con un conjunto de bibliotecas de recursos, que contienen funciones o servicios que puede utilizar el programador. Java también tiene una biblioteca de elementos ya construidos, que puede usar el programador para elaborar aplicaciones complejas más fácilmente, y que se conoce como API (*Application Programming Interface*) de Java.

El API facilita el trabajo de los desarrolladores proporcionándoles clases que ya están construidas y que solo deben integrar en sus programas sin necesidad de programar todo su comportamiento.

El API está formado por varios paquetes, que agrupan a su vez clases y otros paquetes. Un paquete puede verse como una carpeta donde se organizan las clases.

### 9.2. Sentencia *import*

Es importante tener en cuenta que para hacer uso de clases que hay en otros paquetes se debe usar la sentencia *import*, antes del encabezado de la clase que se está escribiendo. De esta manera se está indicando al compilador de Java que busque la clase en el paquete que corresponde.

Ejemplo:

```
import java.io.File; // buscar la clase File en el paquete java.io

/**
 * Ejemplo donde se usa un elemento del API de Java: la clase "File".
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class EjemploImport
{
//continúa
```

```

/**
 * En este método se crea un objeto de la clase File, del API.
 */
public void unMetodo()
{
    File objetoArchivo = new File("archivo.txt");
}
} // fin clase EjemploImport

```

Se debe usar un *import* por cada una de las clases del API que se utilicen.

Hay una excepción: las clases que están en el paquete “java.lang” no necesitan *import*. En este paquete se encuentran, entre otras, las clases *String* y *System*.

### 9.3. Búsqueda en el API

Para que el desarrollo de un programa sea más fácil y rápido puede buscarse en el API para identificar clases y métodos que ya existen y que pueden utilizarse. Por lo general primero se identifican las características o funciones que se necesitan, y luego se determina qué paquetes y clases poseen esas características o funciones en el API.

A continuación, se muestra un cuadro con algunos paquetes del API de Java, dando para cada paquete una descripción general de las clases que contiene.

| Paquetes            | Contenido   |
|---------------------|---|
| java.lang           | Tiene las clases básicas del lenguaje Java, las cuales facilitan la ejecución de los programas.                 |
| java.util           | Clases que prestan diversas utilidades, incluyendo manejo de colecciones de objetos, archivos comprimidos, etc. |
| java.io             | Clases e interfaces que permiten la entrada y salida de datos, generalmente mediante archivos                   |
| java.awt            | Clases que soportan operaciones para manejo de interfaces gráficas y para crear dibujos.                        |
| javax.swing         | Clases que complementan las funciones de awt para crear interfaces gráficas.                                    |
| java.time           | Clases para manejo de fechas, horas y duraciones de tiempo.   |
| java.text           | Clases para el manejo de texto, fechas y números, de manera independiente al lenguaje (internacionalización).   |
| java.math           | Clases que permiten el manejo de números muy grandes o para cálculos que requieren mucha precisión              |
| javax.accessibility | Clases que permiten el trabajo con tecnologías de apoyo como terminales Braille, reconocimiento de voz, etc.    |

Existen varias formas de consultar las clases que componen el API de Java. La forma más común de consulta es a través de la documentación que se encuentra en Internet; donde siempre se encontrará la documentación actualizada de las últimas versiones de Java. Por ejemplo, la información del API de Java 8 está en:

<https://docs.oracle.com/javase/8/docs/api/>

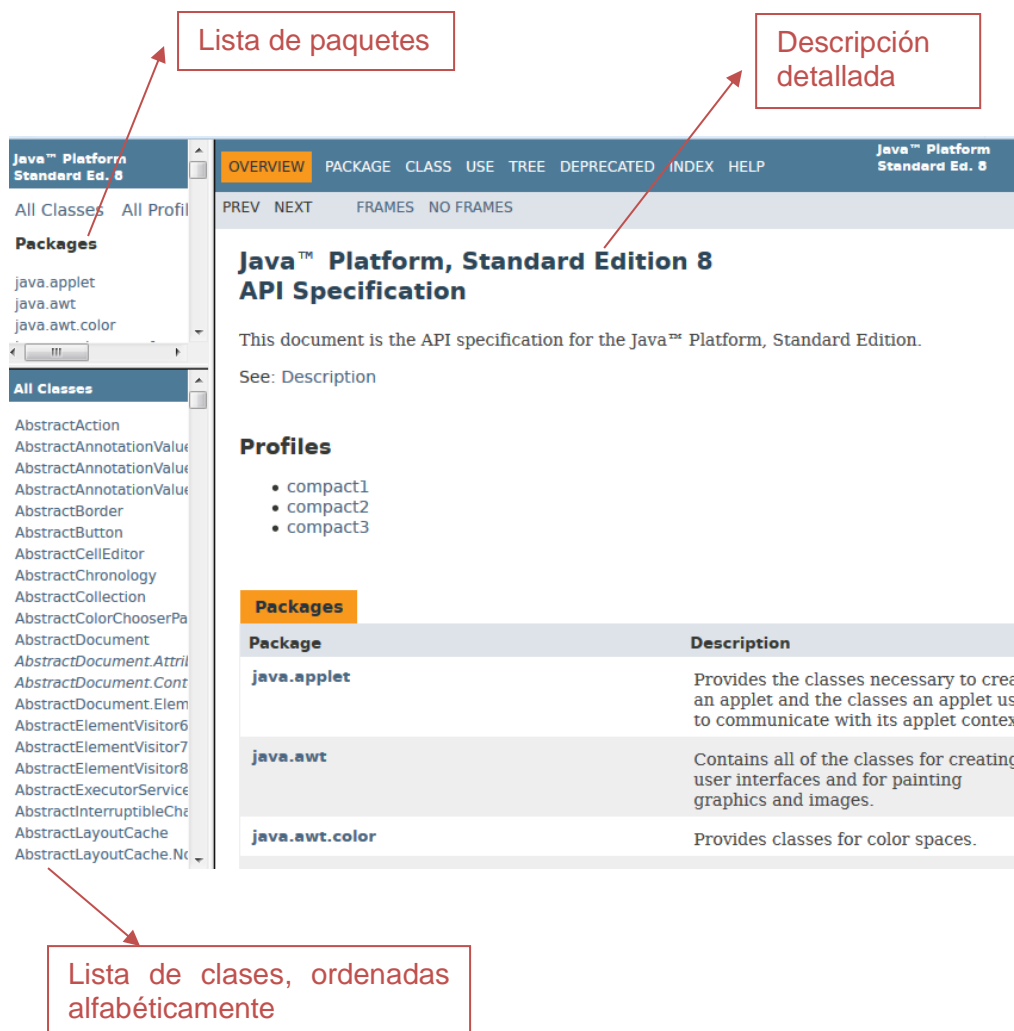


Imagen 9-1 Documentación del API de Java

Si se selecciona un paquete, en la parte de inferior izquierda se muestra la lista de las clases que conforman ese paquete.

Si se selecciona una clase, en la sección de descripción detallada se muestra la descripción de esta clase, junto con sus atributos y métodos.

Por ejemplo, si se selecciona la clase *JOptionPane* (que se ha usado para leer o mostrar datos al usuario), se observa lo siguiente:

javafx.swing

Class JOptionPane

java.lang.Object

java.awt.Component

java.awt.Container

javafx.swing.JComponent

javafx.swing.JOptionPane

All Implemented Interfaces:

ImageObserver, MenuContainer, Serializable, Accessible

```

public class JOptionPane
extends JComponent
implements Accessible

JOptionPane makes it easy to pop up a standard dialog box that prompts users for a value or informs them of something. For information about using
JOptionPane, see How to Make Dialogs, a section in The Java Tutorial.

While the JOptionPane class may appear complex because of the large number of methods, almost all uses of this class are one-line calls to one of
static showXxxDialog methods shown below:

```

| Method Name       | Description                                      |
|-------------------|--|
| showConfirmDialog | Asks a confirming question, like yes/no/cancel.  |
| showInputDialog   | Prompt for some input.                           |
| showMessageDialog | Tell the user about something that has happened. |

Imagen 9-2 Descripción de la clase JOptionPane en la documentación del API de Java

### Ejercicio 9-1

- Busque en el API los métodos que tiene la clase **File** del paquete “java.io”, que representa una carpeta o un archivo, e identifique cuáles métodos sirven para:
  - Determinar si la aplicación puede modificar o no el archivo
  - Determinar si se trata de un archivo oculto
  - Obtener la longitud del archivo
- Investigue en el API cómo se puede generar un número aleatorio utilizando la clase “java.util.Random”

## 9.4. Ejercicio resuelto

La clase FrascoDeDulce tiene un atributo para la fecha en la cual se elaboró y se enfrascó el dulce. Este atributo es de tipo *String*, pero no es el tipo de dato más adecuado, pues no es lo mismo una fecha que una cadena de caracteres. Se desea buscar en el API de Java, para saber si es posible encontrar alguna clase para representar mejor una fecha.

| FrascoDeDulce   |  |
|---|--|
| sabor: String<br>cantidad: int<br>fecha: String   | <div> <div> <div>Buscar en el API un tipo de dato para una fecha</div> </div> </div> |
| envasarDulce(cantidadEnvasar: int, fecha: String): boolean<br>sacarDulce(cantidadSacar: int): int<br>consultarCantidad(): int |  |

Se observa en la documentación del API que el paquete que tiene clases relacionadas con duraciones, fechas y horas es el paquete “java.time”, por lo que se decide buscar directamente en este paquete.

Las clases que tiene este paquete son:

### Package `java.time`

The main API for dates, times, instants, and durations.

See: [Description](#)

#### Class Summary

| Class                 | Description   |
|-----------------------|---|
| <b>Clock</b>          | A clock providing access to the current instant, date and time using a time-zone.                                 |
| <b>Duration</b>       | A time-based amount of time, such as '34.5 seconds'.  |
| <b>Instant</b>        | An instantaneous point on the time-line.  |
| <b>LocalDate</b>      | A date without a time-zone in the ISO-8601 calendar system, such as 2007-12-03.                                   |
| <b>LocalDateTime</b>  | A date-time without a time-zone in the ISO-8601 calendar system, such as 2007-12-03T10:15:30.                     |
| <b>LocalTime</b>      | A time without a time-zone in the ISO-8601 calendar system, such as 10:15:30.                                     |
| <b>MonthDay</b>       | A month-day in the ISO-8601 calendar system, such as --12-03.   |
| <b>OffsetDateTime</b> | A date-time with an offset from UTC/Greenwich in the ISO-8601 calendar system, such as 2007-12-03T10:15:30+01:00. |
| <b>OffsetTime</b>     | A time with an offset from UTC/Greenwich in the ISO-8601 calendar system, such as 10:15:30+01:00.                 |
| <b>Period</b>         | A date-based amount of time in the ISO-8601 calendar system, such as '2 years, 3 months and 4 days'.              |
| <b>Year</b>           | A year in the ISO-8601 calendar system, such as 2007.   |
| <b>YearMonth</b>      | A year-month in the ISO-8601 calendar system, such as 2007-12.  |
| <b>ZonedDateTime</b>  | A date-time with a time-zone in the ISO-8601 calendar system, such as 2007-12-03T10:15:30+01:00 Europe/Paris.     |
| <b>ZoneId</b>         | A time-zone ID, such as Europe/Paris.   |
| <b>ZoneOffset</b>     | A time-zone offset from Greenwich/UTC, such as +02:00.  |

Imagen 9-3 Clases del paquete `java.time` en la documentación del API de Java

Se observan que la clase más relacionada con una fecha es *LocalDate*, donde se presenta un ejemplo de una fecha: 2007-12-03.

Por lo tanto, se cambiará el tipo de dato del atributo fecha, de *String* a *LocalDate*, así:

| FrascoDeDulce  |   |
|--|---|
| sabor: String<br>cantidad: int<br>fecha: LocalDate   | ← |
| envasarDulce(cantidadEnvasar: int, fecha: LocalDate): boolean<br>sacarDulce(cantidadSacar: int): int<br>consultarCantidad(): int |   |

A continuación, se muestra parte del código para esta clase. Solo se mostrará el encabezado y los atributos. Los métodos se omiten por simplicidad.

```
import java.time.LocalDate;

/**
 * Un frasco con delicioso dulce, elaborado por la abuela pata.
 * Por ejemplo, un frasco de dulce de guayaba.
 * @author Sandra V. Hurtado
 * @version 2.2
 */
public class FrascoDeDulce
{
    String sabor;
    int cantidad;
    LocalDate fecha;

    // Se omite el código de los métodos, solo para este ejemplo.
}
```

## 10. Métodos Set, Get y Constructores

En este capítulo se presentan tres tipos de métodos muy conocidos y utilizados en Java, denominados los “set”, los “get” y los constructores. Estos métodos son usados principalmente para dar o consultar un valor de los atributos de un objeto.

Al finalizar este capítulo usted debe ser capaz de:

- Explicar qué es un método “set”, un método “get” y un constructor.
- Escribir métodos “set”, “get” y constructores para una clase, en Java.

### 10.1. Tipos de métodos

Una clase puede tener diferentes tipos de métodos, de acuerdo con lo que se desee realizar con los objetos de esta clase. Aunque los métodos pueden ser muy diferentes, existen algunos métodos especiales que son muy comunes en las clases, como los de consulta o modificación de atributos. Esto permite tener la siguiente clasificación de los métodos:

| Tipo de Método               | Utilidad   |
|------------------------------|--|
| De acceso o consulta – “get” | Para consultar el valor de <b>un</b> atributo.   |
| De modificación – “set”      | Para definir o modificar el valor de <b>un</b> atributo.   |
| Constructor                  | Permite asignar recursos iniciales que necesiten los objetos, en el momento de su construcción o creación. |
| Destructor                   | Permite liberar los recursos que tenga un objeto, en el momento de su destrucción.                         |
| <i>main</i>                  | Permite que la clase sea ejecutada por la máquina virtual de Java.   |
| Otros                        | Todos los demás métodos de una clase, que permiten prestar diferentes servicios.                           |

En este capítulo se explicarán los métodos “get”, “set” y los constructores. Los demás métodos ya han sido presentados en capítulos previos. En el caso de los destructores, como estos métodos no son muy usados en Java, no se trabajarán en este libro.

### 10.2. Métodos “get”: Acceso o consulta de un atributo

Se tiene la clase Cuenta, representada en el siguiente diagrama:

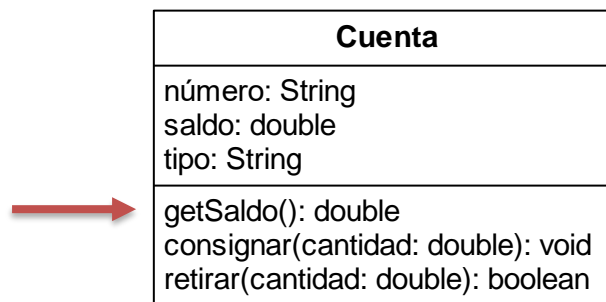
| Cuenta  |
|---|
| número: String<br>saldo: double<br>tipo: String   |
| consultarSaldo(): double<br>consignar(cantidad: double): void<br>retirar(cantidad: double): boolean |

Es posible consignar y retirar dinero de los objetos cuenta, pero ¿cómo se hace para consultar la información, por ejemplo, para saber el tipo de cuenta? Para esto se usará un método de acceso o consulta.

Es muy común tener métodos que permitan consultar u obtener el valor de un atributo, como por ejemplo el método “consultarSaldo” de la clase Cuenta. Sin embargo, para facilitar el desarrollo de los programas en Java, se ha establecido una convención internacional para dar un nombre estándar a los métodos que consultan el valor de **un** atributo.

El nombre de estos métodos debe comenzar con la palabra “**get**”, seguida del nombre del atributo.

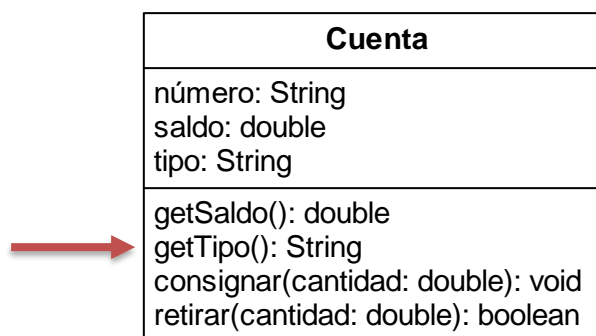
Por ejemplo, para la clase Cuenta, el método para consultar el saldo quedaría:



Un método “get” no recibe parámetros, solo debe retornar el valor del atributo, como se presenta a continuación:

```
/**
 * Consulta el saldo actual de la cuenta (el dinero que tiene).
 * @return el valor del saldo, en pesos.
 */
double getSaldo()
{
    return saldo;
}
```

De la misma forma, se puede definir otro método en la clase Cuenta para consultar su tipo, es decir, para saber si es de ahorros o corriente. El diagrama de la clase Cuenta queda:



El código de este método es:



```

/**
 * Obtiene el tipo de cuenta (ahorros o corriente)
 * @return el tipo de cuenta: "ahorros" o "corriente".
 */
String getTipo()
{
    return tipo;
}

```

Cuando se utiliza un método “get” por lo general se tiene una variable para recibir el valor que se está consultando, y así poder usarlo. Por ejemplo, en el siguiente código se crea una cuenta, se consigna un valor y luego se muestra el saldo:

```

/**
 * Usa un objeto Cuenta para consultar el saldo
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Bancaria
{
    public static void main(String arg[])
    {
        Cuenta cuenta1 = new Cuenta();
        cuenta1.consignar(100000);
        double saldoCuenta = cuenta1.getSaldo();
        System.out.println("El saldo es: "+ saldoCuenta);
    }
}

```

Otra posibilidad es incluir el llamado al método “get” directamente en una expresión, pues lo que se está usando es el valor que retorna este método. Por ejemplo, en el siguiente código se crea un objeto cuenta, se consigna un valor y luego se muestra cuál sería el valor ganado por intereses si el porcentaje de interés fuera el 2 %.

```

/**
 * Consulta el saldo de una cuenta para saber cuál sería el valor
 * ganado si recibiera intereses (del 2 %).
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class CalculoIntereses
{
    public static void main(String arg[])
    {
        Cuenta cuenta1 = new Cuenta();
        cuenta1.consignar(1000000);
        double valorInteres = cuenta1.getSaldo() * 0.02;
        System.out.println("Los intereses serían: "+ valorInteres);
    }
}

```

### Ejercicio 10-1

Escriba el código Java de la clase Carta, que se muestra a continuación:

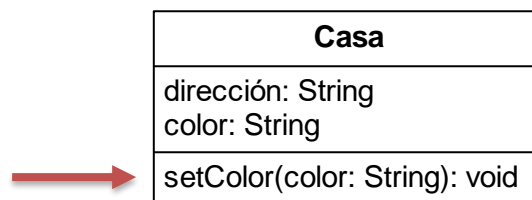
| Carta                               |
|-------------------------------------|
| número: int<br>palo: char           |
| getNúmero(): int<br>getPalo(): char |

### 10.3. Métodos “set”: Modificación de un Atributo

De manera similar a los métodos que consultan el valor de un atributo, también se tienen métodos para asignar, modificar o mutar el valor de **un** atributo.

El estándar internacional para los nombres de los métodos que asignan o modifican el valor de un atributo es: comenzar con la palabra “**set**”, seguida del nombre del atributo.

Por ejemplo, se tiene la clase Casa, que tiene la dirección y el color. En cualquier momento se puede cambiar el color de la casa y para representar esto se tendrá el método *set*, como se ilustra en el diagrama:



Un método “set” no retorna nada (*void*), pero sí recibe como parámetro el nuevo valor que se desea dar al atributo.

```
/**
 * Asignar o cambiar el color de la casa
 * @param color    el nuevo color de la fachada de la casa
 */
void setColor(String color)
{
    this.color = color;
}
```

De esta manera, se puede cambiar el color de la casa, usando el método “setColor()”. Por ejemplo, en el siguiente código se crea una casa que se pinta de blanco.

```

/**
 * Se crea un objeto casa y se le da color blanco
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class PintaCasa
{
    public static void main(String arg[])
    {
        Casa miCasita = new Casa();
        miCasita.setColor("blanco");
    }
}

```

Cuando se utiliza un método “set” no se debe tener ninguna variable para recibir un valor, ni se puede usar en expresiones.

En otro ejemplo, se tiene la clase Cuadrado, con el atributo lado, que puede ser definido en cualquier momento, y para esto se usará el método “set”. El diagrama para esta clase es:

| <b>Cuadrado</b>             |
|-----------------------------|
| lado: double                |
| setLado(lado: double): void |

Es importante tener en cuenta que el valor del lado no puede ser negativo. Esto se puede validar en el método “setLado”, para garantizar que el valor que se asigna al atributo sea adecuado.

El código para esta clase es:

```

/**
 * Figura geométrica con cuatro lados iguales
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Cuadrado
{
    double lado;

```

*//continúa*

```

/**
 * Asignar o modificar el valor del lado del cuadrado
 * @param lado    el valor del lado en centímetros, debe ser > 0
 */
void setLado(double lado)
{
    if (lado > 0)
    {
        this.lado = lado;
    }
}
} // fin clase Cuadrado

```

### **Ejercicio 10-2**

A continuación, se presenta una clase con nombres que no siguen el estándar definido para los métodos de consulta o modificación de atributos. Usted debe escribir el nombre correcto de estos métodos y luego escribir el código Java correspondiente.

*Incorrecto:*

| <b>Vehículo</b>  |
|--|
| marca: String<br>modelo: int   |
| obtenerMarca(): String<br>consultarModelo(): int<br>definirMarca(marca: String): void<br>establecerModelo(modelo: int): void |



*Correcto:*

| <b>Vehículo</b>              |
|------------------------------|
| marca: String<br>modelo: int |
|                              |

## **10.4. Constructor**

Suponga que se tiene la siguiente clase en Java, que representa objetos Cajero (como un cajero electrónico):

| <b>Cajero</b>   |
|---|
| cantidadDinero: int<br>estado: char   |
| getEstado(): char<br>getCantidadDinero(): int<br>retirarDinero(cantidad: int): boolean<br>recargarDinero(cantidad: int): void |

En esta clase, ¿cuáles serían los valores iniciales para los atributos “cantidadDinero” y “estado”? Si se crea un objeto Cajero y se pregunta por el estado, ¿Qué mostraría?

Aunque Java da unos valores iniciales a estos atributos, tal vez no corresponden a lo que se desea tener. Se desea que el cajero comience con cantidad de dinero cero y con el estado 'i', que representa inactivo.

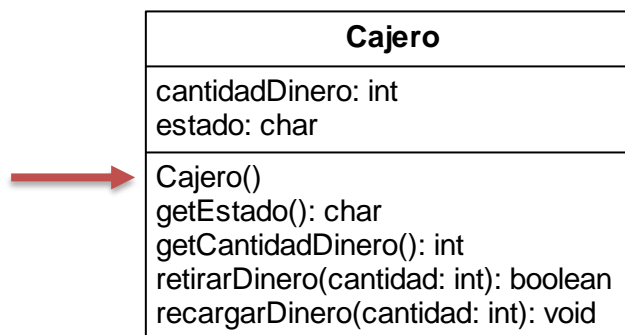
Sería ideal poder darle valores iniciales adecuados a los atributos desde que se construye el objeto, sin tener que llamar a los métodos *set* uno por uno.

Para poder dar valores iniciales a los objetos desde que se crean se usa un método especial, llamado “**constructor**”. Este método ayuda a construir o crear adecuadamente los objetos asignándoles los recursos que necesitan, entre ellos los valores de los atributos.

El método constructor es bien especial porque tiene algunas diferencias con los demás métodos:

- No tiene tipo de retorno (ni siquiera se escribe *void*)
- El nombre es **exactamente** el mismo de la clase. Si la clase comienza con mayúscula, este método también.
- Se utiliza llamado al operador *new* (no se llama con el operador punto como los demás métodos).

Por ejemplo, para definir un constructor para la clase Cajero, en el diagrama quedaría:



El código del constructor es:

```
/**
 * Constructor de los objetos Cajero, que comienzan con cero en
 * su cantidad de dinero y estado 'i' (inactivo)
 */
Cajero()
{
    this.cantidadDinero = 0;
    this.estado = 'i';
}
```

Los constructores se ejecutan cuando se crea un nuevo objeto (**con la instrucción *new***):

```
Cajero cajero1 = new Cajero();
```

Aquí se está usando  
el constructor

Ahora veamos otro ejemplo. Se tiene la clase Cuenta, representada en el siguiente diagrama:

| Cuenta   |
|--|
| número: String<br>saldo: double<br>tipo: String  |
| getSaldo(): double<br>getTipo(): String<br>consignar(cantidad: double): void<br>retirar(cantidad: double): boolean |

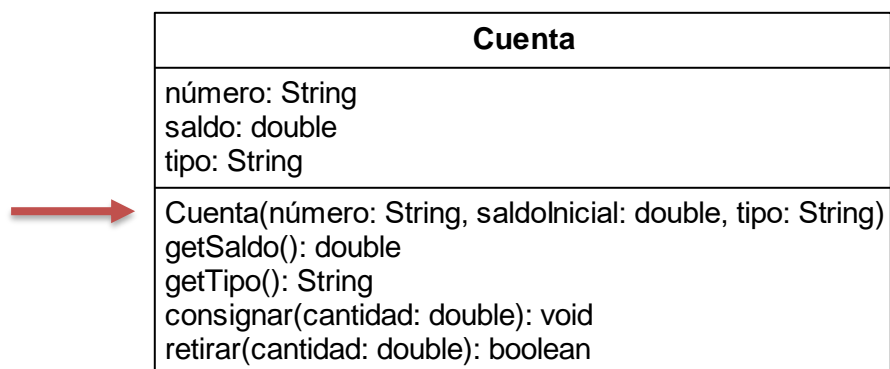
En esta clase tampoco está definido cuáles valores tienen los atributos cuando se crea un objeto. De nuevo Java les dará algunos valores, pero no necesariamente los que se necesitan.

Por esto, también se definirá un constructor para esta clase. Sin embargo, a diferencia de la clase Cajero, donde cada objeto de esta clase empezaba con cantidad cero y estado inactivo, aquí no tiene mucho sentido que todos los objetos Cuenta que se creen sean del mismo tipo o tengan el mismo número ni el mismo saldo.

Al momento de crear el objeto Cuenta se debería definir cuál es el número que tendrá, cuál es el tipo de cuenta y cuál será el saldo inicial –que es el dinero con el cual la persona puede abrir la cuenta en el banco.

Se definirá, por lo tanto, un constructor con tres parámetros: un *String* que corresponde al número de la cuenta, un *double* para el saldo inicial y un *String* para el tipo de cuenta, que puede ser “corriente” o “ahorros”.

El diagrama queda:



A continuación, se mostrará el código de este constructor:

```

/**
 * Constructor de objetos Cuenta, con un saldo inicial.
 * @param numero    el número de la cuenta
 * @param saldoInicial    saldo inicial (en pesos) de la cuenta
 * @param tipo    el tipo de cuenta: "ahorros" o "corriente"
 */
Cuenta(String numero, double saldoInicial, String tipo)
{
    this.numero = numero;
    this.saldo = saldoInicial;
    this.tipo = tipo;
}

```

En este caso, cuando se desee construir un objeto Cuenta (con **new**), será necesario enviar los tres valores requeridos como parámetros:

```
Cuenta cuenta1 = new Cuenta("897-000", 20000, "corriente");
```

Ya NO es posible crear un objeto cuenta sin enviar parámetros, como se hacía en capítulos anteriores:

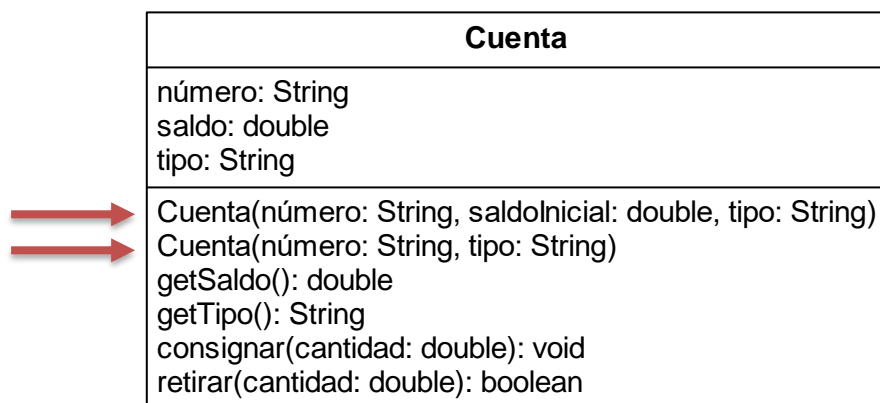
```
Cuenta cuentaNueva = new Cuenta(); //ERROR:
                                // Deben enviarse parámetros al constructor
```

## 10.5. Varios métodos constructores

Algo interesante es que una clase puede tener cero, uno o más constructores, lo que significa que se tiene más de una forma de crear los objetos.

Por ejemplo, ¿qué pasa si una persona desea abrir su cuenta bancaria sin tener dinero para el saldo inicial? En este caso no es adecuado crear el objeto con un saldo, sino que se debe crear la cuenta con saldo cero.

Pero como esto no aplica para todas las cuentas lo mejor es tener dos constructores: uno que no necesite el saldo (comenzaría con saldo cero) y otro que pida el saldo (para cuando sea diferente de cero). El diagrama de clases para Cuenta, con los dos constructores, es:



Se muestra el código completo de la clase Cuenta, resaltando los dos constructores:

```
/**
 * Una cuenta bancaria, es decir, un registro en
 * una entidad bancaria que tiene un saldo (cantidad de dinero)
 * que se puede modificar mediante retiros o consignaciones.
 * @author Sandra V. Hurtado
 * @version 2.0
 */
public class Cuenta
{
    String numero;
    double saldo;
    String tipo;

    /**
     * Constructor de objetos Cuenta, con un saldo inicial.
     * @param numero el número de la cuenta
     * @param saldoInicial saldo inicial (en pesos) de la cuenta
     * @param tipo el tipo de cuenta: "ahorros" o "corriente"
     */
    Cuenta(String numero, double saldoInicial, String tipo)
    {
        this.numero = numero;
        this.saldo = saldoInicial;
        this.tipo = tipo;
    }

    /**
     * Constructor de objetos Cuenta, con saldo inicial cero.
     * @param numero el número de la cuenta
     * @param tipo el tipo de cuenta: "ahorros" o "corriente"
     */
    Cuenta(String numero, String tipo)
    {
        this.numero = numero;
        this.saldo = 0;
        this.tipo = tipo;
    }

    /**
     * Consulta el saldo actual de la cuenta (el dinero que tiene).
     * @return el valor del saldo, en pesos.
     */
    double getSaldo()
    {
        return saldo;
    }
}
```

*//continúa*



```

/**
 * Obtiene el tipo de cuenta (ahorros o corriente)
 * @return el tipo de cuenta: "ahorros" o "corriente".
 */
String getTipo()
{
    return tipo;
}

/**
 * Retira o saca una cantidad de dinero de la cuenta,
 * si hay saldo suficiente.
 * @param cantidad la cantidad que se desea retirar, en pesos.
 * @return si se pudo hacer el retiro porque tenía dinero
 *         suficiente o no (true o false)
 */
boolean retirar(double cantidad)
{
    if (saldo >= cantidad)
    {
        saldo = saldo - cantidad;
        return true;
    }
    else
    {
        return false;
    }
}

/**
 * Consigna o adiciona una cantidad de dinero en la cuenta,
 * lo cual incrementa el saldo.
 * @param cantidad la cantidad de dinero que se desea
 *                 consignar, en pesos
 */
void consignar(double cantidad)
{
    saldo = saldo + cantidad;
}
} // fin clase Cuenta

```

### **Ejercicio 10-3**

Escriba en Java el código para la siguiente clase, teniendo en cuenta que por lo general se crean mermeladas de fresa de 500 gramos, pero a veces se tienen mermeladas de otros sabores y en diferentes cantidades.

| Mermelada   |
|---|
| sabor: String<br>cantidad: double   |
| Mermelada()<br>Mermelada(sabor: String)<br>Mermelada(sabor: String, cantidad: double)<br>sacarMermelada(cantidadSacar: double): boolean |

## 10.6. Constructor por defecto

Incluso cuando no se tienen constructores en una clase, es posible crear objetos de esa clase (con *new*). Esto es posible porque Java, cuando una clase no tiene constructor, proporciona un **constructor por defecto**.

Este es un constructor que no recibe parámetros y que da unos valores iniciales a los atributos del objeto, así:

- Los atributos numéricos los inicializa en cero.
- Los atributos *boolean* los inicializa en *false*.
- Los atributos *char* los inicializa en el carácter nulo.
- Los atributos que sean de tipos referenciados los inicializa en *null*.

Sin embargo, si se escribe algún constructor en la clase, Java ya **no** proporciona el constructor por defecto y se debe usar el constructor que se escribió para crear los objetos.

Por ejemplo, se tienen las siguientes clases:

| Carro                              |
|------------------------------------|
| placa: String<br>modelo: int       |
| acelerar(): void<br>frenar(): void |



Para construir un objeto Carro se puede usar el constructor por defecto:

```
Carro carro1 = new Carro();
```

| Moto   |
|--|
| placa: String<br>modelo: int   |
| Moto(placa: String, modelo: int)<br>acelerar(): void<br>frenar(): void |

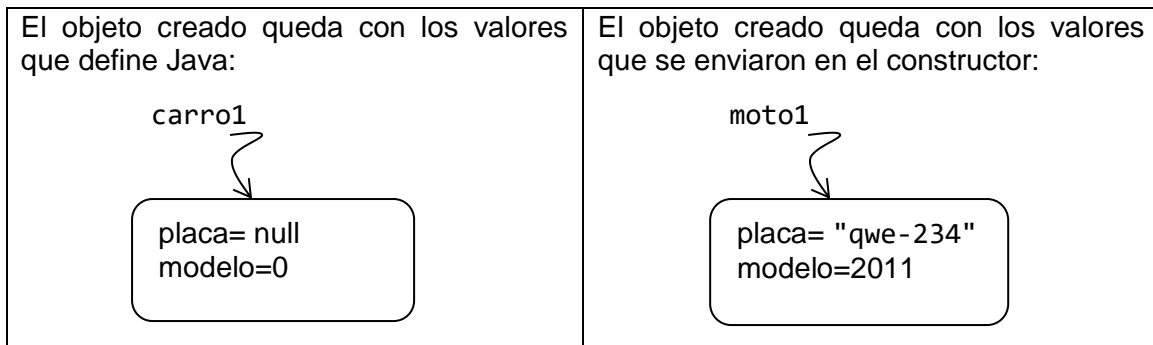


Para construir un objeto Moto **NO se puede usar el constructor por defecto**:

```
Moto moto1 = new Moto();
```

Se debe usar el constructor definido en la clase:

```
Moto moto1 = new Moto("qwe-234", 2011);
```



### **Ejercicio 10-4**

Se tiene la clase Mermelada:

| Mermelada   |
|---|
| sabor: String<br>cantidad: double   |
| Mermelada()<br>Mermelada(sabor: String)<br>Mermelada(sabor: String, cantidad: double)<br>sacarMermelada(cantidadSacar: double): boolean |

Indique si son correctas o no las siguientes instrucciones:

- a) Mermelada frasco1 = new Mermelada();
- b) Mermelada frasco2 = new Mermelada("fresa");
- c) Mermelada frasco3 = new Mermelada(String "mora");
- d) Mermelada frasco4 = new Mermelada(500);
- e) Mermelada frasco5 = new Mermelada("mora", 500);
- f) Mermelada frasco6 = new Mermelada(500, "fresa");

## **10.7. Ejercicio resuelto**

Hasta el momento se tiene el siguiente diagrama para la clase FrascoDeDulce:

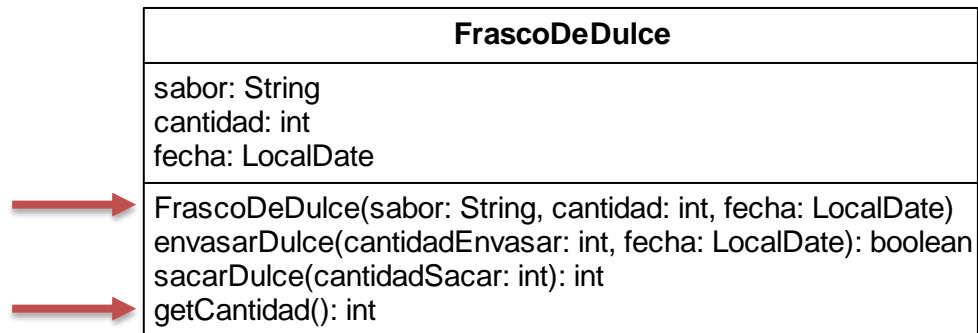
| FrascoDeDulce   |
|---|
| sabor: String<br>cantidad: int<br>fecha: LocalDate  |
| envasarDulce(cantidad: int, fecha: LocalDate): boolean<br>sacarDulce(cantidadSacar: int): int<br>consultarCantidad(): int |

Sin embargo, los frascos de dulce no se están creando con valores adecuados. Cada vez que se crea un objeto tiene los valores por defecto que proporcionaba Java, es decir: sabor

en *null*, cantidad en cero y fecha en *null*, lo cual no es lo deseado. Por lo tanto, se creará un método constructor que permita crear los frascos con la información que se necesita: el sabor, la cantidad y la fecha.

También se cambiará el método “consultarCantidad” por “getCantidad”, siguiendo los estándares definidos para este tipo de métodos que consultan el valor de un atributo.

El diagrama de esta clase, con el constructor y el cambio en el método de consulta, queda:



El código de la clase es:

```
import java.time.LocalDate;

/**
 * Un frasco con delicioso dulce, elaborado por la abuela pata.
 * Por ejemplo, un frasco de dulce de guayaba.
 * @author Sandra V. Hurtado
 * @version 2.5
 */
public class FrascoDeDulce
{
    String sabor;
    int cantidad;
    LocalDate fecha;

    /**
     * Constructor de nuevos objetos FrascoDeDulce.
     * @param sabor el sabor del dulce, por ejemplo, "brevas"
     * @param cantidad la cantidad, en gramos, del dulce en el frasco
     * @param fecha la fecha en la cual se elaboró el dulce
     */
    FrascoDeDulce(String sabor, int cantidad, LocalDate fecha)
    {
        this.sabor = sabor;
        this.cantidad = cantidad;
        this.fecha = fecha;
    }
}
```

*//continúa*

```

/**
 * Envasar o insertar dulce en el frasco, hasta máximo 200 gramos.
 * @param cantidadEnvasar la cantidad, en gramos, que se desea envasar
 * @param fecha          la fecha en la cual se envasa el dulce
 * @return               una indicación (true/false) de si se pudo envasar o no
 *                       toda la cantidad deseada o solo 200 gramos
 */
boolean envasarDulce(int cantidadEnvasar, LocalDate fecha)
{
    this.fecha = fecha;
    if (cantidadEnvasar > 200)
    {
        this.cantidad = 200;
        return false;
    }
    else
    {
        this.cantidad = cantidadEnvasar;
        return true;
    }
}

/**
 * Saca cierta cantidad del dulce del frasco, si hay suficiente
 * en el frasco; pero si hay menos solo se puede sacar lo que queda.
 * @param cantidadSacar  la cantidad, en gramos, que se desea sacar
 * @return               la cantidad de dulce que se pudo sacar, en gramos
 */
int sacarDulce(int cantidadSacar)
{
    int cantidadSacada = 0;
    if (this.cantidad >= cantidadSacar)
    {
        this.cantidad = this.cantidad - cantidadSacar;
        cantidadSacada = cantidadSacar;
    }
    else
    {
        cantidadSacada = this.cantidad;
        this.cantidad = 0;
    }
    return cantidadSacada;
}

/**
 * Consulta la cantidad de dulce del frasco
 * @return      la cantidad de dulce que hay en el frasco, en gramos
 */
int getCantidad()
{
    return this.cantidad;
}
} // fin clase FrascoDeDulce

```

## 11. *ArrayList* en Java

En este capítulo se presentará, de manera resumida, una clase del API de Java que es una colección que permite guardar varios objetos, facilitando así la elaboración de programas: se trata de la clase *ArrayList*.

A finalizar este capítulo usted debe ser capaz de:

- Crear un objeto de la clase *ArrayList*.
- Usar métodos básicos de *ArrayList* para adicionar u obtener objetos de esta colección.

### 11.1. Uso de colecciones

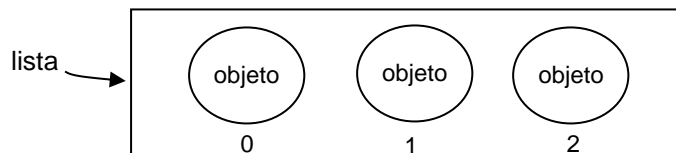
En un programa orientado a objetos lo más común es crear, no uno, sino varios objetos y realizar operaciones con ellos. En un programa muy sencillo se pueden tener variables que referencien cada objeto, pero si se tienen muchos objetos ya no es posible referenciarlos solo con variables, y se hace necesario agruparlos para facilitar su manejo.

Cuando se requiere trabajar con varios objetos por lo general se utiliza una **colección**, es decir, otro objeto que sirva como un contenedor para tener allí agrupados los objetos y así no tener que definir muchas variables que serían difíciles de manejar.

Las colecciones no solo permiten agrupar los objetos que se usarán en el programa, sino que también ofrecen algunos servicios adicionales para facilitar el manejo de estos objetos. En Java existen diferentes tipos de colecciones, que se pueden encontrar en el API, como los arreglos, las listas, los conjuntos y otras. En este capítulo se presentará una de estas colecciones, llamada *ArrayList*, por ser una de las más utilizadas.

### 11.2. Definición

Una lista es una colección donde los elementos están organizados uno detrás de otro (en secuencia), y se puede saber la posición de cada elemento en la lista. Las posiciones comienzan en cero.



No es necesario definir el tamaño de la lista cuando se crea, esta colección simplemente va creciendo a medida que se necesita, adicionando los objetos en la siguiente posición.

En Java se tienen dos tipos de listas: *ArrayList* y *LinkedList*. Ambas se encuentran en el paquete "java.util". En este capítulo se trabajará solo con *ArrayList*.

Cuando se crea un *ArrayList* es importante definir qué tipo de objetos se guardarán, como "personalizando" la colección. El tipo de objetos se define entre símbolos < y >. Por ejemplo, si se tienen la clase Cuadrado:

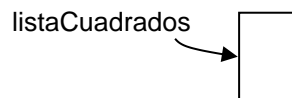
| Cuadrado   |
|--|
| lado: double                                     |
| Cuadrado(lado: double)<br>calcularÁrea(): double |

Para crear un *ArrayList* donde se guarden objetos Cuadrado, la instrucción es:

```
ArrayList<Cuadrado> listaCuadrados = new ArrayList<>();
```

Tipo de objetos que  
tendrá el ArrayList

De esta manera se crea la colección, que por el momento está vacía.



Como se sabe el tipo de objetos que se guardará en la colección, Java validará que no se adicionen, por error, objetos de otras clases. De esta manera se garantiza que los objetos se organicen apropiadamente.

### 11.3. Uso de *ArrayList*

Se presentarán algunos de los métodos que tiene la clase *ArrayList* para adicionar, buscar o eliminar objetos de la lista.

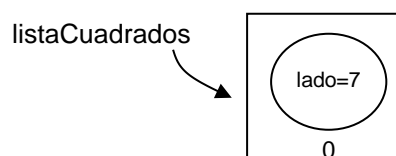
Para definir el tipo de objeto que contiene la colección se usará el nombre “Elemento”, pero este debe reemplazarse con el tipo de dato correspondiente. Por ejemplo, si la colección es de objetos Cuadrado, donde dice “Elemento” será realmente de tipo Cuadrado.

Los métodos más usados de *ArrayList* son:

- boolean add(Elemento objeto): Permite adicionar un objeto a la lista. El tipo de objeto que se adicione debe ser del mismo tipo definido cuando se creó el *ArrayList*. Este método retorna true cuando el objeto se pudo adicionar exitosamente a la colección.

Por ejemplo:

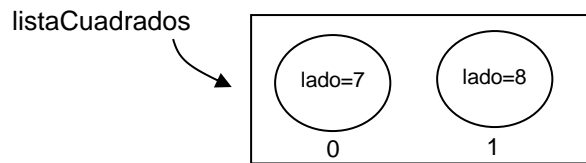
```
Cuadrado cuadro1 = new Cuadrado(7);  
boolean adiciona = listaCuadrados.add(cuadro1);
```



Cada vez que se usa el método “add” se adiciona el objeto en la siguiente posición de la lista (automáticamente).

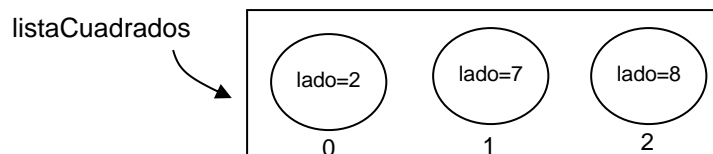
Por ejemplo:

```
Cuadrado cuadro2 = new Cuadrado(8);  
adiciona = listaCuadrados.add(cuadro2);
```



- void add(int índice, Elemento objeto): Adiciona un objeto en la posición o índice indicado, desplazando hacia la derecha todos los objetos posteriores a esta posición.  
Por ejemplo:

```
Cuadrado cuadro3 = new Cuadrado(2);  
listaCuadrados.add(0,cuadro3);
```



- boolean isEmpty(): Retorna true si no hay objetos en la colección.  
Por ejemplo:

```
if (listaCuadrados.isEmpty())  
{  
    System.out.println("La lista está vacía");  
}  
else  
{  
    System.out.println("La lista tiene objetos");  
}
```

Esto mostraría:

```
La lista tiene objetos
```

- int size(): Indica cuántos objetos hay actualmente en la colección.  
Por ejemplo:

```
int cantidadObjetos = listaCuadrados.size();  
System.out.println("En la lista hay "+cantidadObjetos+" objetos");
```

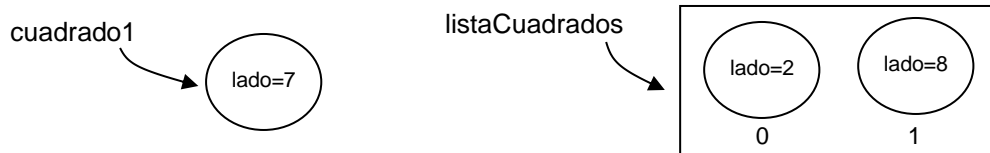
Esto mostraría:

```
En la lista hay 3 objetos
```

- Elemento remove(int índice): Retira o saca de la lista el objeto que hay en la posición dada, y retorna una referencia a este.  
Por ejemplo:

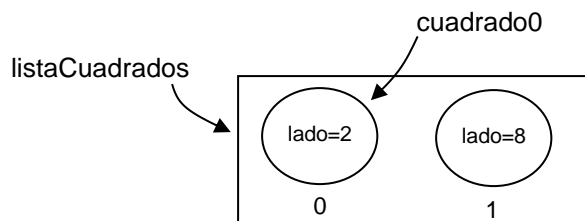


```
Cuadrado cuadrado1 = listaCuadrados.remove(1);
```



- Elemento `get(int índice)`: Retorna una referencia al objeto que se encuentra en la posición que se da en el parámetro, **sin sacarlo de la lista**.  
Por ejemplo:

```
Cuadrado cuadrado0 = listaCuadrados.get(0);
```



Al obtener la referencia al objeto se pueden usar sus métodos. Por ejemplo:

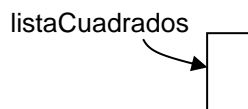
```
double areaCuadrado = cuadrado0.calcularArea();  
System.out.println("El área es: " + areaCuadrado);
```

Esto mostraría:

```
El área es: 4.0
```

- `clear()`: Retira todos los objetos de la lista, y la lista queda vacía.  
Por ejemplo:

```
listaCuadrados.clear();
```



### **Ejercicio 11-1**

Elabore un dibujo que represente cómo quedarían los objetos al ejecutar las siguientes instrucciones:

```

ArrayList<Cuadrado> losCuadrados = new ArrayList<>();
losCuadrados.add(new Cuadrado(3));
losCuadrados.add(new Cuadrado(5));
losCuadrados.add(new Cuadrado(3));
losCuadrados.add(new Cuadrado(10));
Cuadrado cuadradoUno = losCuadrados.get(2);
Cuadrado cuadradoDos = losCuadrados.remove(0);
losCuadrados.add(1,new Cuadrado(6));

```

### **Ejercicio 11-2**

Suponga que YA SE TIENE la clase Estudiante, que se muestra a continuación:

| Estudiante  |
|---|
| nombre: String                                    |
| Estudiante(nombre: String)<br>getNombre(): String |

Escriba las instrucciones en Java para la clase Materia:

| Materia                     |
|-----------------------------|
| inscribirEstudiantes(): int |

En el método “inscribirEstudiantes” debe crearse un *ArrayList* de estudiantes, y luego en un ciclo ir creando los estudiantes y adicionándolos a la lista, hasta que ya no se deseen inscribir más. Al final debe retornar cuántos estudiantes se inscribieron en la materia.

## **11.4. Ciclo mejorado para recorrer listas**

Una de las operaciones más comunes con las listas o colecciones de objetos, es recorrer o pasar por todos los objetos de la lista para realizar alguna operación. Por ejemplo, si se desea saber cuál es el área promedio de los cuadrados, hay que recorrer toda la lista de cuadrados para sumar las áreas de cada uno y luego dividir por la cantidad de cuadrados.

El recorrido de una lista se puede hacer con un ciclo *for* normal, de la siguiente forma:

```

double sumaAreas = 0;
for (int i=0; i < listaCuadrados.size();i++)
{
    Cuadrado cuadrado = listaCuadrados.get(i);
    sumaAreas += cuadrado.calcularArea();
}
double promedioAreas = sumaAreas / listaCuadrados.size();

```

Dos instrucciones para obtener cada elemento de la lista

En el ciclo anterior, es necesario usar el tamaño de la lista (método “size”) como límite para el ciclo, e internamente obtener cada objeto de la lista (con el método “get”), para luego realizar las operaciones.

Sin embargo, como este tipo de recorridos en una lista es tan frecuente, se tiene un ciclo especial, llamado “foreach”, que facilita el proceso.

Por ejemplo, el ciclo anterior queda de la siguiente forma:

```
double sumaAreas = 0;
for (Cuadrado cuadrado : listaCuadrados)
{
    sumaAreas += cuadrado.calcularArea();
}
double promedioAreas = sumaAreas / listaCuadrados.size();
```

Una sola instrucción para obtener cada elemento de la lista

El “foreach” tiene:

- La palabra *for* para indicar que es un ciclo,
- Una variable para los elementos de la colección, seguida de dos puntos, y
- El nombre de la lista

Con esto ya el programa sabe que debe recorrer toda la lista, e ir guardando cada objeto en la variable, para poder hacer las operaciones. De esta manera se simplifica el recorrido.

## 11.5. Ejercicio resuelto

En una entidad bancaria se tienen diferentes cuentas donde guardan el dinero los ahorradores. La clase correspondiente es:

| Cuenta  |
|---|
| número: String<br>saldo: double<br>tipo: String   |
| Cuenta(número: String, saldoInicial: double, tipo: String)<br>Cuenta(número: String, tipo: String)<br>getSaldo(): double<br>getTipo(): String<br>getNúmero(): String<br>consignar(cantidad: double): void<br>retirar(cantidad: double): boolean |

Se desea escribir una clase en Java que le permita al banco tener todas las cuentas, y donde se puedan adicionar nuevas cuentas y saber cuánto dinero en total tiene el banco – que es la suma de los saldos de todas las cuentas. Para adicionar una nueva cuenta al banco se debe enviar el número, el saldo y el tipo; y antes de crear la cuenta se debe verificar que no exista ya una cuenta con ese mismo número.

El diagrama para la clase Banco es:

| Banco   |
|---|
| cuentas: ArrayList<Cuenta>  |
| Banco()<br>buscarCuenta(número: String): Cuenta<br>adicionarCuenta(número: String, saldoInicial: double, tipo: String): boolean<br>consultarTotalDinero(): double |

El código para esta clase es:

```
import java.util.ArrayList;

/**
 * Entidad bancaria que maneja varias cuentas con dinero
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Banco
{
    ArrayList<Cuenta> cuentas;

    /**
     * Constructor de objetos Banco
     */
    Banco ()
    {
        cuentas = new ArrayList<>();
    }

    /**
     * Buscar una cuenta en la lista, a partir del número.
     * @param numero el número que identifica la cuenta que se buscará
     * @return el objeto Cuenta que corresponde al número dado,
     *         o null si no se encuentra.
     */
    Cuenta buscarCuenta(String numero)
    {

```

*//continúa*

```

        for (Cuenta cuentaComparar : cuentas)
        {
            if (cuentaComparar.getNumero().equals(numero))
            {
                return cuentaComparar;
            }
        }
        return null;
    }

    /**
     * Se crea una nueva cuenta y se adiciona a la lista.
     * @param numero el número de la nueva cuenta - debe ser único
     * @param saldoInicial el saldo inicial de dinero en la nueva cuenta
     * @param tipo si es una cuenta corriente o de ahorros
     * @return true si se pudo crear y adicionar la cuenta,
     *         y false en caso contrario
     */
    boolean adicionarCuenta(String numero, double saldoInicial,
                             String tipo)
    {
        Cuenta cuentaExistente = buscarCuenta(numero);
        if (cuentaExistente == null)
        {
            Cuenta nuevaCuenta =
                new Cuenta(numero, saldoInicial, tipo);
            return cuentas.add(nuevaCuenta);
        }
        else
        {
            return false;
        }
    }

    /**
     * Consultar el total de dinero en todas las cuentas del banco
     * @return el total de dinero en el banco
     */
    double consultarTotalDinero()
    {
        double cantidadDinero = 0;
        for (Cuenta cuenta : cuentas)
        {
            cantidadDinero += cuenta.getSaldo();
        }
        return cantidadDinero;
    }
} // fin clase Banco

```

## 12. Asociaciones entre Clases

En este capítulo se mostrará la forma de representar en un diagrama de clases un tipo de relaciones que se presenta entre los objetos/clases; y también cómo estas relaciones se llevan a código Java como atributos de tipo referenciado.

A finalizar este capítulo usted debe ser capaz de:

- Representar una asociación entre objetos/clases en un diagrama de clases.
- Implementar en código Java, mediante atributos, las asociaciones mostradas en un diagrama de clases.

### 12.1. Contexto y definición

Para elaborar un programa usando el paradigma de orientación a objetos, los pasos generales son:

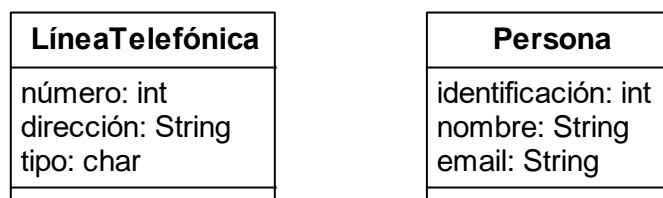
- a. Identificar los tipos de objetos (y de manera más general, las clases) que son parte del sistema y que son importantes para el desarrollo del programa.
- b. Representar las clases en un diagrama, incluyendo sus atributos y métodos.
- c. **Adicionar las relaciones** entre clases.
- d. Definir las pruebas.
- e. Elaborar el código Java correspondiente al diagrama.
- f. Ejecutar las pruebas y realizar las correcciones necesarias.

En este capítulo se profundizará en el paso: “Adicionar las relaciones entre clases”, que permite no solo elaborar un diagrama más completo, sino también programas cada vez más grandes.

Por ejemplo, se tiene el siguiente enunciado:

“Una empresa de comunicaciones tiene diferentes líneas telefónicas. Cada línea telefónica está asociada a una dirección, y puede corresponder a una línea residencial o comercial. Además, cada línea está a nombre de una persona - el usuario - de quien se debe saber su documento de identificación, su nombre completo y el correo electrónico (email) de contacto”.

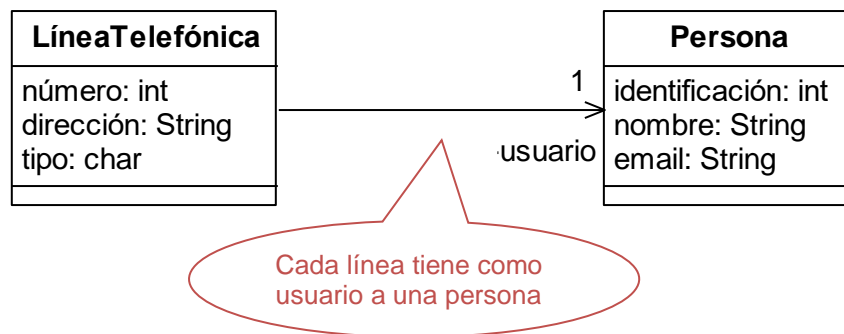
En el anterior enunciado se identifican dos clases, que se representan de la siguiente forma (se omiten los métodos para simplificar el diagrama):



Sin embargo, en el diagrama anterior falta alguna información, porque es importante saber, para cada línea telefónica, cuál es el usuario, es decir, a nombre de cual persona está. Esto se representa con una **asociación** entre las clases.

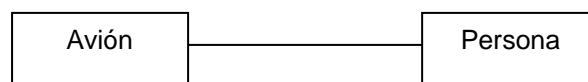
Una asociación muestra cuando objetos de una clase se relacionan con objetos de otra clase.

En el ejemplo, cada línea telefónica está a nombre de una persona, y por lo tanto se tiene una relación entre estas clases. Para representar esta relación en el diagrama se dibuja una línea continua que une las dos clases, como se muestra a continuación:

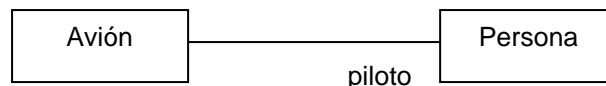


## 12.2. Representación

Una asociación se representa como una línea entre dos clases. Por ejemplo, para decir que los aviones tienen información del piloto, que es una persona:

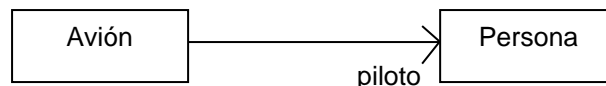


Para dar mayor claridad se puede escribir cómo participan los objetos en la relación. Esto se conoce como el papel (o rol). Por ejemplo, en la relación entre Avión y Persona se desea saber cuál persona es el piloto del avión. Por lo tanto, el rol es "piloto", y se representa de la siguiente forma:



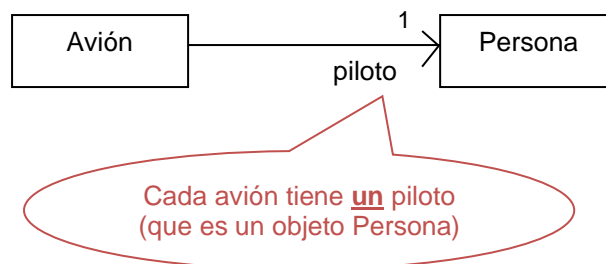
Para mostrar que la relación va de Avión a Piloto, es decir, que es el avión el que tendrá la información de la persona (el piloto), se adiciona una flecha.

Ejemplo:

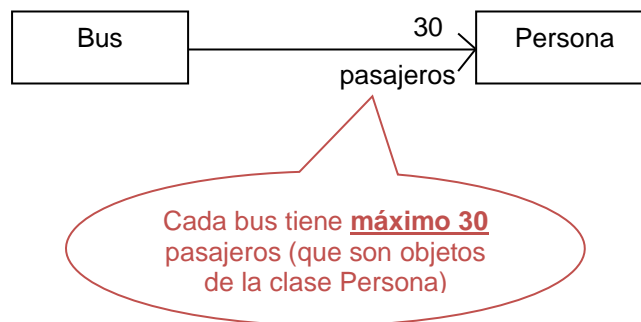


Por último, para mostrar con cuántos objetos se relaciona se escribe el número al lado de la flecha.

Ejemplo:



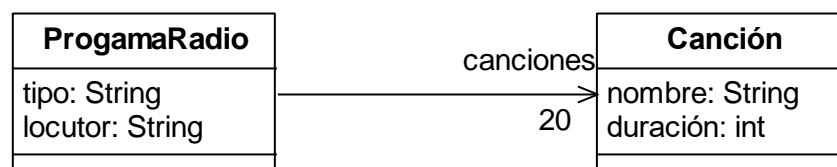
En el caso de tener que la relación sea con varios objetos:



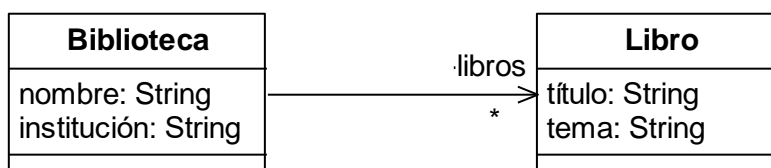
En otro ejemplo:

“Una estación de radio tiene diferentes programas durante el día. Cada programa, que puede ser musical, de concurso, de opinión o variado, tiene un locutor principal. En cada programa se transmiten diferentes canciones. Hay un límite de 20 canciones que se puedan transmitir en un programa. De cada canción se debe indicar el nombre y la duración”.

En este caso el diagrama de clases, donde solo se muestran los atributos por simplicidad, es:



Cuando no se conoce la cantidad máxima de objetos que participan en una relación, se usa el asterisco en lugar del número. Por ejemplo, para mostrar que una biblioteca puede tener muchos libros (sin un máximo definido):



### **Ejercicio 12-1**

Elabore el diagrama de clases para el siguiente enunciado:

En el edificio “Los Pastos” tienen 4 locales comerciales con almacenes variados para servicio al público. Se tienen dos locales estándar de 5 metros cuadrados, uno más pequeño con 2 metros cuadrados (pero con vitrina hacia la calle) y el local más grande con 6 metros cuadrados. Debido al éxito que han tenido con los locales comerciales, los administradores del edificio están planeando construir otro local –pequeño– para ampliar su oferta.

## **12.3. Implementación en Java**

Cuando se tienen asociaciones en un diagrama de clases, éstas se implementan como atributos en las clases, de la siguiente forma:



- Cuando la cantidad es UNO, se convierte en **un atributo de tipo de la clase** con la cual se tiene la relación.
- Cuando la cantidad es MAYOR a UNO, se convierte en **una colección de objetos** de la clase con la cual se tiene la relación. Por ejemplo, en un *ArrayList*.

- **Relaciones UNO:**

Por ejemplo, para el siguiente diagrama:



En la clase Persona no se tendr n atributos adicionales. El c digo para la clase Persona es:

```

/**
 * Informaci n b sica de una Persona, habitante de una ciudad
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Persona
{
    int identificacion;
    String nombre;
    String direccion;
}
  
```

En la clase L neaTel f nica se debe tener un atributo para implementar la asociaci n, y as  saber la persona que tiene registrada la l nea. Este atributo ser  de tipo Persona y se llamar  "usuario", que es el nombre del rol en la asociaci n.

El c digo para la clase L neaTel f nica es:

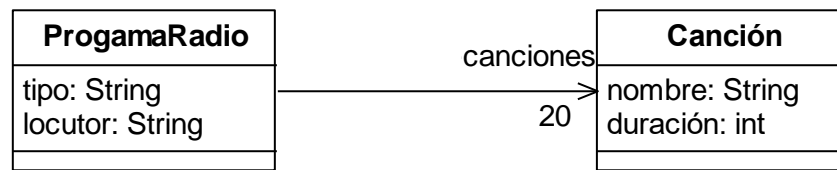
```

/**
 * Representa una l nea de tel fono fijo en una ciudad, que est 
 * registrada para una persona (el usuario)
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class LineaTelefonica
{
    int numero;
    String direccion;
    String tipo;
    Persona usuario;
}
  
```

Atributo "usuario", de tipo **Persona**, para la asociaci n

- **Relaciones MAYOR A UNO:**

Para el caso de los programas de radio:



El código de la clase Canción es:

```

/**
 * Información de música que se transmite en radio
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Cancion
{
    String nombre;
    int duracion;
}
  
```

En la clase ProgramaRadio se creará un atributo de tipo *ArrayList*, porque se necesita una colección debido a que la relación es de varios objetos. El *ArrayList* tendrá los diferentes objetos de la clase canción (máximo 20 para este ejemplo). Este atributo se llamará “canciones”, que es el nombre del rol en el diagrama de clases.

El código de la clase ProgramaRadio es:

```

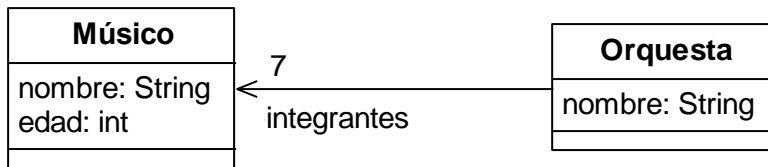
import java.util.ArrayList;

/**
 * Programa de radio que transmite canciones
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class ProgramaRadio
{
    String tipo;
    String locutor;
    ArrayList<Cancion> canciones;
}
  
```

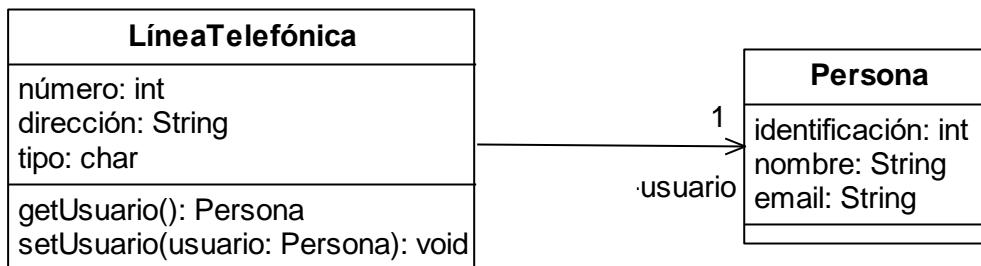
Atributo “canciones”, de tipo **ArrayList** de Canción

### **Ejercicio 12-2**

Escriba el código Java para el siguiente diagrama de clases:



Es importante tener en cuenta que las clases que tienen asociaciones cuentan con un atributo adicional y por lo tanto en los métodos se puede usar este atributo. Por ejemplo, como en la clase LíneaTelefónica se tiene el atributo “usuario”, de tipo Persona, se pueden tener métodos para definir y consultar este atributo, así:



El código para la clase LíneaTelefónica queda:

```

/**
 * Representa una línea de teléfono fijo en una ciudad, que está
 * registrada para una persona (el usuario)
 * @author Sandra V. Hurtado
 * @version 1.2
 */
public class LineaTelefonica
{
    int numero;
    String direccion;
    String tipo;
    Persona usuario;

    /**
     * Consultar el usuario de la línea
     * @return el usuario de la línea - un objeto Persona
     */
    Persona getUsuario()
    {
        return usuario;
    }
}
  
```

*//continúa*

```

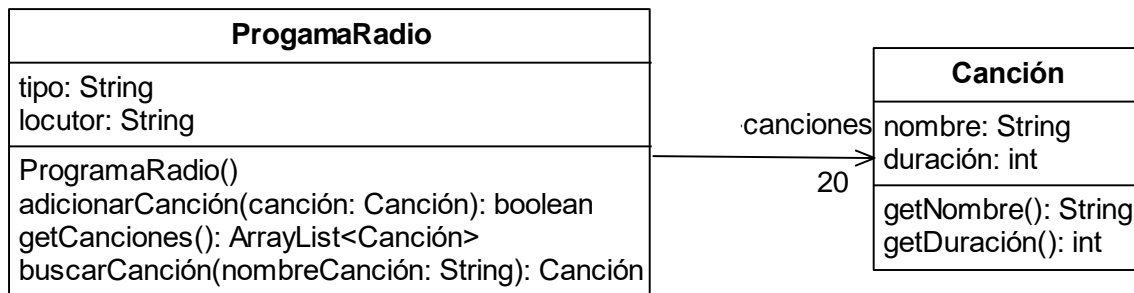
    /**
     * Definir o cambiar el usuario de la línea
     * @param usuario    el nuevo usuario de la línea
     */
    void setUsuario(Persona usuario)
    {
        this.usuario = usuario;
    }
} // fin clase LineaTelefonica

```

En el ejemplo de los programas de radio, cada programa puede tener varias canciones (máximo 20), pero no se adicionan todas a la vez, sino una por una. Por lo tanto, lo que se tiene es un método para adicionar una canción a la lista.

En cuanto a consultar: es posible consultar toda la lista de canciones del programa o consultar solo una canción por su nombre, lo cual lleva a tener dos métodos: “getCanciones” y “buscarCanción”.

El diagrama correspondiente es:



El código para la clase ProgramaRadio queda:

```

import java.util.ArrayList;

/**
 * Programa de radio que transmite canciones
 * @author Sandra V. Hurtado
 * @version 2.0
 */
public class ProgramaRadio
{
    String tipo;
    String locutor;
    ArrayList<Cancion> canciones;

    //continúa

```

```

/**
 * Constructor de programas de radio, con valores por defecto.
 */
ProgramaRadio()
{
    this.tipo= "musical";
    this.locutor = "automático";
    this.canciones = new ArrayList<>();
}

/**
 * Permite adicionar una canción a la lista del programa,
 * hasta un máximo de 20
 * @param cancion    La canción que se desea adicionar
 * @return true si se pudo adicionar, false si no se pudo
 *         adicionar porque ya estaban las 20 canciones
 */
boolean adicionarCancion(Cancion cancion)
{
    if (canciones.size() < 20)
    {
        return canciones.add(cancion);
    }
    return false;
}

/**
 * Consultar las canciones del programa
 * @return la lista de canciones del programa
 */
ArrayList<Cancion> getCanciones()
{
    return canciones;
}

/**
 * Buscar una canción en las canciones del programa
 * @param nombre    el nombre de la canción a buscar
 * @return el objeto Canción con ese nombre, si está en el
 *         programa, o null si no está.
 */
Cancion buscarCancion(String nombre)
{
    for (Cancion cancion : canciones)
    {
        if (cancion.getNombre().equals(nombre))
        {
            return cancion;
        }
    }
    return null;
}
} // fin clase ProgramaRadio

```

### **Ejercicio 12-3**

Elabore el diagrama de clases y escriba el código Java para el siguiente enunciado.

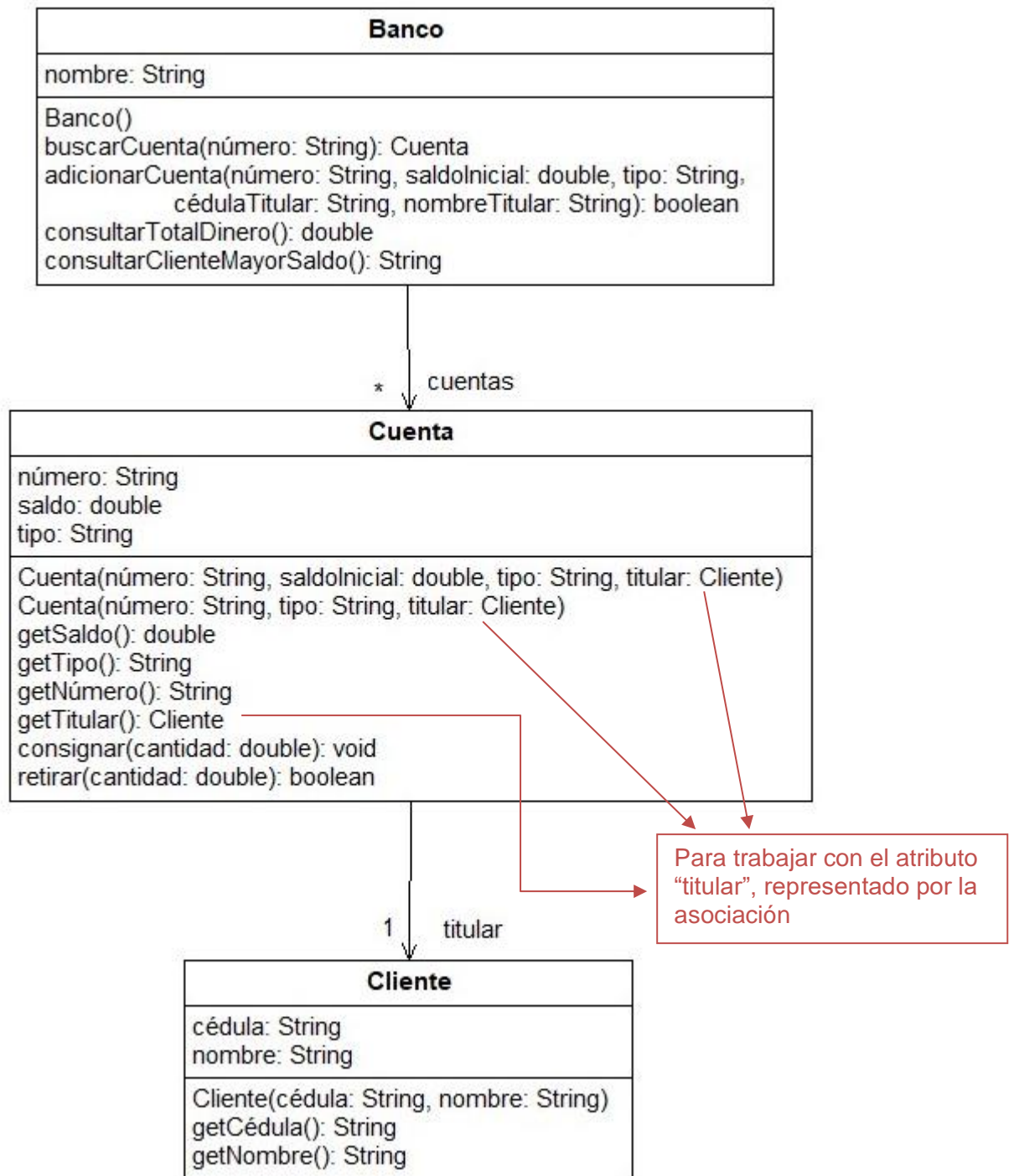
La Oficina de Detectives “El Destape” recibe varios casos cada mes, donde deben seguir las pistas para encontrar al culpable. Como son muy organizados, en El Destape asignan un número único a cada caso, y también asignan a uno de los detectives como el responsable. Cada caso también recibe un nombre clave para poder usarlo en las conversaciones. Por ejemplo, el caso número 8767 es llamado “Nube azul” y está a cargo del detective Scoba Du, quien lleva 5 años trabajando en El Destape. Este caso se trata de una sospecha de falsificación de una firma en un cheque, y el detective está conduciendo la investigación para determinar quién pudo realizar esto.

En cada caso por lo general se tiene un conjunto de sospechosos de los cuales hay que guardar información. De cada sospechoso se guarda su nombre, el alias (si lo tiene), la edad, la última dirección conocida, la foto y una descripción de sus características físicas. Algunas veces los sospechosos de un caso pueden ser sospechosos en otros casos. Por esta razón, cuando se encuentra un sospechoso, primero se debe buscar por el nombre, para ver si ya está registrado (y así no repetir sus datos).

### **12.4. Ejercicio resuelto**

El banco MuchoDinero tiene varias cuentas (de ahorro y corrientes), cada una con cierta cantidad de dinero, que es el saldo, y con un número que la identifica. El banco también debe saber quién es el titular (es decir, el dueño) de cada cuenta, dado que éste es un cliente del banco. Al crear (adicionar) una cuenta, se pide la cédula y el nombre completo del titular. Al banco le interesa saber cuánto dinero en total tiene, y cuál es el nombre del cliente que tiene la cuenta con el mayor saldo.

En este caso el diagrama de clases es:



Se presentará el código para estas clases, donde se seguirá la convención de no colocar comentarios en los métodos "get", pues ya se conoce su funcionalidad: consultar el valor de un atributo. Tampoco se incluirán comentarios en los constructores por defecto, solo en aquellos que reciban parámetros. Este tipo de convenciones no son generales, sino que deben ser establecidas por cada grupo.

```

/**
 * Persona que posee una cuenta en un banco y, por lo tanto,
 * es cliente del banco
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Cliente
{
    String cedula;
    String nombre;

    /**
     * Constructor de un objeto cliente
     * @param cedula el número de cédula que identifica al cliente
     * @param nombre el nombre completo del cliente
     */
    Cliente(String cedula, String nombre)
    {
        this.cedula = cedula;
        this.nombre = nombre;
    }

    String getCedula()
    {
        return cedula;
    }

    String getNombre()
    {
        return nombre;
    }
} // fin clase Cliente

```

```

/**
 * Una cuenta bancaria, es decir, un registro en
 * una entidad bancaria que tiene un saldo (cantidad de dinero)
 * que se puede modificar mediante retiros o consignaciones.
 * @author Sandra V. Hurtado
 * @version 2.6
 */
public class Cuenta
{
    String numero;
    double saldo;
    String tipo;
    Cliente titular;

```

Atributo representado con la asociación

//continúa



```

/**
 * Constructor de objetos Cuenta, con un saldo inicial.
 * @param numero el número de la cuenta
 * @param saldoInicial saldo inicial (en pesos) de la cuenta
 * @param tipo el tipo de cuenta: "ahorros" o "corriente"
 * @param titular el dueño de la cuenta - un objeto Cliente
 */
Cuenta(String numero, double saldoInicial, String tipo,
        Cliente titular)
{
    this.numero = numero;
    this.saldo = saldoInicial;
    this.tipo = tipo;
    this.titular = titular;
}

/**
 * Constructor de objetos Cuenta, con saldo inicial cero.
 * @param numero el número de la cuenta
 * @param tipo el tipo de cuenta: "ahorros" o "corriente"
 * @param titular el dueño de la cuenta - un objeto Cliente
 */
Cuenta(String numero, String tipo, Cliente titular)
{
    this.numero = numero;
    this.saldo = 0;
    this.tipo = tipo;
    this.titular = titular;
}

double getSaldo()
{
    return this.saldo;
}

String getTipo()
{
    return this.tipo;
}

String getNumero()
{
    return this.numero;
}

Cliente getTitular()
{
    return this.titular;
}

```

*//continúa*

```

/**
 * Retira o saca una cantidad de dinero de la cuenta,
 * si hay saldo suficiente.
 * @param cantidad la cantidad que se desea retirar, en pesos.
 * @return si se pudo hacer el retiro porque tenía dinero
 * suficiente o no (true o false)
 */
boolean retirar(double cantidad)
{
    if (saldo >= cantidad)
    {
        saldo = saldo - cantidad;
        return true;
    }
    else
    {
        return false;
    }
}

/**
 * Consigna o adiciona una cantidad de dinero en la cuenta,
 * lo cual incrementa el saldo.
 * @param cantidad la cantidad de dinero que se desea
 * consignar, en pesos
 */
void consignar(double cantidad)
{
    saldo = saldo + cantidad;
}
} // fin clase Cuenta

```

```
import java.util.ArrayList;
```

```

/**
 * Entidad bancaria que maneja varias cuentas con dinero
 * @author Sandra V. Hurtado
 * @version 2.0
 */
public class Banco
{
    String nombre;
    ArrayList<Cuenta> cuentas;

```

Atributo representado con la asociación

//continúa

```

Banco ()
{
    this.nombre = "MuchoDinero";
    this.cuentas = new ArrayList<>();
}

/**
 * Buscar una cuenta en la lista, a partir del número.
 * @param numero el número que identifica la cuenta que se buscará
 * @return el objeto Cuenta que corresponde al número dado,
 *         o null si no se encuentra.
 */
Cuenta buscarCuenta(String numero)
{
    for (Cuenta cuentaComparar : cuentas)
    {
        if (cuentaComparar.getNumero().equals(numero))
        {
            return cuentaComparar;
        }
    }
    return null;
}

/**
 * Se crea una nueva cuenta y se adiciona a la lista.
 * @param numero el número de la nueva cuenta - debe ser único
 * @param saldoInicial el saldo inicial de dinero en la nueva cuenta
 * @param tipo si es una cuenta corriente o de ahorros
 * @param cedulaTitular la cédula del titular de la cuenta
 * @param nombreTitular el nombre completo del titular de la cuenta
 * @return true si se pudo crear y adicionar la cuenta,
 *         y false en caso contrario
 */
boolean adicionarCuenta(String numero, double saldoInicial,
                        String tipo, String cedulaTitular, String nombreTitular)
{
    Cuenta cuentaExistente = buscarCuenta(numero);
    if (cuentaExistente == null)
    {
        // Se crea el nuevo cliente
        Cliente dueno = new Cliente(cedulaTitular,nombreTitular);

        // Se crea la cuenta con el cliente
        Cuenta nuevaCuenta =
            new Cuenta(numero, saldoInicial, tipo, dueno);
        return cuentas.add(nuevaCuenta);
    }
    return false;
}

```

*//continúa*

```

/**
 * Consultar el total de dinero en todas las cuentas del banco
 * @return el total de dinero en el banco
 */
double consultarTotalDinero()
{
    double cantidadDinero = 0;
    for (Cuenta cuenta : cuentas)
    {
        cantidadDinero += cuenta.getSaldo();
    }
    return cantidadDinero;
}

/**
 * Obtener el nombre del cliente con mayor saldo en el banco
 * @return el nombre del cliente con más saldo,
 *         o null si todavía no hay cuentas creadas.
 */
String consultarClienteMayorSaldo()
{
    Cuenta cuentaMayor = null;
    double saldoMayor = 0;
    for (Cuenta cuenta : cuentas)
    {
        if (cuenta.getSaldo() > saldoMayor)
        {
            saldoMayor = cuenta.getSaldo();
            cuentaMayor = cuenta;
        }
    }
    if (cuentaMayor != null)
    {
        Cliente clienteMayorSaldo = cuentaMayor.getTitular();
        return clienteMayorSaldo.getNombre();
    }
    return null;
}
} // fin clase Banco

```

## 13. Paquetes y Visibilidad

En este capítulo se presentará el concepto de paquetes en Java, y cómo pueden ser utilizados para establecer nombres únicos de clases y para definir permisos o visibilidad sobre los miembros de una clase.

A finalizar este capítulo usted debe ser capaz de:

- Representar paquetes en un diagrama de clases.
- Usar la palabra reservada *package* del lenguaje Java, para definir en cuál paquete se encuentra una clase.
- Diferenciar los cuatro tipos de visibilidad que se tienen en Java.
- Escribir en Java modificadores de visibilidad para los miembros de una clase.

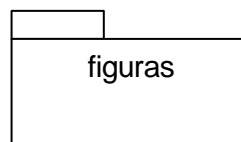
### 13.1. Definición y representación

A medida que los programas en Java van incluyendo más clases se tiene la necesidad de organizar esas clases para facilitar su uso. Esto se hace de manera similar a como se organizan los archivos en un computador: creando “carpetas” para guardar cada archivo en la carpeta que le corresponda. En Java, el equivalente a estas carpetas son los paquetes.

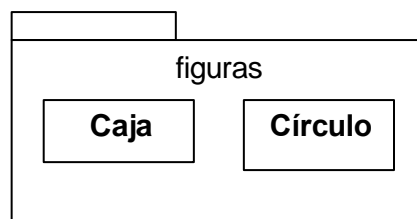
Un **paquete** es un mecanismo para agrupar las clases, de manera similar a como una carpeta guarda varios archivos. Por lo general, las clases que se agrupan en un paquete tienen funciones similares o relacionadas.

Por convención los nombres de los paquetes solo deben contener letras minúsculas, incluso si están formados por varias palabras. Por ejemplo, un nombre adecuado para un paquete sería “mipaquete”, y no “miPaquete”. Además, se recomienda que no tenga números al inicio y que no incluya caracteres extraños.

En los diagramas de clases un paquete se representa con un rectángulo que tiene un recuadro de menor tamaño en la parte superior izquierda, así:



Las clases se muestran dentro del paquete, por ejemplo:



Además de servir para organizar las clases, los paquetes ayudan a:

- Garantizar nombres únicos para las clases.
- Definir permisos para que otras clases puedan usar (o no) atributos y métodos de una clase.

## 13.2. Nombres únicos

En Java cada clase debe tener un nombre único, sin embargo, esto es difícil de lograr cuando los programas crecen, cuando hay varios desarrolladores o cuando se usan clases de otros proyectos. La solución para que se puedan tener clases con el mismo nombre es usar los paquetes. Los paquetes permiten asignar nombres únicos a las clases, porque es posible tener clases que se llamen igual, siempre y cuando estén en paquetes diferentes. Por este motivo algunas veces los paquetes se asocian con la idea de un “espacio de nombres”.

Para garantizar que cada elemento de un programa en Java sea fácilmente identificable, se define que **el nombre completo** de las clases y paquetes está compuesto por el nombre del paquete al que pertenece, seguido de un punto y luego su nombre. De esta forma se garantizan nombres únicos.

Por ejemplo, y haciendo referencia al diagrama de clases anterior:

- El nombre completo de la clase Caja es: “figuras.Caja”.
- El nombre completo de la clase Círculo es: “figuras.Círculo”.

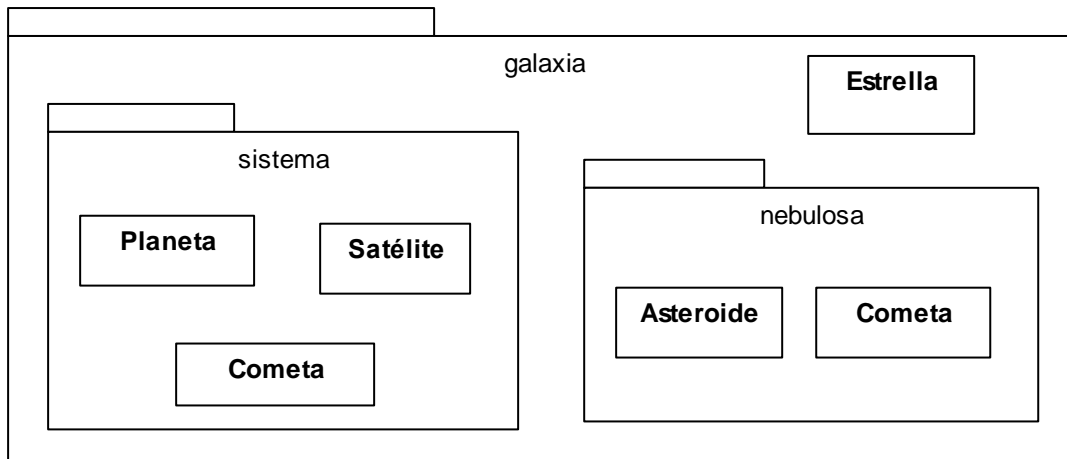
En un programa se puede usar este nombre completo para referirse a la clase. Por ejemplo:

```
/**
 * Muestra el uso del nombre completo de una clase
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class UsoNombreCompletoClase
{
    /**
     * Crea un objeto Caja, usando el nombre completo de la clase.
     */
    void crearObjeto()
    {
        figuras.Caja caja = new figuras.Caja();
    }
}
```

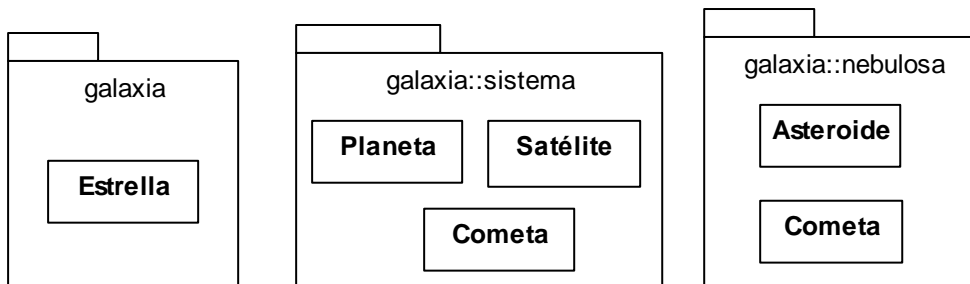
Sin embargo, usar los nombres completos puede ser algo tedioso. Generalmente usamos el nombre simple de las clases para facilitar la escritura de los programas, lo cual es posible gracias al mecanismo de “importar” que tiene Java, el cual se explicará más adelante.

Java permite tener paquetes dentro de paquetes, pero esto solo influye en el nombre. Es decir, aunque se tengan paquetes dentro de otro, **cada uno de estos paquetes internos se comporta como un paquete independiente**.

Por ejemplo, el siguiente diagrama muestra un paquete formado por una clase y dos subpaquetes, cada uno de estos con otras clases:



Sin embargo, para Java, esto es equivalente a tener tres paquetes independientes, así:



Con respecto a los nombres de algunas clases en la figura anterior:

- El nombre completo de Estrella es: “galaxia.Estrella”
- El nombre completo de Planeta es: “galaxia.sistema.Planeta”
- Se tienen dos clases llamadas Cometa, pero son diferentes, porque una está en el paquete “galaxia.sistema” y otra está en el paquete “galaxia.nebulosa”. Los nombres completos son: “galaxia.sistema.Cometa”, y “galaxia.nebulosa.Cometa”.

### 13.3. Paquetes en Java

Si se quiere indicar que una clase pertenece a un paquete se debe utilizar la palabra reservada **package** seguida del nombre del paquete. Esta debe ser la primera instrucción en la clase, antes de cualquier otra instrucción, así:

```

package nombrepaquete;

/**
 * Comentarios de la clase
 */
public class Clase
{
    // código clase
}
  
```

Por ejemplo, para la clase Estrella del diagrama de clases anterior:

```
package galaxia;

/**
 * Clase que está en un paquete externo
 */
public class Estrella
{
    //código clase
}
```

Y otro ejemplo, para la clase Planeta:

```
package galaxia.sistema;

/**
 * Clase que está en un paquete interno
 */
public class Planeta
{
    //código clase
}
```

### 13.4. Uso de clases de otros paquetes (importar)

Cuando las clases están en el mismo paquete no es necesario dar el nombre completo para usarlas; pero cuando están en otro paquete sí es necesario usar una de dos formas para que Java las pueda encontrar:

- Dar su nombre completo: “paquete.clase”, cada vez que se haga referencia a ella, o
- Utilizar la sentencia *import*.

Cuando se incluye la sentencia *import*, se está indicando al programa que debe buscar en otro paquete. La sentencia *import* tiene la siguiente forma:

```
import nombre;
```

El nombre se puede dar de dos maneras:

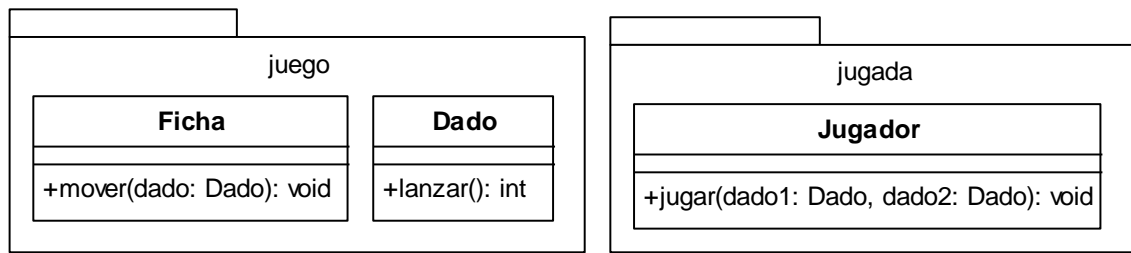
- Dar el nombre completo de la clase -para llegar directamente a ella, o
- Dar el nombre del paquete y luego un asterisco, para que busque en todas las clases del paquete.

Las sentencias *import* van después de la sentencia *package*, y antes del encabezado de la clase.

Si es necesario incluir varias clases o varios paquetes se debe usar **un *import* para cada uno**, ya que esta sentencia no acepta varios elementos en una sola línea.



Por ejemplo, se tiene el siguiente diagrama:



Como puede verse, se necesita usar objetos Dado, tanto en la clase Ficha como en la clase Juego.

En la clase Ficha se puede usar directamente la clase Dado, sin usar un *import*, porque están en el mismo paquete. El código es:

```
package juego;
/**
 * Una ficha de un juego, que usa un dado para avanzar.
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Ficha
{
    /**
     * Mover una ficha, el número de veces que saque el dado
     * @param unDado el dado para obtener el número
     */
    public void mover(Dado unDado)
    {
        int casillas = unDado.lanzar();
        System.out.println("La ficha se mueve "+casillas+ " casillas");
    }
}
```

Uso de clase en el mismo paquete (sin *import*)

Sin embargo, en la clase Juego se debe usar un *import* para poder usar la clase Dado porque están en diferentes paquetes. El código es:

```
package jugada;

import juego.Dado;

/**
 * Un jugador que practica con dos dados (clase que está en otro paquete)
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Jugador
{
```

//continúa

```

/**
 * Juego que compara los números que salen en dos dados
 * @param dado1 el primer dado participante
 * @param dado2 el segundo dado participante
 */
public void jugar(Dado dado1, Dado dado2)
{
    int valor1 = dado1.lanzar();
    int valor2 = dado2.lanzar();

    if (valor1 > valor2)
    {
        System.out.println("Ganó el primer dado");
    }
    else if (valor2 > valor1)
    {
        System.out.println("Ganó el segundo dado");
    }
    else
    {
        System.out.println("Empataron");
    }
}
} // fin clase Jugador

```

### **Ejercicio 13-1**

Se tiene la siguiente definición de clases:

```

package equipos;
/**
 * Equipo de comunicación entre personas.
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Celular
{
    String numero = "123";

    /** Consultar el número del celular.
     * @return el número del celular
     */
    public String getNumero()
    {
        return numero;
    }
} // fin clase Celular

```

```

package medios;
/**
 * Representa una conversación textual ("chat") en un celular.
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Chat
{
    /** Método para mostrar el número de celular del chat.
     */
    void mostrarNumero ()
    {
        Celular celular = new Celular();
        System.out.println("El número es: " + celular.getNumero());
    }
}

```

- a) ¿Qué hace falta en la clase Chat para que pueda usar la clase Celular?
- b) Elabore el diagrama de clases correspondiente.

## 13.5. Visibilidad

Agrupar las clases en paquetes, no sólo permite una mejor organización, sino que también tiene efectos sobre el acceso a las clases y los miembros de una clase (atributos y métodos).

En Java hay cuatro niveles o tipos de visibilidad:

- **Público:** los miembros definidos como públicos pueden ser usados por TODAS las demás clases del programa.

La palabra en Java para este nivel de visibilidad es: *public*

Por ejemplo:

```
public void unMetodo()    // un método público
```

- **Protegido:** los miembros protegidos pueden ser usados por todas las clases que **estén en el mismo paquete y las clases hijas**<sup>1</sup>.

La palabra en Java para este nivel de visibilidad es: *protected*

Por ejemplo:

```
protected void unMetodo()    // un método protegido
```

- **De paquete, amigable u omitido:** los miembros con visibilidad de paquete solo pueden ser usados por otras clases que **estén en el mismo paquete**. No hay una palabra en Java para este nivel de visibilidad. Cuando no se coloca ningún nivel de visibilidad, Java identifica ese elemento como amigable o de paquete. Como no se tiene ningún nombre, a este nivel de visibilidad en Java también se le llama "omitido". Por ejemplo:

<sup>1</sup> El concepto de clase hija se explicará en un capítulo posterior.

```
void unMetodo() // un método amigable o de paquete
```

- **Privado:** Solo se pueden usar dentro de la clase donde están definidos. **No** pueden ser usados por otras clases.

La palabra en Java para este nivel de visibilidad es: *private*

Por ejemplo:

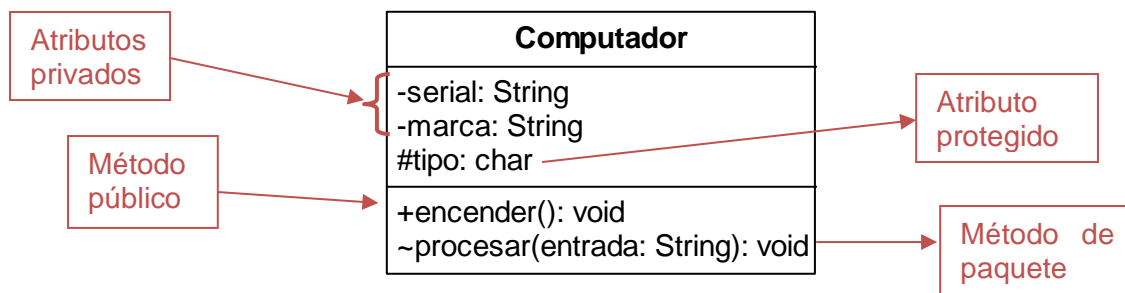
```
private void unMetodo() // un método privado
```

Por lo tanto, los paquetes permiten definir niveles de acceso, para garantizar así más control sobre el código.

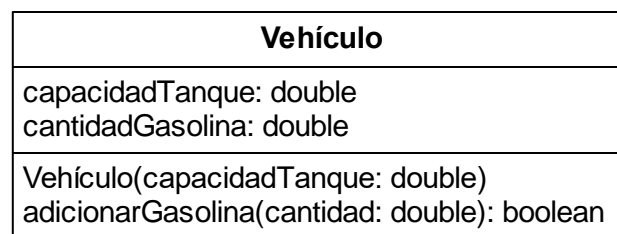
La forma de representar la visibilidad en el diagrama de clases es:

- privado
- + público
- # protegido
- ~ de paquete

Por ejemplo:



Supongamos que tenemos la clase Vehículo, representada en el siguiente diagrama:



En el método “adicionarGasolina” se valida que la cantidad no supere la capacidad del tanque. Sin embargo, como no se ha establecido la visibilidad, los atributos pueden ser modificados en cualquier momento desde otros objetos, lo cual no es conveniente.

Por ejemplo, se puede tener el siguiente código:

```

/**
 * Un conductor despistado que desea adicionar más gasolina
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Conductor
{
    public static void main(String arg[])
    {
        Vehiculo carro1 = new Vehiculo(10);
        boolean pudo;
        pudo = carro1.adicionarGasolina(5);    //BIEN, pudo=true
        pudo = carro1.adicionarGasolina(6);    //BIEN, pudo=false
        carro1.cantidadGasolina = 11;    //MAL: cambia sin validar
    }
}

```

Para evitar que esto suceda los objetos pueden ocultar sus atributos, usando la visibilidad. En la clase Vehículo se definirán los atributos como privados, para que desde ninguna otra clase se tenga acceso a ellos. Los métodos, por otra parte, se definirán como públicos, para que sí puedan ser usados. El diagrama se modifica de la siguiente forma:

| Vehículo  |
|---|
| -capacidadTanque: double<br>-cantidadGasolina: double                               |
| +Vehículo(capacidadTanque: double)<br>+adicionarGasolina(cantidad: double): boolean |

El código de la clase queda:

```

/**
 * Medio de transporte mecánico al cual se le puede adicionar gasolina
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Vehiculo
{
    private double capacidadTanque;
    private double cantidadGasolina;

    /**
     * Constructor de objetos Vehículo
     * @param capacidadTanque la capacidad del tanque, en litros
     */
    public Vehiculo(double capacidadTanque)
    {
        this.capacidadTanque = capacidadTanque;
        this.cantidadGasolina = 0;
    }
}

```

*//continúa*

```

/**
 * Adicionar gasolina, máximo hasta la capacidad del tanque
 * @param cantidad    litros de gasolina que se desean adicionar
 * @return si pudo o no adicionar la cantidad.
 */
public boolean adicionarGasolina(double cantidad)
{
    double maximo = this.capacidadTanque - this.cantidadGasolina;
    if (cantidad <= maximo)
    {
        this.cantidadGasolina += cantidad;
        return true;
    }
    else
    {
        return false;
    }
}
} // fin clase Vehiculo

```

De esta manera se resolvió el problema de que se pudiera modificar un atributo de manera incorrecta desde otros objetos. El código que se presentó al comienzo ya no permite hacer el cambio en el atributo, porque es privado:

```

// Lo siguiente ya NO COMPILA:
carro1.cantidadGasolina = 11;

```

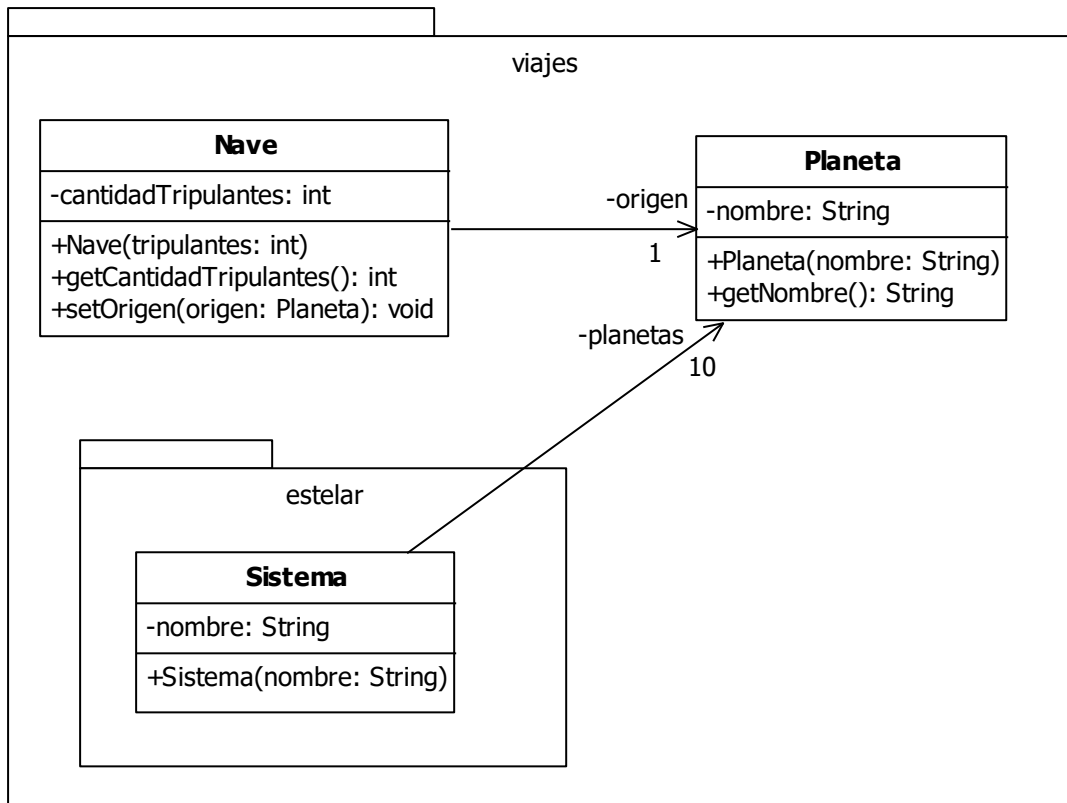
Algunas observaciones adicionales:

- El nivel de visibilidad es diferente al uso del *import*. El *import* es solo para decir dónde está la clase, pero tener permiso de usar los atributos y métodos de esa clase es diferente. Eso lo define el nivel de visibilidad que tenga cada uno.
- Cuando no se define ningún paquete para las clases, Java las coloca en un “paquete por defecto” (sin nombre). Esto puede servir para programas pequeños, pero no es práctico para programas grandes porque se empiezan a presentar problemas de ubicación de las clases.
- Las clases pueden tener visibilidad pública o de paquete. No se puede usar visibilidad protegida ni privada para las clases<sup>2</sup>.

### **Ejercicio 13-2**

Escriba el código Java para el siguiente diagrama de clases:

<sup>2</sup> Hay algunas excepciones, pero eso se escapa del alcance de este libro.



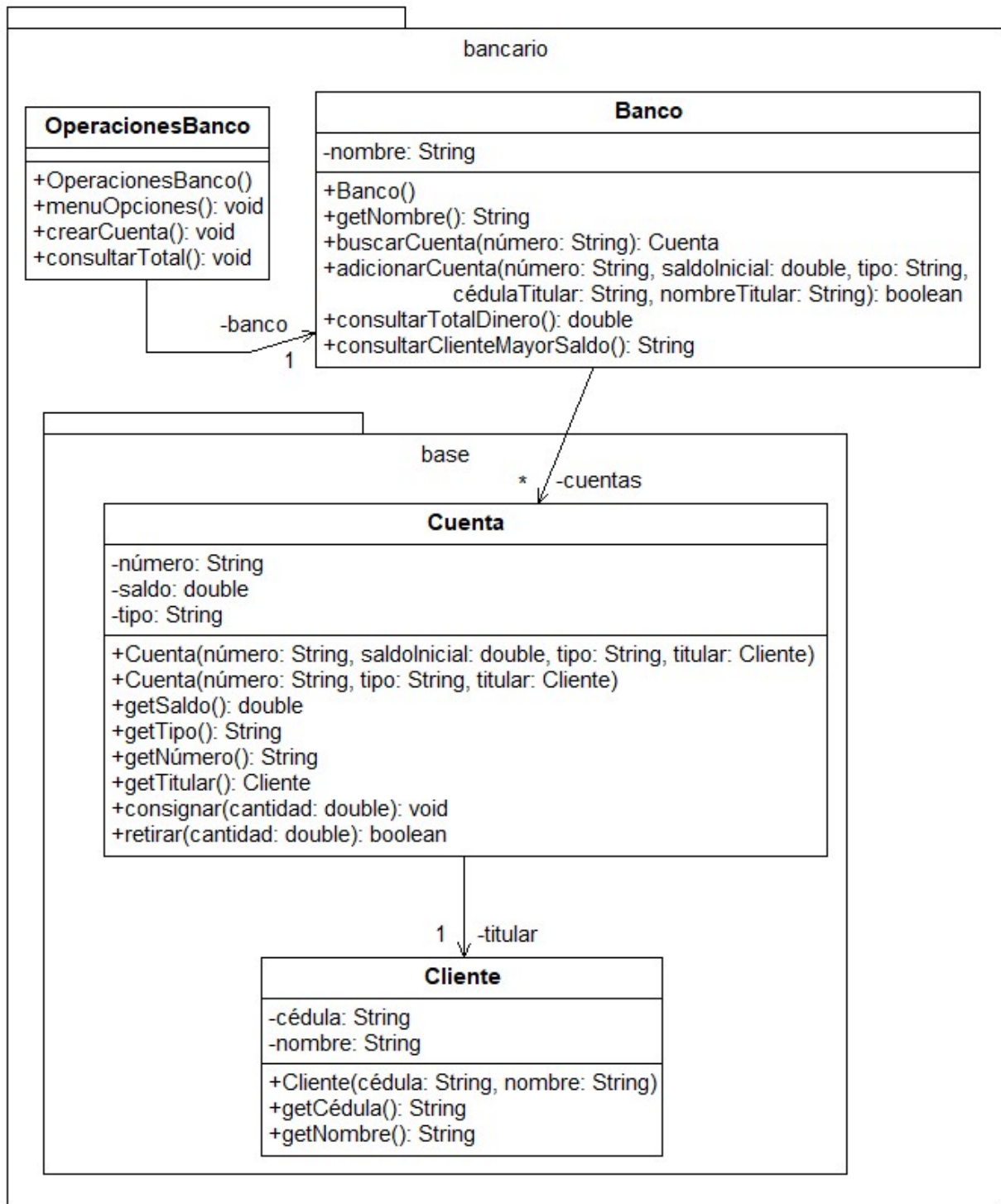
### 13.6. Ejercicio resuelto

Para mejorar la organización y la seguridad, se ha decidido organizar las clases del programa del banco MuchoDinero en paquetes. Se tendrá un paquete para la clase Banco, y un paquete interno para las clases Cliente y Cuenta. Además, los atributos serán todos privados, para que la única forma de consultarlos o modificarlos sea a través de los métodos, los cuales tienen las validaciones necesarias.

Es de resaltar que **los papeles o roles en las asociaciones también tienen visibilidad**, pues estos serán atributos cuando se pasen a código Java. En el caso del banco, tanto las cuentas como el dueño son privados.

Se tendrá una clase adicional, llamada OperacionesBanco, la cual tiene el código necesario para pedir los datos al usuario y mostrar los resultados. Tendrá un método para mostrar las diferentes opciones disponibles (que por el momento serán solo crear una cuenta y consultar el total de dinero en el banco) y un método para cada una de las opciones.

El diagrama de clases y el código correspondiente se presentan a continuación.





```

package bancario.base;

/**
 * Persona que posee una cuenta en un banco y por lo tanto,
 * es cliente del banco
 * @author Sandra V. Hurtado
 * @version 1.5
 */
public class Cliente
{
    private String cedula;
    private String nombre;

    /**
     * Constructor de un objeto cliente
     * @param cedula el número de cédula que identifica al cliente
     * @param nombre el nombre completo del cliente
     */
    public Cliente(String cedula, String nombre)
    {
        this.cedula = cedula;
        this.nombre = nombre;
    }

    public String getCedula()
    {
        return cedula;
    }

    public String getNombre()
    {
        return nombre;
    }
} // fin clase Cliente

```

```

package bancario.base;

/**
 * Una cuenta bancaria, es decir, un registro en
 * una entidad bancaria que tiene un saldo (cantidad de dinero)
 * que se puede modificar mediante retiros o consignaciones.
 * @author Sandra V. Hurtado
 * @version 3.0
 */
public class Cuenta
{
    private String numero;
    private double saldo;

```

*//continúa*

```

private String tipo;
private Cliente titular;

/**
 * Constructor de objetos Cuenta, con un saldo inicial.
 * @param numero el número de la cuenta
 * @param saldoInicial saldo inicial (en pesos) de la cuenta
 * @param tipo el tipo de cuenta: "ahorros" o "corriente"
 * @param titular el dueño de la cuenta - un objeto Cliente
 */
public Cuenta(String numero, double saldoInicial, String tipo,
               Cliente titular)
{
    this.numero = numero;
    this.saldo = saldoInicial;
    this.tipo = tipo;
    this.titular = titular;
}

/**
 * Constructor de objetos Cuenta, con saldo inicial cero.
 * @param numero el número de la cuenta
 * @param tipo el tipo de cuenta: "ahorros" o "corriente"
 * @param titular el dueño de la cuenta - un objeto Cliente
 */
public Cuenta(String numero, String tipo, Cliente titular)
{
    this.numero = numero;
    this.saldo = 0;
    this.tipo = tipo;
    this.titular = titular;
}

public double getSaldo()
{
    return this.saldo;
}

public String getTipo()
{
    return this.tipo;
}

public String getNumero()
{
    return this.numero;
}

public Cliente getTitular()
{
    return this.titular;
}

```

*//continúa*

```

/**
 * Retira o saca una cantidad de dinero de la cuenta,
 * si hay saldo suficiente.
 * @param cantidad la cantidad que se desea retirar, en pesos.
 * @return si se pudo hacer el retiro porque tenía dinero
 * suficiente o no (true o false)
 */
public boolean retirar(double cantidad)
{
    if (saldo >= cantidad)
    {
        saldo = saldo - cantidad;
        return true;
    }
    else
    {
        return false;
    }
}

/**
 * Consigna o adiciona una cantidad de dinero en la cuenta,
 * lo cual incrementa el saldo.
 * @param cantidad la cantidad de dinero que se desea
 * consignar, en pesos
 */
public void consignar(double cantidad)
{
    saldo = saldo + cantidad;
}
} // fin clase Cuenta

```

```
package bancario;
```

```
import java.util.ArrayList;
import bancaria.base.Cliente;
import bancaria.base.Cuenta;
```

Uso de varios *import*

```

/**
 * Entidad bancaria que maneja varias cuentas con dinero
 * @author Sandra V. Hurtado
 * @version 2.2
 */
public class Banco
{
    private String nombre;
    private ArrayList<Cuenta> cuentas;

```

*//continúa*

```

public Banco ()
{
    this.nombre = "MuchoDinero";
    this.cuentas = new ArrayList<>();
}

/**
 * Buscar una cuenta en la lista, a partir del número.
 * @param numero el número que identifica la cuenta que se buscará
 * @return el objeto Cuenta que corresponde al número dado,
 *         o null si no se encuentra.
 */
public Cuenta buscarCuenta(String numero)
{
    for (Cuenta cuentaComparar : cuentas)
    {
        if (cuentaComparar.getNumero().equals(numero))
        {
            return cuentaComparar;
        }
    }
    return null;
}

/**
 * Se crea una nueva cuenta y se adiciona a la lista.
 * @param numero el número de la nueva cuenta - debe ser único
 * @param saldoInicial el saldo inicial de dinero en la nueva cuenta
 * @param tipo si es una cuenta corriente o de ahorros
 * @param cedulaTitular la cédula del titular de la cuenta
 * @param nombreTitular el nombre completo del titular de la cuenta
 * @return true si se pudo crear y adicionar la cuenta,
 *         y false en caso contrario
 */
public boolean adicionarCuenta(String numero, double saldoInicial,
                                String tipo, String cedulaTitular, String nombreTitular)
{
    Cuenta cuentaExistente = buscarCuenta(numero);
    if (cuentaExistente == null)
    {
        // Se crea el nuevo cliente
        Cliente dueno = new Cliente(cedulaTitular,nombreTitular);
        // Se crea la cuenta con el cliente
        Cuenta nuevaCuenta =
            new Cuenta(numero, saldoInicial,tipo, dueno);
        return cuentas.add(nuevaCuenta);
    }
    return false;
}

```

*//continúa*

```

public String getNombre()
{
    return this.nombre;
}

/**
 * Consultar el total de dinero en todas las cuentas del banco
 * @return el total de dinero en el banco
 */
public double consultarTotalDinero()
{
    double cantidadDinero = 0;
    for (Cuenta cuenta : cuentas)
    {
        cantidadDinero += cuenta.getSaldo();
    }
    return cantidadDinero;
}

/**
 * Obtener el nombre del cliente con mayor saldo en el banco
 * @return el nombre del cliente con más saldo,
 *         o null si todavía no hay cuentas creadas.
 */
public String consultarClienteMayorSaldo()
{
    Cuenta cuentaMayor = null;
    double saldoMayor = 0;
    for (Cuenta cuenta : cuentas)
    {
        if (cuenta.getSaldo() > saldoMayor)
        {
            saldoMayor = cuenta.getSaldo();
            cuentaMayor = cuenta;
        }
    }
    if (cuentaMayor != null)
    {
        Cliente clienteMayorSaldo = cuentaMayor.getTitular();
        return clienteMayorSaldo.getNombre();
    }
    return null;
}

} // fin clase Banco

```

```

package bancario;

import javax.swing.JOptionPane;

/**
 * Clase que permite realizar operaciones básicas con las cuentas
 * de un banco. En esta clase se piden los datos necesarios al usuario
 * y se muestran los resultados.
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class OperacionesBanco
{
    private Banco banco;

    public OperacionesBanco()
    {
        this.banco = new Banco();
    }

    /**
     * Presenta al usuario las diferentes operaciones que puede realizar,
     * para que seleccione una de ellas.
     */
    public void menuOpciones()
    {
        int opcion;
        do {
            opcion=0;
            String valorSeleccionado =
                JOptionPane.showInputDialog("OPERACIONES BANCARIAS \n" +
                    "1. Crear nueva cuenta      \n" +
                    "2. Consultar el total de dinero en el banco \n" +
                    "0. Terminar      \n\n" +
                    "Ingrese el número correspondiente a la opción: ");

            // Verifica que no se haya presionado "Cancelar"
            if (valorSeleccionado != null)
            {
                opcion = Integer.parseInt(valorSeleccionado);
            }

            switch (opcion)
            {
                case 1:
                    crearCuenta();
                    break;
                case 2:
                    consultarTotal();
                    break;
                case 0:
                    break;
            }
        } while (opcion != 0);
    }

    //continúa

```

```

        default:
            JOptionPane.showMessageDialog(null,
                "Opción no disponible");
        }
    }
    while (opcion != 0);
}

/**
 * Solicita todos los datos necesarios para una nueva cuenta,
 * y los envía al banco para su creación.
 * También muestra un mensaje al usuario indicando si se pudo crear
 * la cuenta o no (dependiendo de la respuesta del banco).
 */
public void crearCuenta()
{
    String numeroCuenta =
        JOptionPane.showInputDialog("Número de la cuenta:");
    String saldoCadena =
        JOptionPane.showInputDialog("Saldo inicial de la cuenta:");
    double saldoCuenta = Double.parseDouble(saldoCadena);
    String tipoCuenta =
        JOptionPane.showInputDialog("Tipo (corriente o ahorros:");
    String cedulaTitular =
        JOptionPane.showInputDialog("Número de cédula del dueño:");
    String nombreTitular =
        JOptionPane.showInputDialog("Nombre completo del dueño:");
    boolean pudoCrear =
        banco.adicionarCuenta(numeroCuenta, saldoCuenta,
            tipoCuenta, cedulaTitular, nombreTitular);
    if (pudoCrear)
    {
        JOptionPane.showMessageDialog(null, "Cuenta creada");
    }
    else
    {
        JOptionPane.showMessageDialog(null,
            "No se pudo crear la cuenta. Número repetido");
    }
}

/**
 * Muestra el total de dinero que tiene el banco.
 */
public void consultarTotal()
{
    double totalDinero = banco.consultarTotalDinero();
    JOptionPane.showMessageDialog(null,
        "El total de dinero en el banco es: $" + totalDinero);
}
} // fin clase OperacionesBanco

```

## 14. Polimorfismo por Sobrecarga

En este capítulo se presentará el concepto de polimorfismo, y se explicará uno de los tipos de polimorfismo que se tienen en orientación a objetos: el polimorfismo por sobrecarga.

A finalizar este capítulo usted debe ser capaz de:

- Identificar clases que tengan polimorfismo por sobrecarga.
- Escribir una clase en Java que use polimorfismo por sobrecarga.

### 14.1. Definición

Una de las características importantes de los objetos, y en general de la Programación Orientada a Objetos (POO), es la capacidad de comportarse diferente ante un mismo mensaje, dependiendo de los parámetros que recibe o del tipo de objeto que sea. Esto se conoce como **polimorfismo**.

Esta capacidad de comportarse diferente puede darse en dos casos:

- Cuando se pide el mismo servicio a diferentes objetos, y cada uno de ellos puede hacerlo de forma diferente. Esto se conoce como **polimorfismo por sobrescritura**.
- Cuando la información que se le da al objeto para solicitarle el servicio cambia. Es decir, se envían diferente tipo y cantidad de parámetros a un método. Esto se conoce como **polimorfismo por sobrecarga**.

El polimorfismo por sobrecarga se presenta cuando se tiene un mismo método pero que cambia su comportamiento dependiendo de los parámetros que recibe.

Por ejemplo, si al objeto “Juanito” (de tipo Persona) se le pide “comer” puede hacerlo de diferentes formas. Si se le dice comer y se le manda una hamburguesa (y nada más), Juanito la come sosteniéndola con sus manos; pero si se le dice comer y se le manda arroz y una cuchara, Juanito come el arroz usando la cuchara.

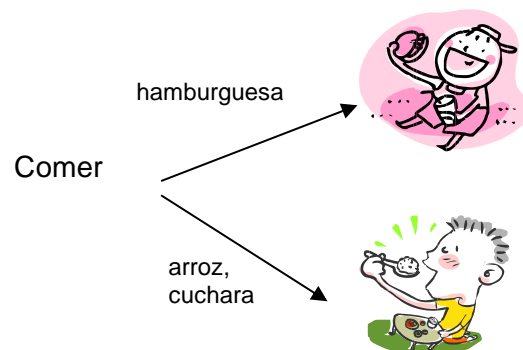


Imagen 14-1 Comportamiento “comer” con diferentes parámetros

Es el mismo comportamiento (comer) pero recibe diferentes parámetros y por lo tanto el objeto se comporta diferente.

Otro ejemplo: Las cadenas (objetos *String*), tienen el método “substring”, que permite obtener una sub-cadena a partir de otra cadena, lo cual se puede hacer de dos formas:



- Enviando como parámetros dos números enteros que representan las posiciones inicial y final dentro de la cadena inicial, o
- Enviando un solo número entero, de manera que la sub-cadena se obtendrá a partir de esa posición y hasta el final de la cadena inicial.

Por ejemplo:

```
String cadenaOriginal = new String("Esta es una cadena de texto");
String subcadenaUno = cadenaOriginal.substring(5,18);
String subcadenaDos = cadenaOriginal.substring(5);
System.out.println(subcadenaUno);
System.out.println(subcadenaDos);
```

La salida del anterior código es:

```
es una cadena
es una cadena de texto
```

En este caso se usa el mismo método: “substring”, sobre el mismo objeto –cadenaOriginal-, pero como tienen un número diferente de parámetros el resultado cambia.

### Ejercicio 14-1

Revise en el API de Java la documentación de la clase *String* y observe si hay otros ejemplos de métodos sobrecargados.

## 14.2. Implementación en Java

En Java el polimorfismo por sobrecarga se logra cuando en la misma clase se tienen **varios métodos con el mismo nombre**. Sin embargo, los métodos deben diferenciarse en el tipo o el número de parámetros que reciben.

Por ejemplo, se tiene la clase Vehículo, que se muestra a continuación:

| Vehículo     |   |
|--------------|---|
| Sobrecarga { | -capacidadTanque: double                      |
|              | -cantidadGasolina: double                     |
|              | +Vehículo(capacidadTanque: double)            |
|              | +getCantidadGasolina(): double                |
|              | +adicionarGasolina(cantidad: double): boolean |
|              | +adicionarGasolina(): void                    |

En este caso al vehículo se le puede adicionar más gasolina de dos formas: indicando la cantidad de gasolina que se desea echar o sin indicar nada. Si no se indica nada entonces se llena el tanque.

El código de esta clase es:

```

/**
 * Medio de transporte mecánico al cual se le puede adicionar gasolina
 * @author Sandra V. Hurtado
 * @version 2.0
 */
public class Vehiculo
{
    private double capacidadTanque;
    private double cantidadGasolina;

    /**
     * Constructor de objetos Vehículo
     * @param capacidadTanque la capacidad del tanque, en litros
     */
    public Vehiculo(double capacidadTanque)
    {
        this.capacidadTanque = capacidadTanque;
        this.cantidadGasolina = 0;
    }

    public double getCantidadGasolina()
    {
        return this.cantidadGasolina;
    }

    /**
     * Adicionar gasolina, máximo hasta la capacidad del tanque
     * @param cantidad litros de gasolina que se desean adicionar
     * @return si pudo o no adicionar la cantidad.
     */
    public boolean adicionarGasolina(double cantidad)
    {
        double maximo = this.capacidadTanque - this.cantidadGasolina;
        if (cantidad <= maximo)
        {
            this.cantidadGasolina += cantidad;
            return true;
        }
        return false;
    }

    /**
     * Adicionar gasolina hasta la capacidad del tanque
     */
    public void adicionarGasolina()
    {
        this.cantidadGasolina = this.capacidadTanque;
    }
}

```

Java determina automáticamente cuál es el método que se debe llamar dependiendo de los parámetros que se envíen.

Por ejemplo, en el siguiente código se adiciona gasolina a un vehículo, usando alguno de los dos métodos definidos para ello:

```
import java.util.Scanner;

/**
 * Lugar donde se echa gasolina a los carros
 * @author Sandra V. Hurtado
 * @version 2015-10-15
 */
public class Gasolineria
{
    public static void main(String[] args)
    {
        Scanner lector = new Scanner(System.in);

        // Se crea un carro de 30 galones de capacidad
        Vehiculo carro = new Vehiculo(30);

        System.out.println("¿Desea llenar el tanque? (Si/No)");
        String llenado = lector.nextLine();

        if (llenado.equalsIgnoreCase("Si"))
        {
            carro.adicionarGasolina();           // sin parámetros
        }
        else
        {
            System.out.println("Litros que desea echar: ");
            int cantidad = lector.nextInt();
            carro.adicionarGasolina(cantidad); // con un parámetro
        }
        double litros = carro.getCantidadGasolina();
        System.out.println("Carro con "+litros+" litros");
        lector.close();
    }
}
```

### **Ejercicio 14-2**

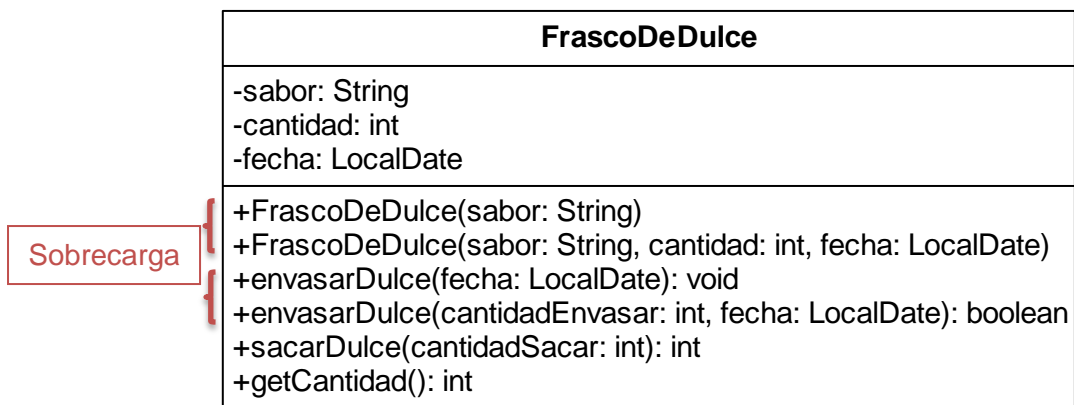
En el país de los cuentos hay un espejo mágico en la puerta de entrada a la ciudad de la imaginación. Cuando un viajero llega le puede decir al espejo qué tipo de ser es, o le puede decir cuál es su edad. Dependiendo de lo que diga el viajero el espejo determina si puede entrar o no. Lo que no saben los viajeros del país de los cuentos, es que cuando se dice qué tipo de ser es, el espejo deja pasar solo a los que dicen que son magos, hadas o dragones, los demás no pueden entrar. En caso de decir la edad, el espejo solo deja pasar a los que digan tener menos de 18 años.

Usted debe elaborar el diagrama de clases para el enunciado anterior y escribir el código Java correspondiente. Recuerde que debe usar sobrecarga.

### 14.3. Ejercicio resuelto

La abuela Pata elabora deliciosos dulces que envasa para su conservación y posiblemente para la venta. La mayoría de las veces la abuela crea cada frasco de dulce con todos los datos: el sabor, la cantidad y la fecha. Sin embargo, a veces la abuela deja listos los frascos para envasar dulces de cierto sabor (es decir, están listos los frascos, pero sin cantidad ni fecha), y más tarde ese día o el día siguiente envasa el dulce. Además, cuando envasa puede simplemente llenar el envase (son 200 gramos), o puede ser que defina la cantidad que desea envasar, cuando ya se está terminando el dulce.

Se hará uso del polimorfismo por sobrescritura para tener dos constructores y dos formas de envasar los frascos. El diagrama que representa esto es:



Y el código Java correspondiente:

```
import java.time.LocalDate;

/**
 * Un frasco con delicioso dulce, elaborado por la abuela pata.
 * Por ejemplo, un frasco de dulce de guayaba.
 * @author Sandra V. Hurtado
 * @version 3.0
 */
public class FrascoDeDulce
{
    private String sabor;
    private int cantidad;
    private LocalDate fecha;

    //continúa
```

```

/**
 * Constructor de nuevos objetos FrascoDeDulce, sin
 * definir todavía la cantidad ni la fecha.
 * @param sabor el sabor del dulce, por ejemplo, "brevas"
 */
public FrascoDeDulce(String sabor)
{
    this.sabor = sabor;
    this.cantidad = 0;
    this.fecha = null;
}

/**
 * Constructor de nuevos objetos FrascoDeDulce.
 * @param sabor el sabor del dulce, por ejemplo, "brevas"
 * @param cantidad la cantidad, en gramos, del dulce en el frasco
 * @param fecha la fecha en la cual se elaboró el dulce
 */
public FrascoDeDulce(String sabor, int cantidad, LocalDate fecha)
{
    this.sabor = sabor;
    this.cantidad = cantidad;
    this.fecha = fecha;
}

/**
 * Envasar o insertar dulce en el frasco (el total: 200 gramos).
 * @param fecha la fecha en la cual se envasa el dulce
 */
public void envasarDulce(LocalDate fecha)
{
    this.fecha = fecha;
    this.cantidad = 200;
}

/**
 * Envasar o insertar dulce en el frasco, hasta máximo 200 gramos.
 * @param cantidadEnvasar la cantidad, en gramos, para envasar
 * @param fecha la fecha en la cual se envasa el dulce
 * @return una indicación (true/false) de si se pudo envasar o no
 * toda la cantidad deseada o solo 200 gramos
 */
public boolean envasarDulce(int cantidadEnvasar, LocalDate fecha)
{
    this.fecha = fecha;
    if (cantidadEnvasar > 200)
    {
        this.cantidad = 200;
        return false;
    }
}

```

*//continúa*

```

        else
        {
            this.cantidad = cantidadEnvasar;
            return true;
        }
    }

    /**
     * Saca cierta cantidad del dulce del frasco, si hay suficiente
     * en el frasco; pero si hay menos solo se puede sacar lo que queda.
     * @param cantidadSacar la cantidad, en gramos, para sacar
     * @return la cantidad de dulce que se pudo sacar, en gramos
     */
    public int sacarDulce(int cantidadSacar)
    {
        int cantidadSacada = 0;
        if (this.cantidad >= cantidadSacar)
        {
            this.cantidad = this.cantidad - cantidadSacar;
            cantidadSacada = cantidadSacar;
        }
        else
        {
            cantidadSacada = this.cantidad;
            this.cantidad = 0;
        }
        return cantidadSacada;
    }

    public int getCantidad()
    {
        return this.cantidad;
    }

    public String getSabor()
    {
        return sabor;
    }

    public LocalDate getFecha()
    {
        return fecha;
    }

} // fin clase FrascoDeDulce

```

## 15. Herencia

En este capítulo se presentará el concepto de herencia, el cual es uno de los más importantes en el paradigma orientado a objetos, y contribuye a la reutilización y a la flexibilidad del código.

A finalizar este capítulo usted debe ser capaz de:

- Representar la herencia en un diagrama de clases.
- Escribir clases en Java que hagan uso del mecanismo de herencia.

### 15.1. Ejemplo introductorio

La programación orientada a objetos simplifica en gran medida el trabajo con elementos del mundo real, gracias a que se pueden modelar muchos objetos que comparten características y comportamiento con una sola clase.

Sin embargo, en ocasiones hay objetos similares, que comparten características y comportamiento, pero que tienen algunas pequeñas diferencias. Por ejemplo, Doña Pepa tiene de mascota a un perrito llamado Fido, que es muy juguetón y le gustan las galletas para perro en forma de hueso. A Fido hay que sacarlo a pasear cada día, y cuando está alegre ladra y mueve la cola. Por otra parte, su vecina, Doña Marujita, tiene como mascota a un gatico llamado Fifi, que también es juguetón y le gusta mucho el atún. Fifi no necesita que lo saquen a pasear porque es muy independiente, y solo maúlla cuando desea advertir de algo extraño.



Fido



Fifi

*Imagen 15-1 Mascotas Fido y Fifi*

Para estos dos objetos se ha pensado en tener dos clases diferentes:

| Perro  |
|--|
| -nombre: String<br>-comidaFavorita: String   |
| +jugar(): void<br>+comer(): void<br>+pasear(): void<br>+ladrar(): void<br>+moverCola(): void |

| Gato   |
|--|
| -nombre: String<br>-comidaFavorita: String           |
| +jugar(): void<br>+comer(): void<br>+maullar(): void |

Se puede observar que, aunque hay elementos diferentes, también hay cosas comunes. Al analizar en detalle, observamos que tanto Fido como Fifi son **mascotas**, y por lo tanto se puede tener una clase compartida:

| Mascota                                    |
|--|
| -nombre: String<br>-comidaFavorita: String |
| +jugar(): void<br>+comer(): void           |

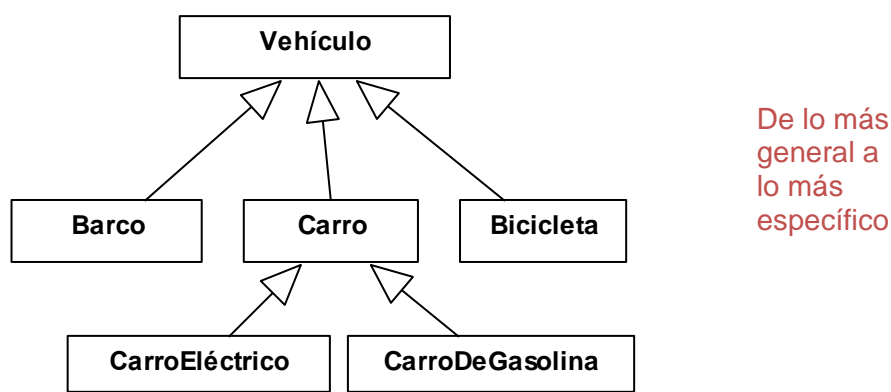
Esta clase tiene los elementos comunes, pero se perdería información de las diferencias. Para conservar tanto lo común como las diferencias se usa la herencia, la cual se explica a continuación.

## 15.2. Concepto de herencia y representación

En Orientación a Objetos uno de los mecanismos que ayudan a modelar un sistema es la **herencia**. Mediante este mecanismo es posible identificar clases que son especializaciones o subtipos de otras.

Por ejemplo, los barcos, los carros y las bicicletas son tipos o especializaciones de vehículos. Además, los carros pueden ser de varios tipos: eléctricos o de gasolina.

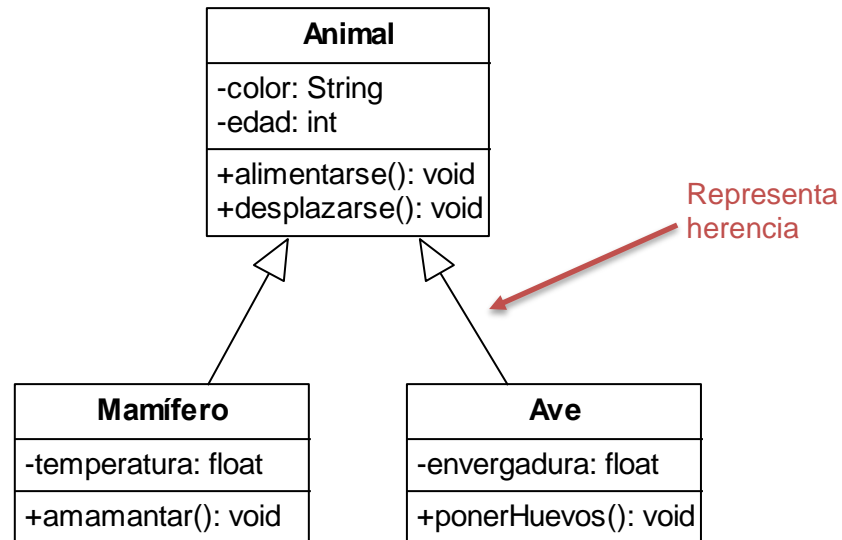
Esto se representa en un diagrama de clases de la siguiente forma:



Es decir, un barco es un tipo de vehículo, lo mismo que un carro y una bicicleta: todos son vehículos. A su vez, un carro eléctrico es un carro, y también es un vehículo, y lo mismo un carro de gasolina.

Al definir una jerarquía de clases, las clases que son subtipos o especializaciones tienen todo lo que está definido en las clases generales (**"lo heredan"**), y además pueden tener atributos o métodos que sean solo de ellas. Por ejemplo:



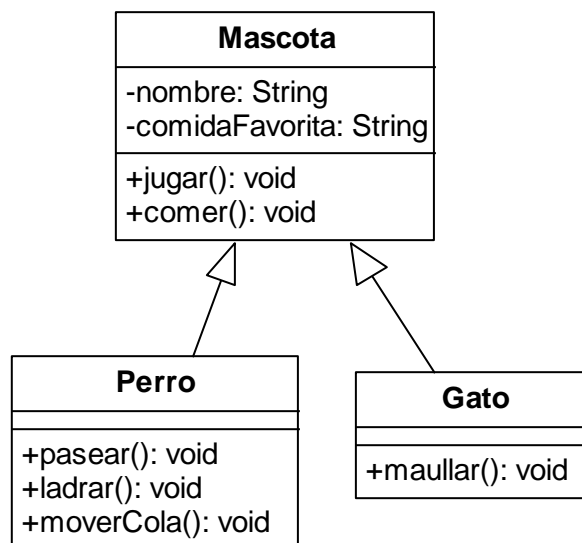


Los mamíferos **SON** animales, y por lo tanto tienen todas las características y el comportamiento de un animal, pero además tiene características y comportamientos propios que no tienen otros tipos de animales.

Los mamíferos, por lo tanto, tendrán: color, edad y temperatura, y pueden: alimentarse, desplazarse y amamantar.

Las aves también SON animales, y tienen como características: color, edad y envergadura<sup>3</sup>. Para las aves no es necesario conocer la temperatura. En cuanto al comportamiento pueden alimentarse y desplazarse, como todos los animales, pero no amamantan a sus crías. En su lugar las aves ponen huevos.

En el caso de Fido y Fifi, con la herencia se puede establecer que tanto el perro como el gato son mascotas. De esta manera pueden tener atributos y métodos compartidos, más el comportamiento que sea de cada uno. El diagrama queda:



<sup>3</sup> Distancia entre las puntas de las alas

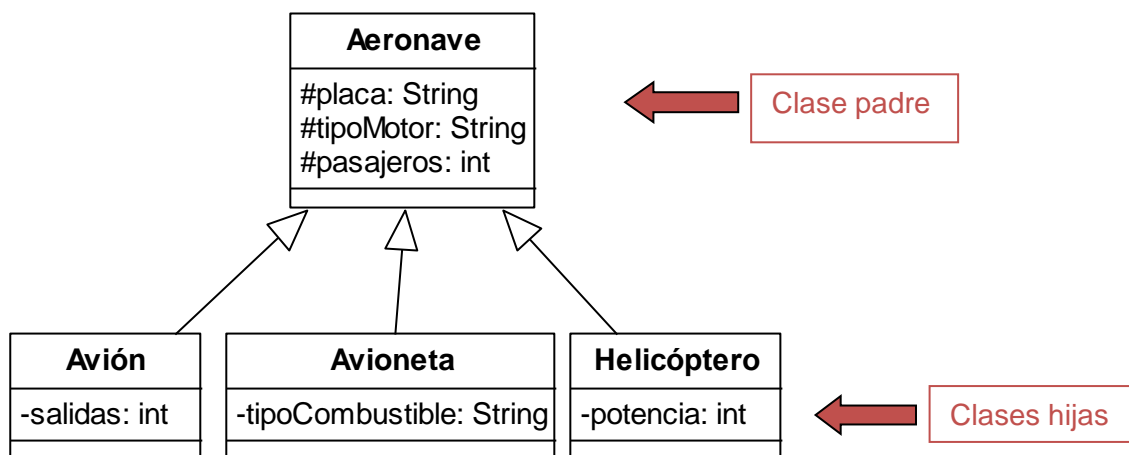
### Ejercicio 15-1

Presente un ejemplo del mundo real donde se pueda definir una clase general y unas clases especializadas o subtipos de esa clase general.

Al definir una herencia en programación Orientada a Objetos también se simplifica la elaboración del código, porque las **clases hijas** (los subtipos) no tienen que volver a definir todo lo que está en la **clase padre** (la más general).

Por ejemplo, en una empresa de aviación tienen diferentes tipos de aeronaves: los aviones, las avionetas y los helicópteros. Todas las aeronaves vienen en diferentes colores y tienen establecidos el tipo de motor y el número de pasajeros. Para los aviones es importante definir el número de salidas que tienen y para los helicópteros se debe saber cuál es su potencia máxima en condiciones de carga completa. En el caso de las avionetas es importante definir el tipo de combustible que usan: gasolina o energía solar, pues se está popularizando el uso de energías alternativas.

Para el anterior enunciado el diagrama de clases correspondiente, mostrando solo los atributos, es:



El código de las clases **Aeronave** y **Avión** es:

```
/**
 * Medio de transporte aéreo, que posee una empresa de aviación
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Aeronave
{
    protected String placa;
    protected String tipoMotor;
    protected int pasajeros;
}
```

```

/**
 * Un tipo de aeronave de gran tamaño, con varias salidas,
 * por lo general usado para vuelos comerciales
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Avion extends Aeronave
{
    private int salidas;
}

```

Es importante tener presente que las clases hijas **heredan** todo lo que está definido en la clase padre. En el ejemplo anterior:

- La clase Avión tiene cuatro atributos: tres que hereda (“placa”, “tipoMotor” y “pasajeros”) y uno propio (“salidas”). No es necesario que defina de nuevo los atributos de la clase padre, los incluye automáticamente cuando indica que es su hija.
- La clase Avioneta también tiene cuatro atributos: los tres que hereda de la clase Aeronave y su atributo propio: “tipoCombustible”.

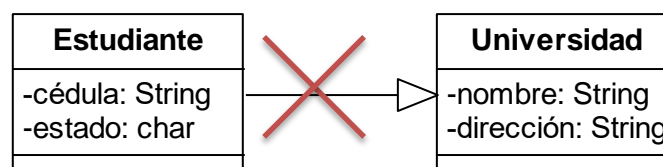
### Ejercicio 15-2

¿Cuáles son los atributos de la clase Helicóptero? ¿Cuáles de esos atributos se tienen que definir en el código y cuáles no?

Es importante tener en cuenta que la herencia no es solo para simplificar la elaboración del código, pues tiene implicaciones más importantes: una clase hija no solo debe compartir atributos y métodos de la clase padre, sino que debe tener el mismo significado. De esta manera, donde se pueda tener un objeto de la clase padre, se podrá tener un objeto de la clase hija.

Se puede entender que cada clase hija “ES UN(A)” tipo de la clase padre, lo cual ayuda a verificar si la herencia se ha identificado apropiadamente. Por ejemplo: un cuadrado **es una** figura geométrica, un mamífero **es un** animal, un colectivo **es un** vehículo.

Por lo tanto, en el siguiente diagrama no es correcta la relación de herencia entre las clases:



Aunque el estudiante tenga nombre y dirección, no se debe usar la herencia para “definir” estos atributos, puesto que un estudiante **no es una** universidad.

En este caso, en la clase Estudiante se deben definir todos los atributos, incluyendo nombre y dirección. Entre Estudiante y Universidad se puede tener otra relación, por ejemplo, una asociación, pero no una herencia.

### Ejercicio 15-3

Elabore el diagrama de clases para el siguiente enunciado:

En el museo de arte “Arte Actual para Todos - AAT” se exhiben diferentes obras para que las personas puedan apreciarlas y a la vez conocer las tendencias en el arte moderno. En el museo se exhiben pinturas y esculturas, todas de reconocidos artistas nacionales. Por ejemplo, está exhibido la escultura “Gato durmiente” de Hernando Tejada, y el cuadro “Retrato” de Alejandro Obregón. En el momento el museo tiene exhibidas cerca de 80 obras, pero se espera llegar hasta las 120 obras, que es la capacidad máxima del museo.

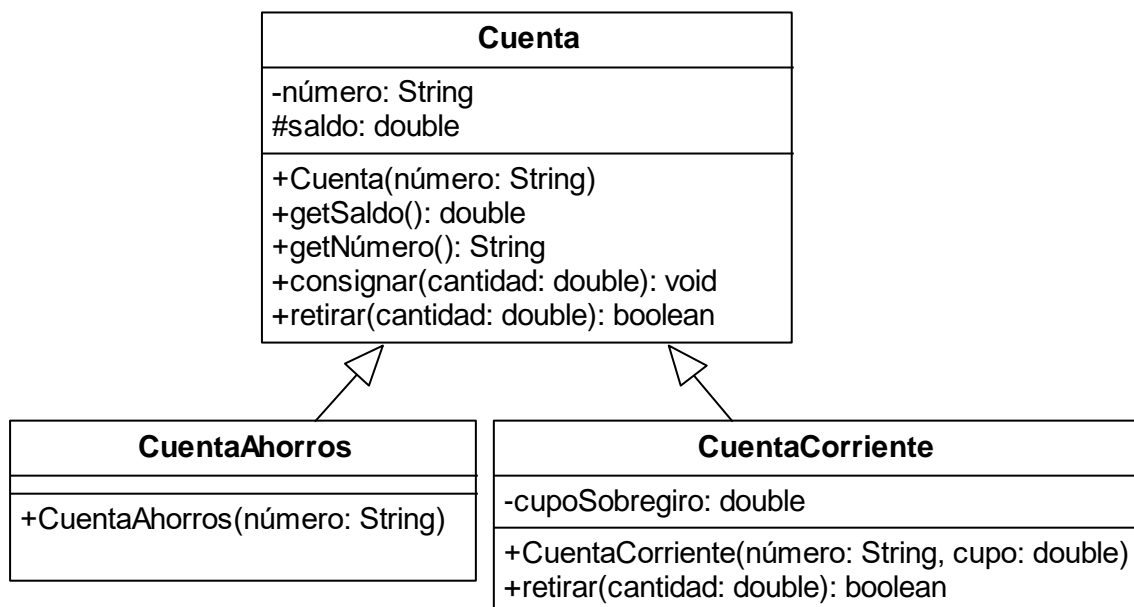
Como la idea del museo es dar a conocer el arte, de cada obra se tiene su información básica, que incluye la técnica utilizada y una breve descripción. Además, para las esculturas se indica el material utilizado y para las pinturas se indican sus dimensiones (alto y largo).

### 15.3. Implementación en Java

Para definir la herencia en Java, se tiene la palabra **extends** en las clases hijas, para indicar de cual clase heredan. La sintaxis es:

```
public class ClaseHija extends ClasePadre
{
    //código de la clase
}
```

Por ejemplo, en un banco existen cuentas corrientes y cuentas de ahorro. Aunque ambos tipos de cuentas tienen características y servicios comunes, cada una se comporta de manera diferente: En las cuentas de ahorro no se pueden tener saldos negativos, mientras que en las cuentas corrientes sí se pueden tener saldos negativos, hasta cierta cantidad o cupo (lo que se llama sobregiro). Esto hace que la forma de retirar dinero de cada cuenta sea diferente, porque se deben hacer diferentes validaciones. El diagrama de clases que muestra esto, es:



Es importante tener en cuenta que cada clase debe tener su propio constructor. **Los constructores son los únicos métodos que no se heredan.** Sin embargo, en las hijas es posible usar el constructor de la clase padre, lo cual se verá en la siguiente sección.

El código para las clases Cuenta y sus hijas:

```
/**
 * Una cuenta bancaria, es decir, un registro en
 * una entidad bancaria que tiene un saldo (cantidad de dinero)
 * que se puede modificar mediante retiros o consignaciones.
 * @author Sandra V. Hurtado
 * @version 3.5
 */
public class Cuenta
{
    private String numero;
    protected double saldo;

    /**
     * Constructor de objetos Cuenta, con saldo inicial cero.
     * @param numero el número de la cuenta
     */
    public Cuenta(String numero)
    {
        this.numero = numero;
        this.saldo = 0;
    }

    public double getSaldo()
    {
        return this.saldo;
    }

    public String getNumero()
    {
        return this.numero;
    }

    /**
     * Consigna o adiciona una cantidad de dinero en la cuenta,
     * lo cual incrementa el saldo.
     * @param cantidad la cantidad de dinero que se desea
     *                consignar, en pesos
     */
    public void consignar(double cantidad)
    {
        saldo = saldo + cantidad;
    }
} //fin clase Cuenta
```

```

/**
 * Cuenta corriente en una entidad financiera,
 * en la cual se puede retirar incluso si no hay saldo,
 * hasta el monto dado por el cupo de sobregiro.
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class CuentaCorriente extends Cuenta
{
    private double cupoSobregiro;

    /**
     * Constructor de objetos cuenta corriente
     * @param numero el número de la cuenta
     * @param cupo el máximo que puede retirar
     * después de tener saldo cero
     */
    public CuentaCorriente(String numero, double cupo)
    {
        super(numero);
        cupoSobregiro = cupo;
    }

    /**
     * Retira o saca una cantidad de dinero de la cuenta,
     * incluso quedando negativo, hasta el cupo de sobregiro.
     * @param cantidad la cantidad que se desea retirar, en pesos.
     * @return si se pudo hacer el retiro porque tenía dinero
     * suficiente o no (true o false)
     */
    public boolean retirar(double cantidad)
    {
        if ((getSaldo() + cupoSobregiro) >= cantidad)
        {
            saldo = saldo - cantidad;
            return true;
        }
        else
        {
            return false;
        }
    }
}

```

CuentaCorriente hereda de Cuenta

Llama al constructor de la clase padre

Usa uno de los métodos que hereda

```

/**
 * Cuenta de ahorros en una entidad financiera,
 * donde solo se puede retirar dinero hasta el saldo.
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class CuentaAhorros extends Cuenta
{
    /**
     * Constructor de objetos cuenta de ahorros
     * @param numero el número que identifica la cuenta
     */
    public CuentaAhorros(String numero)
    {
        super(numero);
    }

    /**
     * Retira o saca una cantidad de dinero de la cuenta,
     * si hay saldo suficiente.
     * @param cantidad la cantidad que se desea retirar, en pesos.
     * @return si se pudo hacer el retiro porque tenía dinero
     * suficiente o no (true o false)
     */
    public boolean retirar(double cantidad)
    {
        if (getSaldo() >= cantidad)
        {
            saldo = saldo - cantidad;
            return true;
        }
        else
        {
            return false;
        }
    }
}

```

CuentaAhorros hereda de Cuenta

Llama al constructor de la clase

Usa uno de los métodos que hereda

Es importante notar que, aunque las clases hijas heredan todos los atributos de la clase padre, no los pueden usar directamente si tienen visibilidad privada (*private*). En ese caso deben usarlos mediante los métodos. Por ese motivo no se le puede asignar un valor directamente al atributo “número” en las clases hijas, y es necesario usar el constructor de la clase padre.

## 15.4. Palabra reservada *super*

En el constructor de la clase hija se reciben los valores de los atributos, e internamente, para asignar el valor a los atributos que heredó, hace uso del constructor de la clase papá. Esto se hace con la palabra reservada ***super***, pasándole los argumentos correspondientes.

El uso de *super* para llamar al constructor de la clase papá, debe ser la primera instrucción dentro del método constructor de la clase hija. No se pueden tener otras instrucciones antes.

La palabra reservada *super* es similar a la palabra *this*, pero se hace referencia a los atributos o métodos definidos en la clase padre. Por ejemplo, en el método “retirar” de la cuenta de ahorros se puede tener algo como:

```
/**
 * Retira o saca una cantidad de dinero de la cuenta,
 * si hay saldo suficiente.
 * @param cantidad la cantidad que se desea retirar, en pesos.
 * @return si se pudo hacer el retiro porque tenía dinero
 * suficiente o no (true o false)
 */
public boolean retirar(double cantidad)
{
    if (super.getSaldo() >= cantidad)
    {
        super.saldo = super.saldo - cantidad;
        return true;
    }
    else
    {
        return false;
    }
}
```

## 15.5. Uso de objetos con herencia

La creación y el uso de los objetos de clases hijas es igual a la creación y uso de objetos de cualquier otra clase. Como ejemplo, se mostrará un código (en otra clase) donde se crean una cuenta corriente y una cuenta de ahorros; en cada una se consignan \$500.000 y luego se intenta retirar \$600.000 de cada una, teniendo en cuenta que la cuenta corriente tendrá un cupo de sobregiro de \$200.000.

```
/**
 * Clase donde se crean unas cuentas bancarias (corriente y de ahorros),
 * se consigna dinero en ellas y luego se retiran
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class PruebaCuentas
{
    public static void main(String[] args)
    {

```

*//continúa*



```

CuentaAhorros ahorros = new CuentaAhorros("123-a");
CuentaCorriente corriente =
    new CuentaCorriente("456-c", 200000);

ahorros.consignar(500000);
corriente.consignar(500000);

boolean pudoAhorro = ahorros.retirar(600000);
boolean pudoCorriente = corriente.retirar(600000);

if (pudoAhorro)
{
    System.out.println("Pudo retirar -cuenta de ahorros");
}
else
{
    System.out.println("No pudo retirar -cuenta de ahorros");
}

if (pudoCorriente)
{
    System.out.println("Pudo retirar -cuenta corriente");
}
else {
    System.out.println("No pudo retirar -cuenta corriente ");
}
}
} // fin clase PruebaCuentas

```

La salida al ejecutar este código es:

```

No pudo retirar -cuenta de ahorros
Pudo retirar -cuenta corriente

```

Al observar el código anterior se puede notar que hay partes del código que son iguales para los dos tipos de objetos (ahorros y corriente), porque ambos son cuentas bancarias. De hecho, cuando se tiene herencia es posible definir variables de la clase que referencien objetos de las clases hijas.

Por ejemplo, en el siguiente fragmento de código, se crea primero un objeto CuentaCorriente y luego un objeto CuentaAhorros, pero ambos usando una sola variable, de tipo Cuenta. Esto es posible porque tanto las cuentas corrientes como las cuentas de ahorro son cuentas.

**Cuenta** bancaria;

Variable de la clase padre

```
// Primero se crea una cuenta corriente
bancaria = new CuentaCorriente("890-c",350000);
bancaria.consignar(450000);
System.out.println("Cuenta con saldo: $" +bancaria.getSaldo());

// Luego se crea una cuenta de ahorros
bancaria = new CuentaAhorros("654-a");
bancaria.consignar(450000);
System.out.println("Cuenta con saldo: $" +bancaria.getSaldo());
```

Es importante resaltar que esto solo es posible con clases hijas, no es posible definir una variable de una clase y asignarle un objeto de otra clase cuando no hay herencia entre ellas. Por ejemplo, lo siguiente no es válido:

```
// Esto no es válido:
CuentaCorriente cuentaDinero = new CuentaAhorros("730-a"); //ERROR
```

En este caso, una cuenta de ahorros NO es una cuenta corriente.

Cuando se utiliza una variable de la clase padre para usar los objetos de las clases hijas hay una limitante: por medio de esa variable solo es posible usar los métodos que estén definidos en la clase papá, no los que estén definidos en la clase hija. Sin embargo, existen varias formas de superar esta limitante, algunas de las cuales se verán en los siguientes capítulos.

### **Ejercicio 15-4**

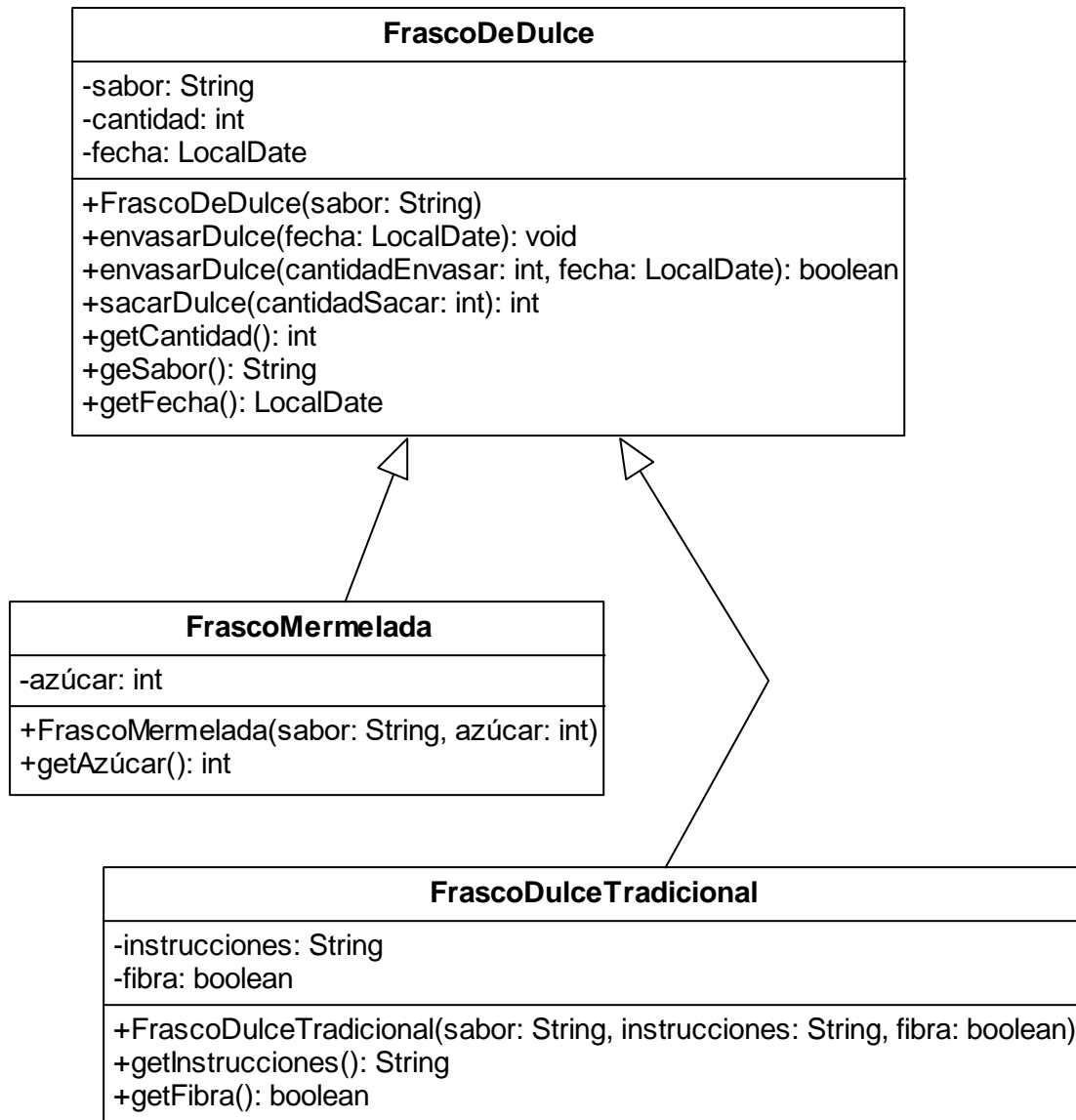
Elabore el diagrama de clases y el código en Java, para el siguiente enunciado:

En una empresa de repuestos para automóviles se fabrican partes simples y partes compuestas. Las partes simples tienen número, nombre y precio, y las partes compuestas tienen número, nombre, precio base, precio de ensamble y una descripción del proceso de fabricación. En la empresa desean registrar la información de las diferentes partes que fabrican y también saber cuál es el precio de las partes. Hay que tener en cuenta que el precio de una parte simple es su precio base más un 8 %, mientras que el precio de una parte compuesta es su precio base más el precio de ensamble, todo con un incremento del 11 %.

## **15.6. Ejercicio resuelto**

La abuela Pata está pensando en vender los deliciosos dulces que elabora, para lo cual debe definir muy bien cuáles tipos de dulces ofrecerá. Ha pensado inicialmente en vender frascos de mermelada y frascos de dulces tradicionales. Todos los frascos deben tener el sabor, la cantidad y la fecha de elaboración, como hasta el momento, pero además las mermeladas deben tener la cantidad de azúcar que poseen, y los dulces tradicionales deben tener las instrucciones de conservación y una indicación de si tienen o no fibra.

El diagrama de clases correspondiente es:



Y el código en Java de estas clases, es:

```

import java.time.LocalDate;

/**
 * Un frasco con delicioso dulce, elaborado por la abuela pata.
 * Por ejemplo, un frasco de dulce de guayaba.
 * @author Sandra V. Hurtado
 * @version 3.5
 */
public class FrascoDeDulce
{
    private String sabor;
    private int cantidad;
    private LocalDate fecha;

    /**
     * Constructor de nuevos objetos FrascoDeDulce, sin
     * definir todavía la cantidad ni la fecha.
     * @param sabor el sabor del dulce, por ejemplo, "brevas"
     */
    public FrascoDeDulce(String sabor)
    {
        this.sabor = sabor;
        this.cantidad = 0;
        this.fecha = null;
    }

    /**
     * Envasar o insertar dulce en el frasco (el total: 200 gramos).
     * @param fecha la fecha en la cual se envasa el dulce
     */
    public void envasarDulce(LocalDate fecha)
    {
        this.fecha = fecha;
        this.cantidad = 200;
    }

    /**
     * Envasar o insertar dulce en el frasco, hasta máximo 200 gramos.
     * @param cantidadEnvasar la cantidad, en gramos, para envasar
     * @param fecha la fecha en la cual se envasa el dulce
     * @return una indicación (true/false) de si se pudo envasar o no
     * toda la cantidad deseada o solo 200 gramos
     */
    public boolean envasarDulce(int cantidadEnvasar, LocalDate fecha)
    {
        this.fecha = fecha;
        if (cantidadEnvasar > 200)
        {
            this.cantidad = 200;
            return false;
        }
    }
}

```

*//continúa*

```

        else
        {
            this.cantidad = cantidadEnvasar;
            return true;
        }
    }

    /**
     * Saca cierta cantidad del dulce del frasco, si hay suficiente
     * en el frasco; pero si hay menos solo se puede sacar lo que queda.
     * @param cantidadSacar la cantidad, en gramos, para sacar
     * @return la cantidad de dulce que se pudo sacar, en gramos
     */
    public int sacarDulce(int cantidadSacar)
    {
        int cantidadSacada = 0;
        if (this.cantidad >= cantidadSacar)
        {
            this.cantidad = this.cantidad - cantidadSacar;
            cantidadSacada = cantidadSacar;
        }
        else
        {
            cantidadSacada = this.cantidad;
            this.cantidad = 0;
        }
        return cantidadSacada;
    }

    public int getCantidad()
    {
        return this.cantidad;
    }

    public String getSabor()
    {
        return sabor;
    }

    public LocalDate getFecha()
    {
        return fecha;
    }

} // fin clase FrascoDeDulce

```

```

/**
 * Un frasco de dulce tradicional, que es la especialidad
 * de dulces elaborados por la abuela Pata.
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class FrascoDulceTradicional extends FrascoDeDulce
{
    private String instrucciones;
    private boolean fibra;

    /**
     * Constructor de un frasco de dulce tradicional,
     * inicialmente vacío y sin fecha de elaboración
     * @param sabor el sabor del dulce, por ejemplo: brevas.
     * @param instrucciones las instrucciones de conservación
     * @param fibra indicación de si tiene o no fibra natural
     */
    public FrascoDulceTradicional(String sabor,
                                   String instrucciones, boolean fibra)
    {
        super(sabor);
        this.instrucciones = instrucciones;
        this.fibra = fibra;
    }

    public String getInstrucciones()
    {
        return instrucciones;
    }

    public boolean getFibra()
    {
        return fibra;
    }
} // fin clase FrascoDulceTradicional

```

```

/**
 * Un frasco de mermelada, que es uno de los tipos de dulces
 * que elabora la abuela Pata.
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class FrascoMermelada extends FrascoDeDulce
{
    private int azucar;

```

*//continúa*

```
/**
 * Constructor de un frasco de mermelada,
 * inicialmente vacío y sin fecha de elaboración
 * @param sabor el sabor de la mermelada, por ejemplo: piña
 * @param azucar la cantidad (en gramos) de azúcar
 */
public FrascoMermelada(String sabor, int azucar)
{
    super(sabor);
    this.azucar = azucar;
}

public int getAzucar()
{
    return this.azucar;
}
} // fin clase FrascoMermelada
```

## 16. Polimorfismo por Sobrescritura

En este capítulo se presentará el concepto de polimorfismo por sobrescritura, que es otro de los tipos de polimorfismo que se puede tener en la programación orientada a objetos, y que permite utilizar algunas de las ventajas de la herencia.

A finalizar este capítulo usted debe ser capaz de:

- Identificar posibles jerarquías de clases donde aplique el polimorfismo por sobrescritura.
- Representar la sobrescritura de métodos en un diagrama de clases.
- Escribir métodos que sobrescriban otros heredados de una clase padre.

### 16.1. Definición

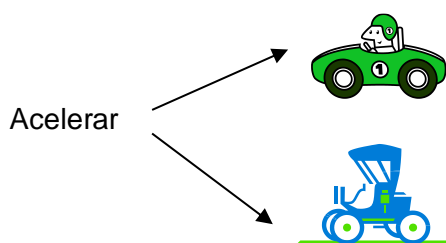
Una de las características importantes de los objetos, y en general de la Programación Orientada a Objetos (POO), es la capacidad de comportarse diferente ante un mismo mensaje, dependiendo de los parámetros que recibe o del tipo de objeto que sea. Esto se conoce como **polimorfismo**.

Esta capacidad de comportarse diferente puede darse en dos casos:

- Cuando la información que se le da al objeto para solicitarle el servicio cambia. Es decir, se envían diferente tipo y cantidad de parámetros a un método. Esto se conoce como **polimorfismo por sobrecarga**.
- Cuando se pide el mismo servicio a diferentes objetos, y cada uno de ellos puede hacerlo de forma diferente. Esto se conoce como **polimorfismo por sobrescritura**.

El polimorfismo por sobrescritura se presenta cuando se tienen objetos que son especialización (hijos) de otro, y cada uno se comporta de forma diferente cuando se les pide el mismo método.

Por ejemplo, si se le pide a un carro de carreras que acelere, lo hace de manera casi inmediata y alcanza una gran velocidad, pero si se le pide a un carro clásico que acelere, lo hará lentamente y no alcanzará una gran velocidad.



*Imagen 16-1 Comportamiento “acelerar” con diferentes objetos*

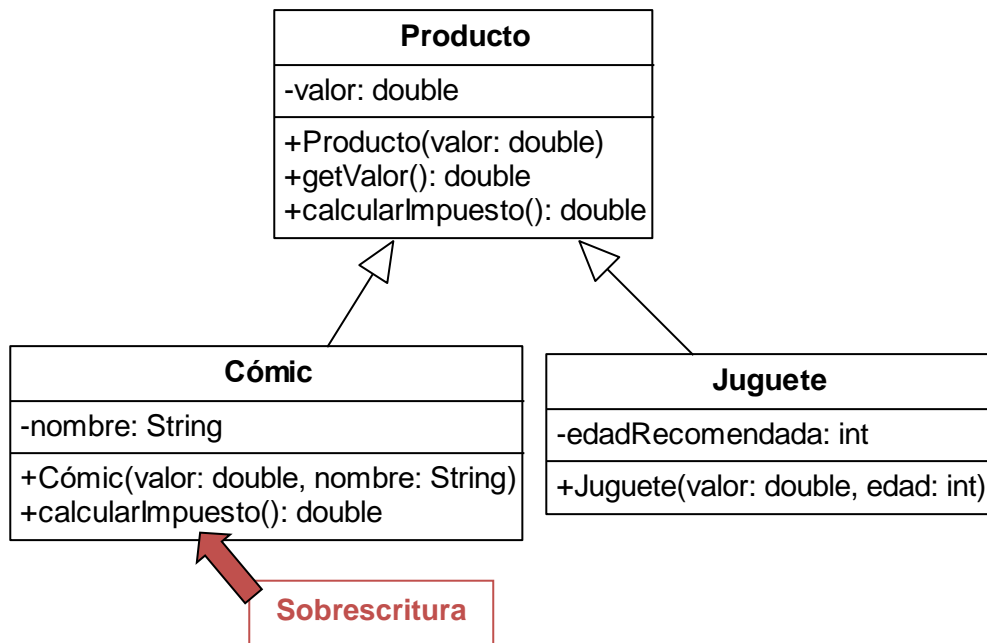
En este caso es el mismo servicio (“acelerar”) solicitado a dos objetos que son carros. La diferencia está en que, aunque ambos son carros, uno es un carro de carreras y el otro es un carro clásico, es decir, son especializaciones o tipos de carro.



## 16.2. Implementación en Java

En Java el polimorfismo por sobrescritura se logra cuando se tiene un método definido en la clase padre y las clases hijas **definen de nuevo** este método para darle un comportamiento diferente al que estaba en la clase padre.

Por ejemplo, se tiene el siguiente diagrama de clases de una tienda de variedades:



En la clase **Producto** se define un método llamado “calcularImpuesto”, que calcula el impuesto de venta que se debe aplicar a los productos que se venden en la tienda. Normalmente el impuesto corresponde al 14 % del valor del producto. Sin embargo, los comics se consideran un producto de lujo, por lo que pagan 16 % de impuesto.

En la clase **Producto** el método “calcularImpuesto” está definido para calcular el impuesto con el 14 % y todas sus clases hijas lo heredan. Sin embargo, la clase hija **Cómic**, como calcula diferente el impuesto, define de nuevo el método (**con el mismo encabezado**) y se queda con esta nueva definición.

De acuerdo con el diagrama anterior:

- La clase **Cómic** **sobrescribe** el método
- La clase **Juguete** NO sobrescribe el método, es decir, se queda con el mismo que hereda de la clase padre porque así le sirve.

El código para este diagrama es:

```

/**
 * Elemento que se vende en una tienda de variedades,
 * y al cual se le calcula el impuesto a partir del valor.
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Producto
{
    private double valor;

    /**
     * Constructor de objetos Producto
     * @param valor el valor del producto, en pesos
     */
    public Producto(double valor)
    {
        this.valor = valor;
    }

    public double getValor()
    {
        return valor;
    }

    /**
     * Calcula el impuesto del producto.
     * Normalmente el impuesto es el 14% del valor del producto.
     * @return el impuesto correspondiente para el producto
     */
    public double calcularImpuesto()
    {
        double impuesto = valor * 0.14;
        return impuesto;
    }
}

```

```

/**
 * Elemento para jugar, que se vende en la tienda de variedades
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Juguete extends Producto
{
    private int edadRecomendada;

```

*//continúa*

```

/**
 * Constructor para objeto Juguete, que recibe:
 * @param valor el valor del juguete, en pesos
 * @param edad la edad recomendada para el juguete
 */
public Juguete(double valor, int edad)
{
    super(valor);
    this.edadRecomendada = edad;
}
}

```

```

/**
 * Revista de historietas que se vende en la tienda
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Comic extends Producto
{
    private String nombre;

    /**
     * Constructor para objetos Cómic, con los siguientes datos:
     * @param valor el valor del cómic, en pesos
     * @param nombre el nombre del cómic
     */
    public Comic(double valor, String nombre)
    {
        super(valor);
        this.nombre = nombre;
    }

    /**
     * Calcula el impuesto, que en este caso es el 16% del valor
     * (por ser un bien de lujo).
     * @return el valor del impuesto que se debe pagar
     */
    @Override
    public double calcularImpuesto()
    {
        double impuesto = getValor()*0.16;
        return impuesto;
    }
}

```



Anotación que indica  
que el método se  
sobrescribe

La anotación `@Override` ayuda a prevenir errores porque permite que el compilador de Java revise que el encabezado del método coincida con el que está definido en la clase papá.

Ahora, se tiene el siguiente código correspondiente a la clase Tienda, con el método *main*, donde se crean dos productos (un juguete y un cómic) con el mismo valor y se calcula el impuesto de cada uno:

```
/**
 * Lugar donde se venden productos variados como juguetes y cómics
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Tienda
{
    public static void main(String[] args)
    {
        Producto productoA = new Comic(1000, "Superhéroes");
        Producto productoB = new Juguete(1000, 3);
        System.out.println("Impuesto A: " + productoA.calcularImpuesto());
        System.out.println("Impuesto B: " + productoB.calcularImpuesto());
    }
}
```

La salida es:

|  |
|--|
| Impuesto A: 160.0<br>Impuesto B: 140.0 |
|--|

Puede verse como los objetos, aunque tienen el mismo valor y se les llama el mismo método, se comportan de forma diferente, por la sobrescritura.

Incluso aunque los objetos son referenciados por una variable de la clase papá, Java utiliza el método que corresponda a cada objeto. Es como “engañar” a Java, porque el método está definido en la clase papá y por lo tanto se puede usar con la variable de tipo Producto, pero como el objeto es realmente un Cómic, entonces se usa la nueva definición que estableció esta clase. En cuanto al objeto Juguete, se usa el método definido en la clase padre, dado que lo hereda y no lo cambia.

### **Ejercicio 16-1**

Elabore el diagrama de clases y el código en Java, para el siguiente enunciado, haciendo uso de polimorfismo por sobrescritura:

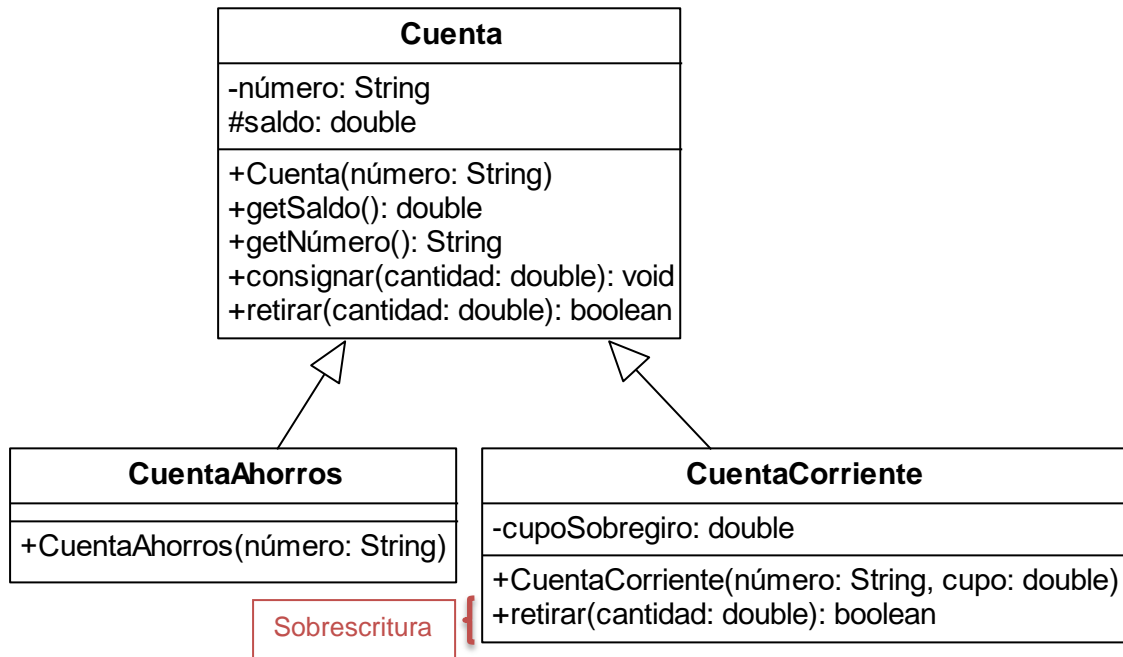
En la empresa ECME desean un programa que les ayude a calcular el valor que deben pagar mensualmente a sus empleados. En la empresa tienen empleados con contrato y empleados por prestación de servicios. Cada empleado, sin importar su modalidad de contratación, tiene definido un salario mensual.

El pago a todos los empleados se realiza teniendo en cuenta el salario mensual, pero descontando un 8 % para salud y pensiones. Además, a los empleados por prestación de servicios se les descuenta un 2 % adicional por retención en la fuente.

## **16.3. Ejercicio resuelto**

En el banco MuchoDinero ya tienen cuentas corrientes y cuentas de ahorros, pero el método “retirar” es diferente para cada una.

Conociendo el polimorfismo por sobrescritura, se ha considerado incluir el método “retirar” en la clase padre, y que la clase CuentaCorriente lo sobrescriba considerando el cupo de sobregiro que puede tener. El diagrama de clases y el código quedan:



```

/**
 * Una cuenta bancaria, es decir, un registro en
 * una entidad bancaria que tiene un saldo (cantidad de dinero)
 * que se puede modificar mediante retiros o consignaciones.
 * @author Sandra V. Hurtado
 * @version 4.0
 */
public class Cuenta
{
    private String numero;
    protected double saldo;

    /**
     * Constructor de objetos Cuenta, con saldo inicial cero.
     * @param numero el número de la cuenta
     */
    public Cuenta(String numero)
    {
        this.numero = numero;
        this.saldo = 0;
    }

    public double getSaldo()
    {
        return this.saldo;
    }
}

```

//continúa

```

public String getNumero()
{
    return this.numero;
}

/**
 * Consigna o adiciona una cantidad de dinero en la cuenta,
 * lo cual incrementa el saldo.
 * @param cantidad la cantidad de dinero que se desea
 *                consignar, en pesos
 */
public void consignar(double cantidad)
{
    saldo = saldo + cantidad;
}

/**
 * Retira o saca una cantidad de dinero de la cuenta,
 * si hay saldo suficiente.
 * @param cantidad la cantidad que se desea retirar, en pesos.
 * @return si se pudo hacer el retiro porque tenía dinero
 *         suficiente o no (true o false)
 */
public boolean retirar(double cantidad)
{
    if (this.saldo >= cantidad)
    {
        this.saldo = this.saldo - cantidad;
        return true;
    }
    return false;
}
} // fin clase Cuenta

```

```

/**
 * Cuenta corriente en una entidad financiera,
 * en la cual se puede retirar incluso si no hay saldo,
 * hasta el monto dado por el cupo de sobregiro.
 * @author Sandra V. Hurtado
 * @version 1.2
 */
public class CuentaCorriente extends Cuenta
{
    private double cupoSobregiro;

```

*//continúa*

```

/**
 * Constructor de objetos cuenta corriente
 * @param numero el número de la cuenta
 * @param cupo el máximo que puede retirar
 *             después de tener saldo cero
 */
public CuentaCorriente(String numero, double cupo)
{
    super(numero);
    cupoSobregiro = cupo;
}

/**
 * Retira o saca una cantidad de dinero de la cuenta,
 * incluso quedando negativo, hasta el cupo de sobregiro.
 * @param cantidad la cantidad que se desea retirar, en pesos.
 * @return si se pudo hacer el retiro porque tenía dinero
 *         suficiente o no (true o false)
 */
@Override
public boolean retirar(double cantidad)
{
    if ((this.saldo + cupoSobregiro) >= cantidad)
    {
        this.saldo -= cantidad;
        return true;
    }
    return false;
}
} // fin clase CuentaCorriente

```

```

/**
 * Cuenta de ahorros en una entidad financiera,
 * donde solo se puede retirar dinero hasta el saldo.
 * @author Sandra V. Hurtado
 * @version 1.3
 */
public class CuentaAhorros extends Cuenta
{
    /**
     * Constructor de objetos cuenta de ahorros
     * @param numero el número de la cuenta
     */
    public CuentaAhorros(String numero)
    {
        super(numero);
    }
}

```

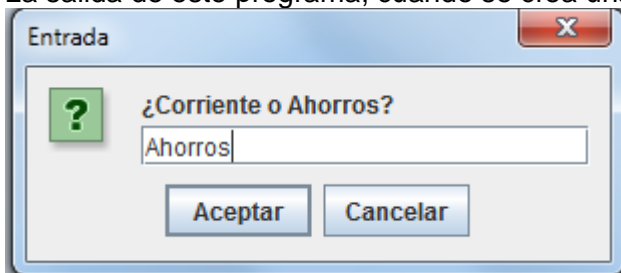
Ahora el banco podrá manejar las cuentas de manera uniforme. En el siguiente código se muestra la creación de una cuenta, a la cual se le consignan \$350.000 y se retira \$120.000. Solo al momento de crear la cuenta se debe saber si es de ahorros o corriente, las demás operaciones se hacen con una variable de tipo Cuenta (la clase padre).

```
/**
 * Clase donde se crean una cuenta bancaria (corriente o de ahorros),
 * se consigna dinero y luego se retira
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class CreaCuenta
{
    public static void main(String[] args)
    {
        Cuenta bancaria;
        String tipoCuenta =
            JOptionPane.showInputDialog("¿Corriente o Ahorros?");
        String numeroCuenta =
            JOptionPane.showInputDialog("Número cuenta:");
        if (tipoCuenta.equalsIgnoreCase("Corriente"))
        {
            String cupoCadena =
                JOptionPane.showInputDialog("Cupo sobregiro");
            double sobregiro = Double.parseDouble(cupoCadena);
            bancaria = new CuentaCorriente(numeroCuenta,sobregiro);
        }

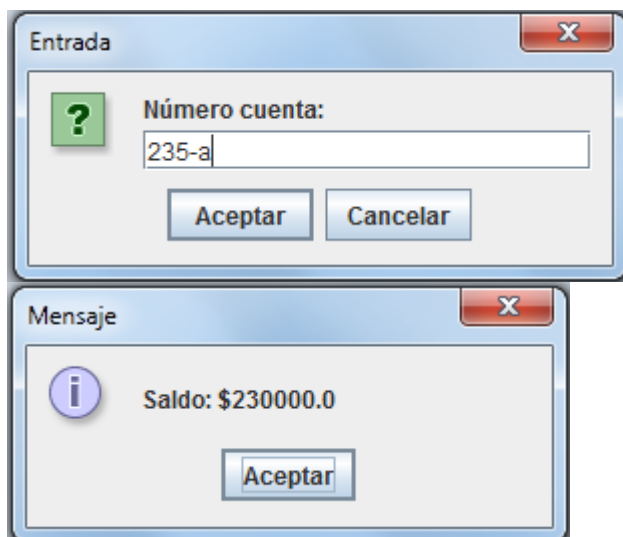
        else
        {
            bancaria = new CuentaAhorros(numeroCuenta);
        }

        // Se consigna y se retira el dinero
        bancaria.consignar(350000);
        bancaria.retirar(120000);
        double saldo = bancaria.getSaldo();
        JOptionPane.showMessageDialog(null, "Saldo: $" + saldo);
    }
}
```

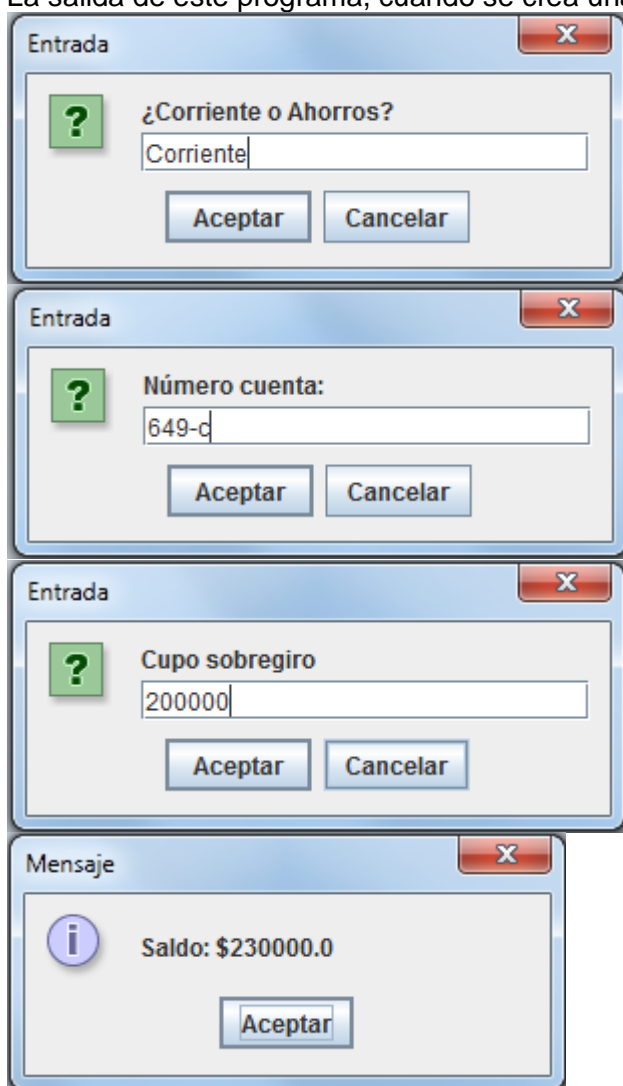
La salida de este programa, cuando se crea una cuenta de ahorros:







La salida de este programa, cuando se crea una cuenta corriente:



## 17. Conversión de variables (Cast)

En este capítulo se presentará el concepto de “cast” o conversión forzada de variables, aplicado a una jerarquía de clases.

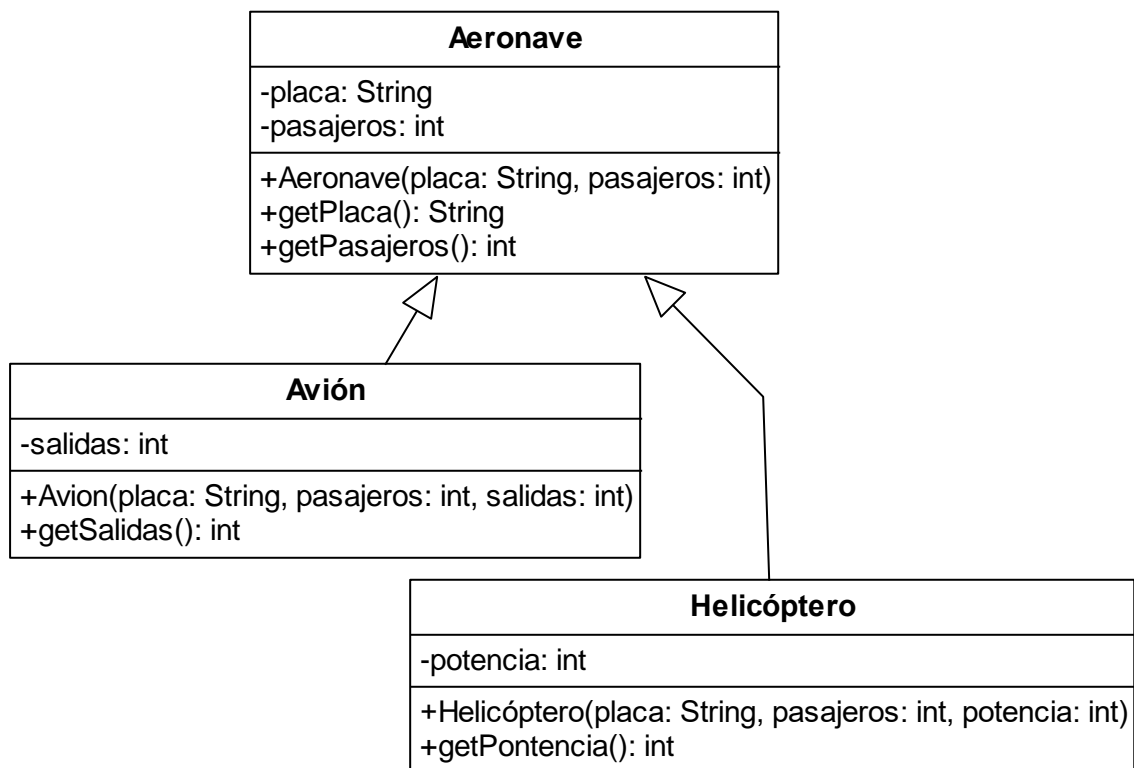
A finalizar este capítulo usted debe ser capaz de:

- Usar el operador “cast” para realizar una conversión de variables.
- Usar el operador *instanceof* para saber si un objeto es de una clase en particular.

### 17.1. Problemática métodos de clases hijas

Una de las ventajas de la herencia es que simplifica la forma de trabajar con objetos diferentes, haciendo uso de las clases padre.

Por ejemplo, se tiene el siguiente diagrama de clases:



Es posible crear objetos **Avión** u objetos **Helicóptero**, y referenciarlos con variables de tipo **Aeronave**. Por ejemplo, se creará un avión y se mostrará su información básica, y luego se usará la misma variable para referenciar un helicóptero y mostrar su información:

```

Aeronave aeronave;

aeronave = new Avion("ZAX-98",200,5);
System.out.println("La aeronave 1 tiene placa: "+aeronave.getPlaca()+
    " y capacidad: "+aeronave.getPasajeros());

aeronave = new Helicoptero("11-HEX",2,1);
System.out.println("La aeronave 2 tiene placa: "+aeronave.getPlaca()+
    " y capacidad: "+aeronave.getPasajeros());

```

La salida de este programa es:

```

La aeronave 1 tiene placa: ZAX-98 y capacidad: 200
La aeronave 2 tiene placa: 11-HEX y capacidad: 2

```

Sin embargo, si se desea mostrar información específica de cada clase hija, no funciona con la variable Aeronave. Por ejemplo, el siguiente código muestra un error:

```

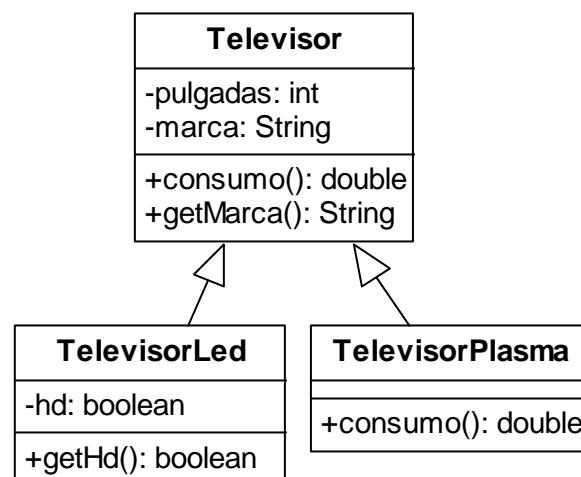
Aeronave voladora = new Avion("ZAW-76",150,4);
System.out.println("Salidas: "+voladora.getSalidas()); //ERROR

```

Aunque es un objeto de la clase Avión, la variable solo permite usar los métodos definidos en la clase Aeronave.

### **Ejercicio 17-1**

Se tiene un diagrama de clases que representa una jerarquía de televisores:



Indique cuáles de las siguientes son instrucciones válidas y cuáles no:

```

Televisor tvPlasma = new TelevisorPlasma();
double consumo = tvPlasma.consumo();
Televisor tvLed = new TelevisorLed();
boolean hd = tvLed.getHd();

```

Para poder usar los métodos definidos únicamente en una clase hija es necesario que la variable sea de esa clase, por ejemplo:

```
Avion avion = new Avion("ZAW-76",150,4);
System.out.println("Salidas: "+avion.getSalidas());
```

Sin embargo, de esta manera se pierde la posibilidad de trabajar todas las aeronaves de manera uniforme, y el programa se puede hacer más extenso, porque se deben tener variables diferentes para cada aeronave.

Se puede pensar en tener la variable de tipo Aeronave, y solo asignar el objeto a una variable de tipo Avión cuando se desee usar el método “getSalidas”. El código sería:

```
Aeronave voladora = new Avion("ZAW-76",150,4);
System.out.println("La aeronave tiene placa: "+voladora.getPlaca()+
                  " y capacidad: "+voladora.getPasajeros());
Avion avion = voladora;           // ERROR
System.out.println("Salidas: "+avion.getSalidas());
```

Sin embargo, esto genera un error, porque no es posible asignar directamente un objeto Aeronave a una variable de tipo Avión.

## 17.2. Cast

La solución en este caso es usar la operación de “**cast**” (conversión de variables) que tiene Java. Mediante esta operación es posible forzar la asignación, es decir, indicar a Java que sí se desea que una variable diferente referencie al objeto.

La sintaxis es:

```
TipoVariableDeseado nombre = (TipoVariableDeseado) objeto;
```

En este caso se desea que una variable Avión referencie el objeto que antes estaba con una variable Aeronave. La instrucción es:

```
Avion avion = (Avion) voladora;
```



Con esta operación ya se puede usar la variable avión y mostrar la información. El código queda:

```
Aeronave voladora = new Avion("ZAW-76",150,4);
System.out.println("La aeronave tiene placa: "+voladora.getPlaca()+
                  " y capacidad: "+voladora.getPasajeros());
Avion avion = (Avion) voladora;
System.out.println("Salidas: "+avion.getSalidas());
```

La salida es:

```
La aeronave tiene placa: ZAW-76 y capacidad: 150
Salidas: 4
```

Hay que tener en cuenta que lo que se está haciendo **NO es convertir el objeto sino asignarlo a una variable más adecuada.**

Es importante verificar que el “cast” sea realmente posible, pues solo se debe hacer cuando el objeto realmente cumple con la definición. En el ejemplo anterior se podía hacer el “cast” porque el objeto era un objeto de tipo Avión, y por lo tanto referenciarlo con la variable Avión no tenía problema. Sin embargo, por descuido se puede intentar hacer un “cast” de objetos que no son compatibles, por ejemplo, de un Helicóptero a un Avión, y eso generaría errores inesperados.

El siguiente código, por ejemplo, es incorrecto:

```
Aeronave nave = new Helicoptero("12-HFX",6,2);
Avion unAvion = (Avion) nave;
```

**¡Incorrecto!** Un helicóptero no es un avión

### 17.3. Operador *instanceof*

Para validar si un objeto cumple con la definición de una clase se usa el operador *instanceof*. Este operador permite preguntar si un objeto es instancia de alguna clase en particular. Es un operador lógico, es decir, retorna un valor booleano (*true* o *false*) y por lo general se usa en un *if*.

La sintaxis es:

```
if (objeto instanceof Clase)
{
    // usar los métodos que se necesitan
}
```

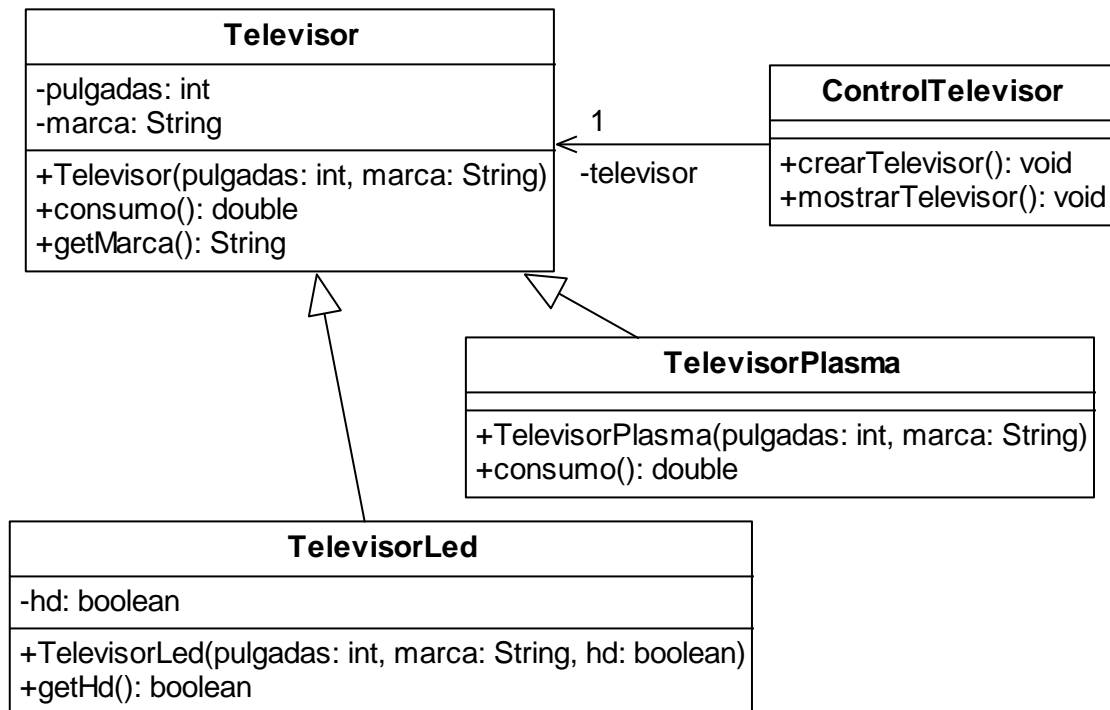
Con este operador ya se puede usar el “cast” con seguridad, para no cometer errores. El código final queda:

```
Aeronave voladora = new Avion("ZAW-76",150,4);
System.out.println("La aeronave tiene placa: "+voladora.getPlaca()+
    " y capacidad: "+voladora.getPasajeros());
if (voladora instanceof Avion)
{
    Avion avion = (Avion) voladora;
    System.out.println("Salidas: "+avion.getSalidas());
}
```

### **Ejercicio 17-2**

Escriba el código Java para el diagrama que se presenta a continuación, teniendo en cuenta los siguientes lineamientos:

- El consumo de cada televisor es 1.5 veces las pulgadas que tiene. Por ejemplo, si tiene 30 pulgadas, el consumo es 45. Sin embargo, los televisores de plasma tienen un consumo mayor: 2 veces las pulgadas que tienen. Por ejemplo, si es un televisor de plasma de 30 pulgadas, el consumo es 60.
- En el método “crearTelevisor” se deben pedir al usuario los datos necesarios: qué tipo de televisor desea crear, las pulgadas, la marca y si es HD o no (esto último solo para televisores LED). Con estos datos se debe crear el objeto correspondiente y asignarlo al atributo “televisor”.
- En el método “mostrarTelevisor” se deben mostrar los siguientes datos del televisor: la marca, el consumo y si es HD o no (esto último solo si es un televisor LED).

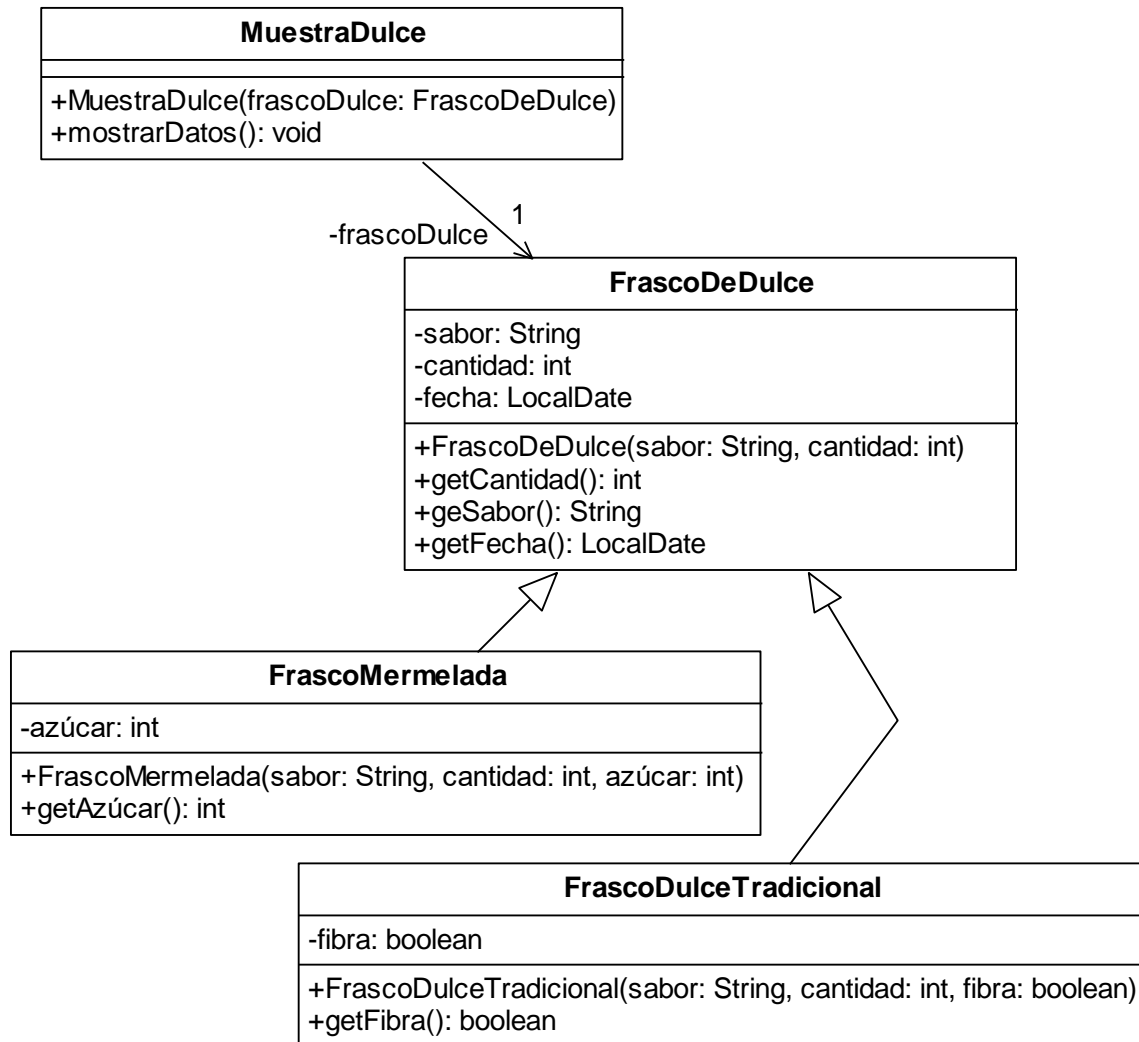


## 17.4. Ejercicio resuelto

Los nietos de la abuela Pata son muy curiosos, y desean ver toda la información de un frasco de dulce, después de que la abuela lo ha creado. Ellos desean conocer el sabor del dulce, la cantidad, la fecha de envasado y también la cantidad de azúcar (en el caso de las mermeladas) o si tiene fibra o no (en el caso de los dulces tradicionales).

El sabor, la cantidad y la fecha son atributos comunes a todos los frascos de dulce, pero la cantidad de azúcar y la indicación de si tiene fibra son propios de las clases hijas, por lo que se hará uso del “cast” para mostrar esta información.

Se muestra el diagrama de clases, donde se simplifican un poco los métodos y atributos, para hacer énfasis en el problema presentado. Posteriormente se muestra el código Java correspondiente, donde puede verse el uso del “cast”, en la clase MuestraDulce.



```
import java.time.LocalDate;
```

```
/**
```

```
 * Un frasco con delicioso dulce, elaborado por la abuela pata.
```

```
 * Por ejemplo, un frasco de dulce de guayaba.
```

```
 * @author Sandra V. Hurtado
```

```
 * @version 3.8
```

```
 */
```

```
public class FrascoDeDulce
```

```
{
```

```
    private String sabor;
```

```
    private int cantidad;
```

```
    private LocalDate fecha;
```

```
//continúa
```

```

/**
 * Constructor de nuevos objetos FrascoDeDulce,
 * sin fecha de envasado definida.
 * @param sabor el sabor del dulce, por ejemplo, "brevas"
 * @param cantidad la cantidad, en gramos, para envasar
 */
public FrascoDeDulce(String sabor, int cantidad)
{
    this.sabor = sabor;
    this.cantidad = cantidad;
    this.fecha = null;
}

public int getCantidad()
{
    return this.cantidad;
}

public String getSabor()
{
    return sabor;
}

public LocalDate getFecha()
{
    return fecha;
}
} // fin clase FrascoDeDulce

```

```

/**
 * Un frasco de mermelada, que es uno de los tipos de dulces
 * que elabora la abuela Pata.
 * @author Sandra V. Hurtado
 * @version 1.5
 */
public class FrascoMermelada extends FrascoDeDulce
{
    private int azucar;

    /**
     * Constructor de un frasco de mermelada
     * @param sabor el sabor de la mermelada, por ejemplo: piña
     * @param cantidad la cantidad, en gramos, para envasar
     * @param azucar la cantidad (en gramos) de azúcar
     */
    public FrascoMermelada(String sabor, int cantidad, int azucar)
    {

```

*//continúa*



```

        super(sabor,cantidad);
        this.azucar = azucar;
    }

    public int getAzucar()
    {
        return this.azucar;
    }
} // fin clase FrascoMermelada

```

```

/**
 * Un frasco de dulce tradicional, que es la especialidad
 * de dulces elaborados por la abuela Pata.
 * @author Sandra V. Hurtado
 * @version 1.5
 */
public class FrascoDulceTradicional extends FrascoDeDulce
{
    private boolean fibra;

    /**
     * Constructor de un frasco de dulce tradicional.
     * @param sabor    el sabor del dulce, por ejemplo: brevas.
     * @param cantidad la cantidad, en gramos, para envasar
     * @param fibra    indicación de si tiene o no fibra natural
     */
    public FrascoDulceTradicional(String sabor,
                                   int cantidad, boolean fibra)
    {
        super(sabor,cantidad);
        this.fibra = fibra;
    }

    public boolean getFibra()
    {
        return fibra;
    }
}

```

```

/**
 * Clase que tiene un frasco de dulce, del cual muestra los datos
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class MuestraDulce
{

```

*//continúa*

```

private FrascoDeDulce frascoDulce;

/**
 * Constructor de un objeto MuestraDulce
 * @param frascoDulce el frasco del que mostrarán los datos
 */
public MuestraDulce(FrascoDeDulce frascoDulce)
{
    this.frascoDulce = frascoDulce;
}

/**
 * Muestra toda la información de un frasco de dulce.
 * Para mostrar los atributos de las clases hijas usa cast
 */
public void mostrarDatos()
{
    System.out.println("Sabor: "+frascoDulce.getSabor());
    System.out.println("Cantidad: "+frascoDulce.getCantidad());
    System.out.println("Fecha: "+frascoDulce.getFecha());

    if (frascoDulce instanceof FrascoMermelada)
    {
        FrascoMermelada frasco = (FrascoMermelada)frascoDulce;
        System.out.println("Azúcar: "+frasco.getAzucar());
    }

    if (frascoDulce instanceof FrascoDulceTradicional)
    {
        FrascoDulceTradicional frasco =
            (FrascoDulceTradicional)frascoDulce;
        System.out.println("Fibra: "+frasco.getFibra());
    }
}
} // fin clase MuestraDulce

```

Uso de cast

## 18. Atributos y Métodos de Clase (Static)

En este capítulo se presentará el concepto de atributos y métodos de clase o estáticos (por la palabra “static” en inglés), que pueden ser usados esporádicamente para manejar valores propios de todo un conjunto de objetos y no de cada uno en particular.

A finalizar este capítulo usted debe ser capaz de:

- Representar clases y métodos de clase en un diagrama de clases.
- Escribir el código Java correspondiente a clases y métodos de clase.
- Usar un método de clase.

### 18.1. Contexto

Se tiene el siguiente diagrama de clases que representa un préstamo bancario:

| Préstamo   |
|--|
| -valor: double<br>-interés: double<br>-plazo: int  |
| +Préstamo(valor: double, plazo: int)<br>+setInterés(interés: double): void<br>+getInterés(): double<br>+cuotaMes(): double |

Se plantea la siguiente restricción: el interés es el mismo para todos los préstamos, es decir, no se pueden tener préstamos con intereses diferentes. Si el interés cambia, esto aplica para todos los préstamos. En este caso surge entonces una inquietud: ¿Cómo se puede modelar esto?

Lo que se busca es definir un atributo que no sea de los objetos, sino de la clase. Si se define un atributo “normal” (de instancia o de los objetos), cada objeto puede tener un valor diferente, es decir, se pueden tener múltiples valores en un momento dado. Pero si se declara un atributo de la clase este será un valor único para esta clase, no para cada objeto.

### 18.2. Atributos y métodos de clase (estáticos)

La forma de declarar que un atributo o un método son de la clase es con el modificador **static**. Este modificador hace que ese atributo o método NO pertenezca a cada objeto, sino que se tenga un solo valor para la clase.

Por ejemplo, en la clase Préstamo se definirá un atributo estático para guardar el valor del interés que aplica para todas estas cuentas, y también los métodos estáticos que permiten cambiar o consultar el valor de este atributo.

Esto se representa en el diagrama subrayando el atributo y los métodos que sean estáticos, así:

| Préstamo  |
|---|
| -valor: double<br>-interés: double<br>-plazo: int   |
| +Préstamo(valor: double, plazo: int)<br>+setInterés(nuevoInterés: double): void<br>+getInterés(): double<br>+cuotaMes(): double |

El código Java correspondiente para esta clase es:

```

/**
 * Préstamo de dinero realizado por un banco, con su valor y plazo.
 * Todos los préstamos tienen siempre el mismo interés.
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Prestamo
{
    private static double interes;
    private double valor;
    private int plazo;

    /**
     * Constructor para crear objetos Préstamo
     * @param valor el valor total del préstamo
     * @param plazo el plazo, en meses, para el pago del préstamo
     */
    public Prestamo (double valor, int plazo)
    {
        this.valor = valor;
        this.plazo = plazo;
    }

    /**
     * Permite cambiar el interés
     * @param interes el nuevo valor para el interés
     */
    public static void setInteres(double nuevoInteres)
    {
        interes = nuevoInteres;
    }

    public static double getInteres()
    {
        return interes;
    }
}

```

Atributo de clase

Métodos de clase

//continúa

```

/**
 * Calcula el valor de la cuota (es algo ficticio).
 * @return el valor mensual de la cuota
 */
public double cuotaMensual()
{
    double cuota = (valor / plazo);
    cuota += cuota * Prestamo.interes;
    return cuota;
}
} // fin clase Prestamo

```

Como los atributos y métodos estáticos pertenecen a la clase y no a los objetos, no es necesario crear un objeto para poder usarlos, sino que se usan a través del nombre de la clase, por ejemplo:

```

// Ejemplo de uso de método de instancia (del objeto):
Prestamo prestado1 = new Prestamo(2500000,12);
double valorCuota = prestado1.cuotaMensual();

```

```

// Ejemplo de uso de método de clase (estático):
Prestamo.setInteres(0.5);

```

Se llama con la clase,  
no con el objeto

Es importante tener presente que los demás métodos y atributos de la clase siguen siendo de cada objeto y por lo tanto no pueden usarse con la clase.

```

// ESTO SERÍA ERROR: double valor = Prestamo.cuotaMensual();

```

Los atributos y métodos estáticos no son muy usados en la programación orientada a objetos porque no cumplen adecuadamente con los principios de este tipo de programación. Usar este tipo de atributos y métodos puede significar un análisis inadecuado.

### **Ejercicio 18-1**

Determine, buscando en la documentación del API de Java, cuáles de los siguientes métodos son de la clase (estáticos) o de los objetos (de instancia):

- El método “showMessageDialog” de la clase *JOptionPane*
- El método “nextInt” de la clase *Random*
- El método “equals” de la clase *String*
- El método “sqrt” de la clase *Math*

### **Ejercicio 18-2**

El siguiente código presenta un error (en la línea señalada). Explique por qué se presenta este error y cómo se podría solucionar.

```

public class Foto
{
    private int largo;

    public static int getValorLimite()
    {
        int valorLimite = largo + 10;           // aquí está el error
        return valorLimite;
    }
}

```

### 18.3. Método *main*

El *main* es un método especial, porque es un método de clase que Java reconoce como el método que inicia un programa.

El encabezado del método *main* es:

```

public static void main(String[] args)

```

Este encabezado no se puede cambiar, el método *main* debe ser público y estático, no retorna nada (*void*) y recibe un arreglo de cadenas como parámetro. Lo único que puede cambiar es el nombre del parámetro.

Este método puede estar en cualquier clase, pero se recomienda tener **un solo método *main*** en todo el programa, porque es el punto donde comienza.

### 18.4. Ejercicio resuelto

La abuela Pata desea que cada vez que crea un nuevo dulce se asigne como fecha de envasado la fecha de ese mismo día, es decir, la fecha que tenga el sistema.

Para poder lograr esto se busca en el API de Java los métodos que tiene la clase *LocalDate*, y se encuentra el siguiente método estático:

```

static LocalDate    now()
                     Obtains the current date from the system clock in the default time-
                     zone.

```

*Imagen 18-1 Descripción del método "now" de la clase LocalDate en la documentación del API de Java*

Este método retorna un objeto de la misma clase (*LocalDate*), que representa la fecha que tenga en el momento el sistema, que es precisamente lo que se necesita. Se usará este método dentro del constructo de *FrascoDeDulce*, para darle el valor al atributo fecha.

El diagrama de clases y el código para la clase *FrascoDeDulce* quedan:

| <b>FrascoDeDulce</b>  |
|---|
| -sabor: String<br>-cantidad: int<br>-fecha: LocalDate   |
| +FrascoDeDulce(sabor: String, cantidad: int)<br>+getCantidad(): int<br>+geSabor(): String<br>+getFecha(): LocalDate |

```

/**
 * Un frasco con delicioso dulce, elaborado por la abuela pata.
 * Por ejemplo, un frasco de dulce de guayaba.
 * @author Sandra V. Hurtado
 * @version 4.0
 */
public class FrascoDeDulce
{
    private String sabor;
    private int cantidad;
    private LocalDate fecha;

    /**
     * Constructor de nuevos objetos FrascoDeDulce,
     * tomando la fecha del sistema como fecha de envasado.
     * @param sabor el sabor del dulce, por ejemplo, "brevas"
     * @param cantidad la cantidad, en gramos, para envasar
     */
    public FrascoDeDulce(String sabor, int cantidad)
    {
        this.sabor = sabor;
        this.cantidad = cantidad;
        this.fecha = LocalDate.now();
    }

    public int getCantidad()
    {
        return this.cantidad;
    }

    public String getSabor()
    {
        return sabor;
    }

    public LocalDate getFecha()
    {
        return fecha;
    }
}

```

## 19. Clases y Métodos Abstractos

En este capítulo se presentará el concepto de clases y métodos abstractos, que contribuyen en gran medida a hacer uso de las ventajas de la herencia y del polimorfismo en Java. Al final se hace una breve mención a la palabra reservada *final*.

A finalizar este capítulo usted debe ser capaz de:

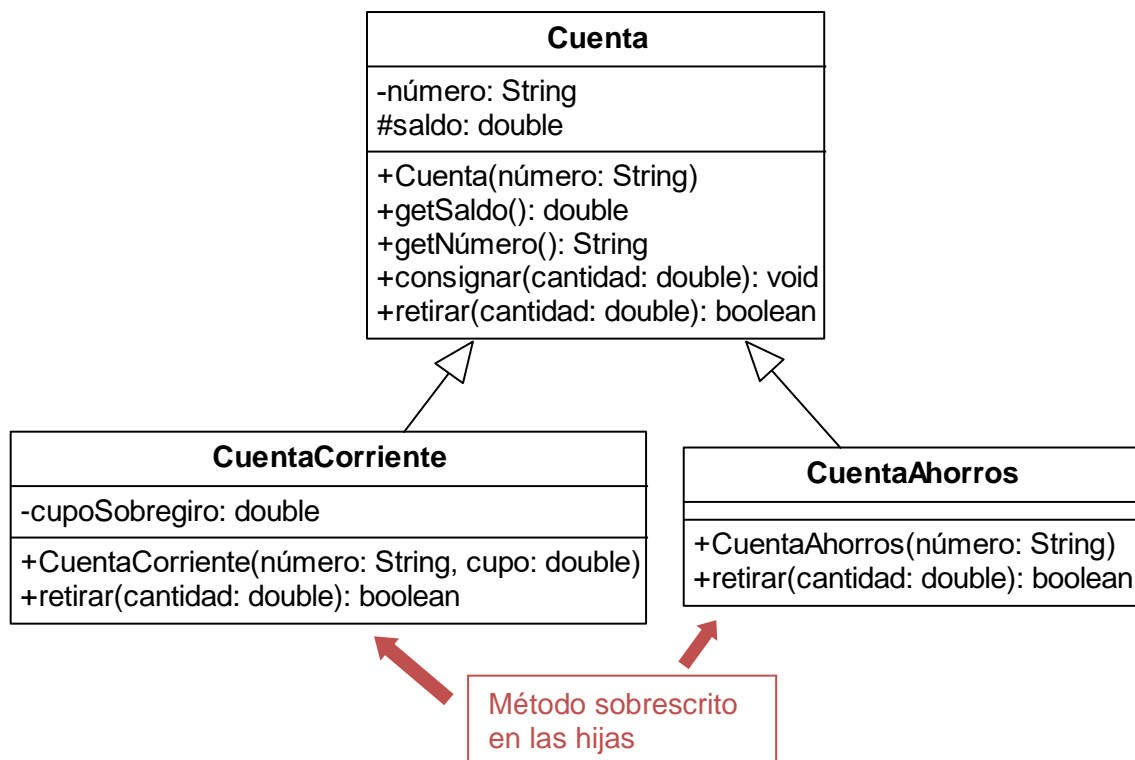
- Representar clases y métodos abstractos en un diagrama de clases.
- Escribir el código Java correspondiente a clases y métodos abstractos.

### 19.1. Problemática instancias clase padre

El banco MuchoDinero ofrece cuentas de ahorro y cuentas corrientes. De todas las cuentas guarda el número y el saldo, y, además, en las cuentas corrientes guarda la cantidad autorizada de sobregiro. Ahora el banco tiene una nueva regla: en los retiros de una cuenta de ahorro debe verificar que el saldo final no puede ser inferior a \$20.000.

El método “retirar” estaba definido en la clase padre y solo la clase CuentaCorriente lo sobrescribía, para tener en cuenta el cupo de sobregiro. Sin embargo, con esta nueva regla la clase CuentaAhorros también tiene que sobrescribir este método.

El diagrama de clases sería:



Incluso si más adelante se crean nuevos tipos de cuentas, cada una tendrá unas condiciones diferentes para retirar dinero. Esto lleva a pensar que el método “retirar” en la clase Cuenta ya no será usado directamente por ninguna de las clases hijas. Es posible



pensar en dejar el código de este método solo con lo mínimo necesario para que compile, por ejemplo:

```
/**
 * Retira o saca una cantidad de dinero de la cuenta
 * @param cantidad la cantidad que se desea retirar, en pesos.
 * @return si se pudo hacer el retiro o no
 */
public boolean retirar(double cantidad)
{
    return false;    // Solo retorna un valor, no hace nada
                    // porque las hijas lo sobrescribirán
}
```

Pero esto lleva a otro problema: ¿Qué pasaría si por error se crea un objeto de la clase Cuenta, y no de las clases CuentaCorriente y CuentaAhorros?

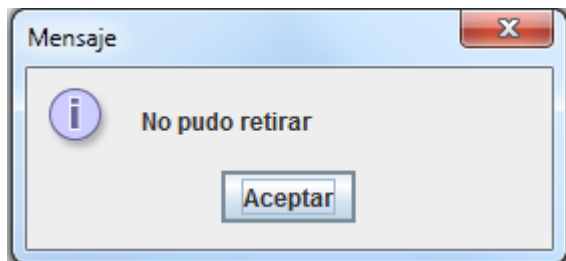
Por ejemplo, si se tiene el siguiente código donde se crea un objeto Cuenta, se consignan \$450.000 y se intenta retirar \$150.000:

```
import javax.swing.JOptionPane;

/**
 * Clase donde se crean una cuenta bancaria (clase padre)
 * se consigna dinero y luego se trata de retirar.
 * @author Sandra V. Hurtado
 * @version 1.5
 */
public class CreaCuenta
{
    public static void main(String[] args)
    {
        String numeroCuenta =
            JOptionPane.showInputDialog("Número cuenta:");
        Cuenta bancaria = new Cuenta(numeroCuenta);

        // Se consigna y se retira el dinero
        bancaria.consignar(450000);
        boolean pudoRetirar = bancaria.retirar(150000);
        if (pudoRetirar)
        {
            JOptionPane.showMessageDialog(null, "Pudo retirar");
        }
        else
        {
            JOptionPane.showMessageDialog(null, "No pudo retirar");
        }
    }
}
```

El resultado, después de ingresar el número de la cuenta, es:



El programa no funciona apropiadamente, pues no se puede retirar de una cuenta que no es corriente ni de ahorros, porque no se sabe cuáles reglas aplicar.

Este problema es causado por crear un objeto de la clase padre, y no de las clases hijas, que es lo esperado. Aunque en algunos programas sí es posible crear objetos de las clases padres, en este ejemplo del banco no es correcto.

## 19.2. Clases abstractas y su representación

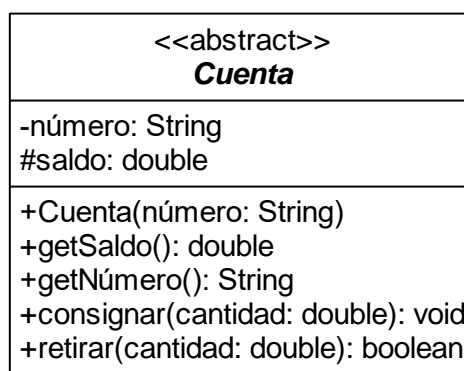
La solución es definir la clase Cuenta como una clase especial, que tiene atributos y métodos, pero de la cual no se desean crear instancias (objetos). Los atributos y métodos que define esta clase son usados solo en las clases hijas. Este tipo de clases se llaman clases abstractas.

Una **clase abstracta** es una clase de la cual no se pueden crear objetos o instancias; es decir, no se puede usar el operador *new* directamente para objetos de esa clase, pero sí se pueden crear objetos de sus clases hijas.

En el diagrama de clases, una clase abstracta se representa escribiendo antes del nombre de la clase la palabra *abstract*, encerrada en símbolos: << y >>.

En el caso de contar con alguna herramienta para elaborar los diagramas de clases, el nombre de las clases abstractas aparece en cursiva.

Por ejemplo, para el caso de la clase Cuenta:



Con la clase Cuenta definida como una clase abstracta, ya no puede crear por error un objeto de esta clase. El código que se tenía previamente ya no compilaría:

```
// Si Cuenta es una clase abstracta, esta instrucción no compila:
Cuenta bancaria = new Cuenta(numeroCuenta);    //ERROR
```

De esta manera se garantiza que el programa solo puede crear objetos de las clases CuentaCorriente y CuentaAhorros, para así funcionar adecuadamente.

### 19.3. Métodos abstractos y su representación

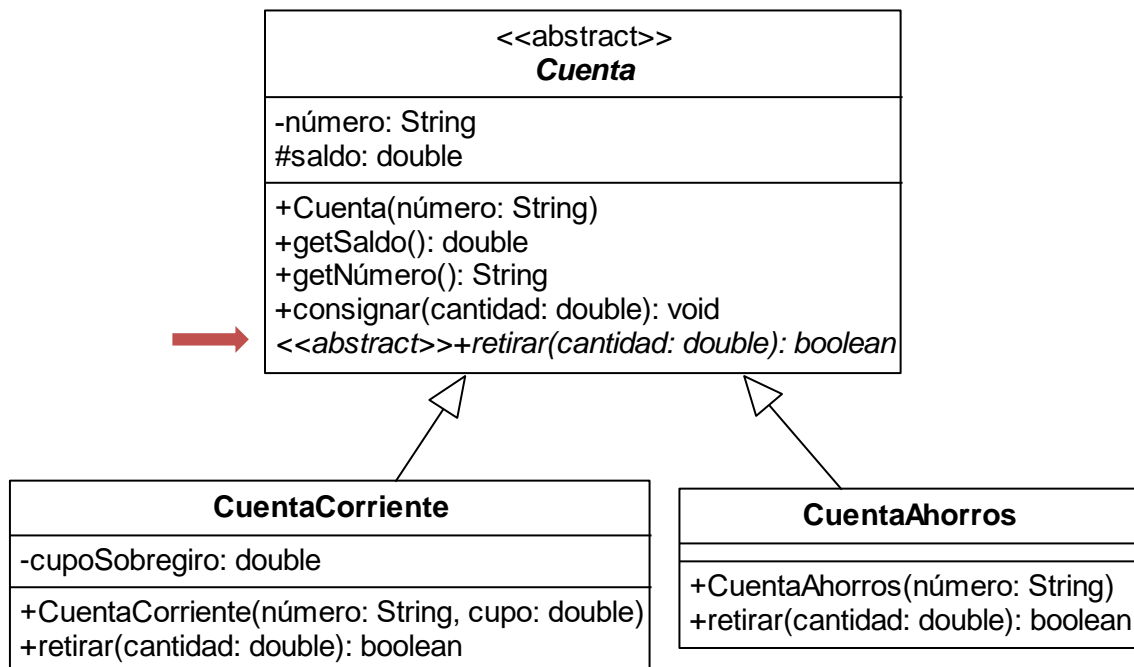
Una de las ventajas que tienen las clases abstractas es que pueden definir métodos que no tengan cuerpo, solo el encabezado, debido a que siempre serán sobrescritos por las clases hijas.

En el caso de la clase Cuenta, se tenía el método “retirar”, el cual se dejó solo con una instrucción para que compilara y para poder ser usado con variables de tipo Cuenta, pero sin que fuera la operación real de retirar. Este es un ejemplo de un método que no necesita tener código en la clase padre.

Un método que no tiene cuerpo, solo el encabezado, y que debe ser sobrescrito por las clases hijas, es un **método abstracto**. Se debe tener en cuenta que los métodos abstractos solo pueden ser definidos en clases abstractas.

De manera similar a las clases abstractas, para mostrar que un método es abstracto en el diagrama de clases, se escribe la palabra *abstract* antes del nombre del método, entre los símbolos << y >>. También el nombre del método va en cursiva.

Con todos estos elementos, el diagrama de clases para las cuentas bancarias queda:



### 19.4. Implementación en Java

En Java se utiliza la palabra reservada *abstract* en las clases y métodos abstractos. Además, los métodos abstractos solo tienen el encabezado, el cual termina en punto y coma. Estos métodos no tienen cuerpo.

Por ejemplo, el código de la clase Cuenta es:

```

/**
 * Una cuenta bancaria, es decir, un registro en
 * una entidad bancaria que tiene un saldo (cantidad de dinero)
 * que se puede modificar mediante retiros o consignaciones.
 * @author Sandra V. Hurtado
 * @version 4.5
 */
public abstract class Cuenta
{
    private String numero;
    protected double saldo;

    /**
     * Constructor de objetos Cuenta, con saldo inicial cero.
     * @param numero el número de la cuenta
     */
    public Cuenta(String numero)
    {
        this.numero = numero;
        this.saldo = 0;
    }

    public double getSaldo()
    {
        return this.saldo;
    }

    public String getNumero()
    {
        return this.numero;
    }

    /**
     * Consigna o adiciona una cantidad de dinero en la cuenta,
     * lo cual incrementa el saldo.
     * @param cantidad la cantidad de dinero que se desea
     *                consignar, en pesos
     */
    public void consignar(double cantidad)
    {
        saldo = saldo + cantidad;
    }

    /**
     * Retira o saca una cantidad de dinero de la cuenta
     * @param cantidad la cantidad que se desea retirar, en pesos.
     * @return si se pudo hacer el retiro o no
     */
    public abstract boolean retirar(double cantidad);
}

```

Clase abstracta

método abstracto

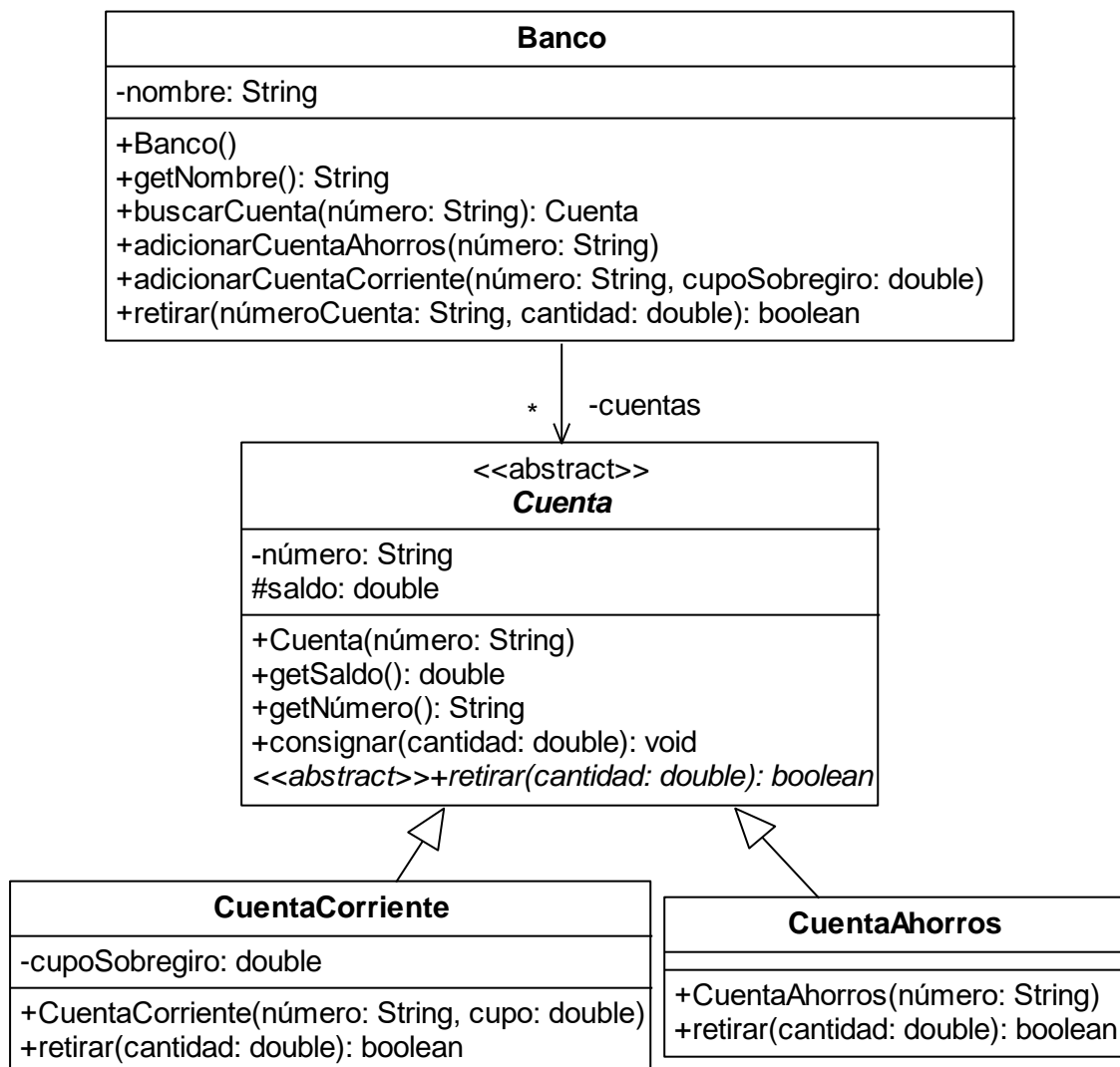
Las clases hijas están obligadas a sobrescribir los métodos abstractos que heredan. En caso de que no tengan el método, sale un error de compilación.

### Ejercicio 19-1

Escriba el código Java para las clases CuentaCorriente y CuentaAhorros.

Las clases abstractas permiten que las clases hijas realicen de forma diferente las operaciones, pero sin que esto afecte el esquema general de trabajo con ellas. Por ejemplo, el banco MuchoDinero no necesita tener diferentes colecciones para las cuentas corrientes y las cuentas de ahorro, sino que con solo una colección de cuentas puede manejarlas todas. El único momento donde se deben diferenciar las cuentas es cuando se crean, por ese motivo se tendrá un método para adicionar cuentas corrientes y otro para cuentas de ahorro, pero los demás métodos manejan las cuentas de manera uniforme.

El diagrama de clases queda:



El código de la clase Banco es:

```

import java.util.ArrayList;

/**
 * Entidad bancaria que maneja varias cuentas con dinero
 * @author Sandra V. Hurtado
 * @version 2.5
 */
public class Banco
{
    private String nombre;
    private ArrayList<Cuenta> cuentas;

    public Banco ()
    {
        this.nombre = "MuchoDinero";
        this.cuentas = new ArrayList<>();
    }

    public String getNombre()
    {
        return nombre;
    }

    /**
     * Buscar una cuenta en la lista, a partir del número.
     * @param numero el número que identifica la cuenta que se buscará
     * @return el objeto Cuenta que corresponde al número dado,
     *         o null si no se encuentra.
     */
    public Cuenta buscarCuenta(String numero)
    {
        for (Cuenta cuentaComparar : cuentas)
        {
            if (cuentaComparar.getNumero().equals(numero))
            {
                return cuentaComparar;
            }
        }
        return null;
    }

    /**
     * Se crea una nueva cuenta de ahorros y se adiciona a la lista.
     * @param numero el número de la nueva cuenta - debe ser único
     * @return indicación de si se pudo crear y adicionar la cuenta,
     *         y false en caso contrario
     */
    public boolean adicionarCuentaAhorros(String numero)
    {

```

Un solo *ArrayList*

//continúa

```

        Cuenta cuentaExistente = buscarCuenta(numero);
        if (cuentaExistente == null)
        {
            // Se crea la cuenta de ahorros
            Cuenta nuevaCuenta = new CuentaAhorros(numero);
            return cuentas.add(nuevaCuenta);
        }
        return false;
    }

    /**
     * Se crea una nueva cuenta corriente y se adiciona a la lista.
     * @param numero el número de la nueva cuenta - debe ser único
     * @param cupoSobregiro el cupo de retiro más allá del saldo
     * @return true si se pudo crear y adicionar la cuenta,
     *         y false en caso contrario
     */
    public boolean adicionarCuentaCorriente(String numero,
        double cupoSobregiro)
    {
        Cuenta cuentaExistente = buscarCuenta(numero);
        if (cuentaExistente == null)
        {
            // Se crea la cuenta corriente
            Cuenta nuevaCuenta =
                new CuentaCorriente(numero, cupoSobregiro);
            return cuentas.add(nuevaCuenta);
        }
        return false;
    }

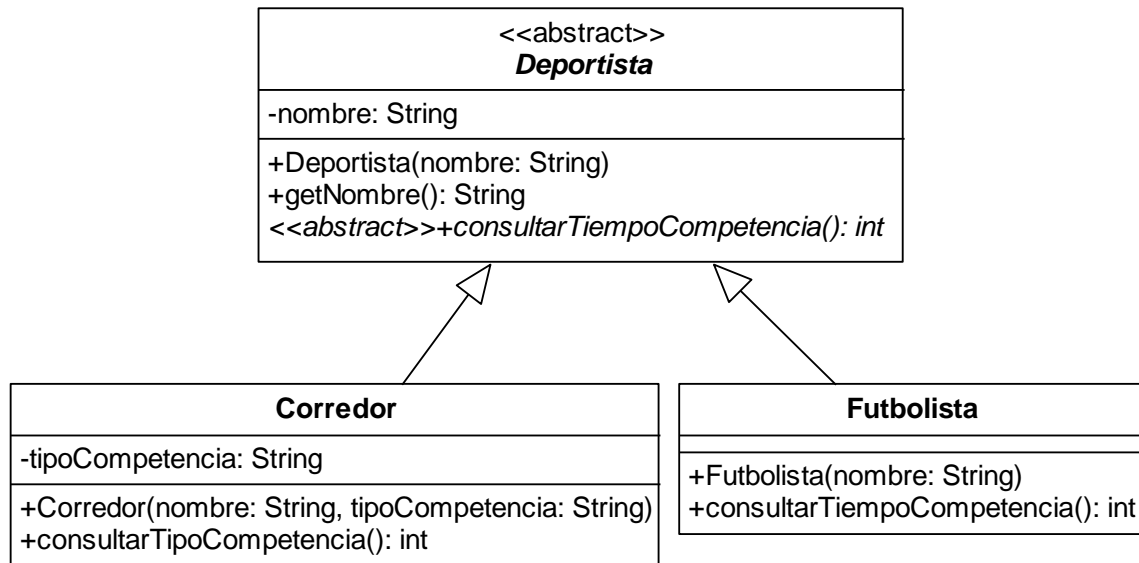
    /**
     * Busca una cuenta en el banco y retira dinero de ella
     * @param numeroCuenta el número que identifica la cuenta
     * @param cantidad cantidad, en pesos, que desea retirar
     * @return true si pudo retirar, y false si no pudo
     *         retirar o si no encontró la cuenta.
     */
    public boolean retirar(String numeroCuenta, double cantidad)
    {
        Cuenta cuenta = buscarCuenta(numeroCuenta);
        if (cuenta != null)
        {
            return cuenta.retirar(cantidad);
        }
        return false;
    }
} // fin clase Banco

```

Se llama al método "retirar", sin importar si es cuenta corriente o de ahorros

### Ejercicio 19-2

Escriba el código Java correspondiente al siguiente diagrama de clases, considerando que los futbolistas juegan en promedio 90 minutos, mientras que los corredores de maratón corren en promedio 150 minutos, y los de carreras cortas 10 minutos.



## 19.5. Métodos y clases final

Las clases abstractas son usadas para comenzar una jerarquía de clases, es decir, se espera que tengan clases hijas. Lo opuesto de esto es definir una clase que NO pueda tener clases hijas. Sin embargo, es muy raramente usado, porque limita la flexibilidad que deben tener los programas orientados a objetos.

En Java, para indicar que una clase no puede tener hijas, se usa la palabra reservada *final*. Por ejemplo:

```
/**
 * Herramienta usada para cortar
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public final class Tijera
{
    /**
     * Cortar o sacar trozos de un papel con la tijera
     */
    public void cortarPapel()
    {
        System.out.println("Cortando un papel");
    }
}
```

No se pueden tener  
clases hijas de Tijera



También es posible tener métodos que NO pueden ser sobrescritos, lo cual se define en Java con la misma palabra reservada *final*. Por ejemplo:

```
/**
 * Método que muestra un mensaje por consola
 * @param mensaje la cadena de texto que se mostrará
 */
public final void mostrarMensaje(String mensaje)
{
    System.out.println(mensaje);
}
```

No se puede  
sobrescribir este método

## 19.6. Ejercicio resuelto

La abuela Pata ha encontrado que el precio de sus frascos de dulce es algo diferente para los frascos de mermelada y para los frascos de dulces tradicionales. Para las mermeladas el precio es de \$10 por gramo, para todos los sabores, pero si tiene más de 80 gramos de azúcar, entonces tiene un incremento del 10 %. Para los dulces tradicionales el precio es de \$15 por gramo, si el frasco es de menos de 300 gramos, y de \$12 por gramo, si el frasco es de 300 gramos o más. Además, si el dulce tradicional es de brevas tiene un incremento del 5 %.

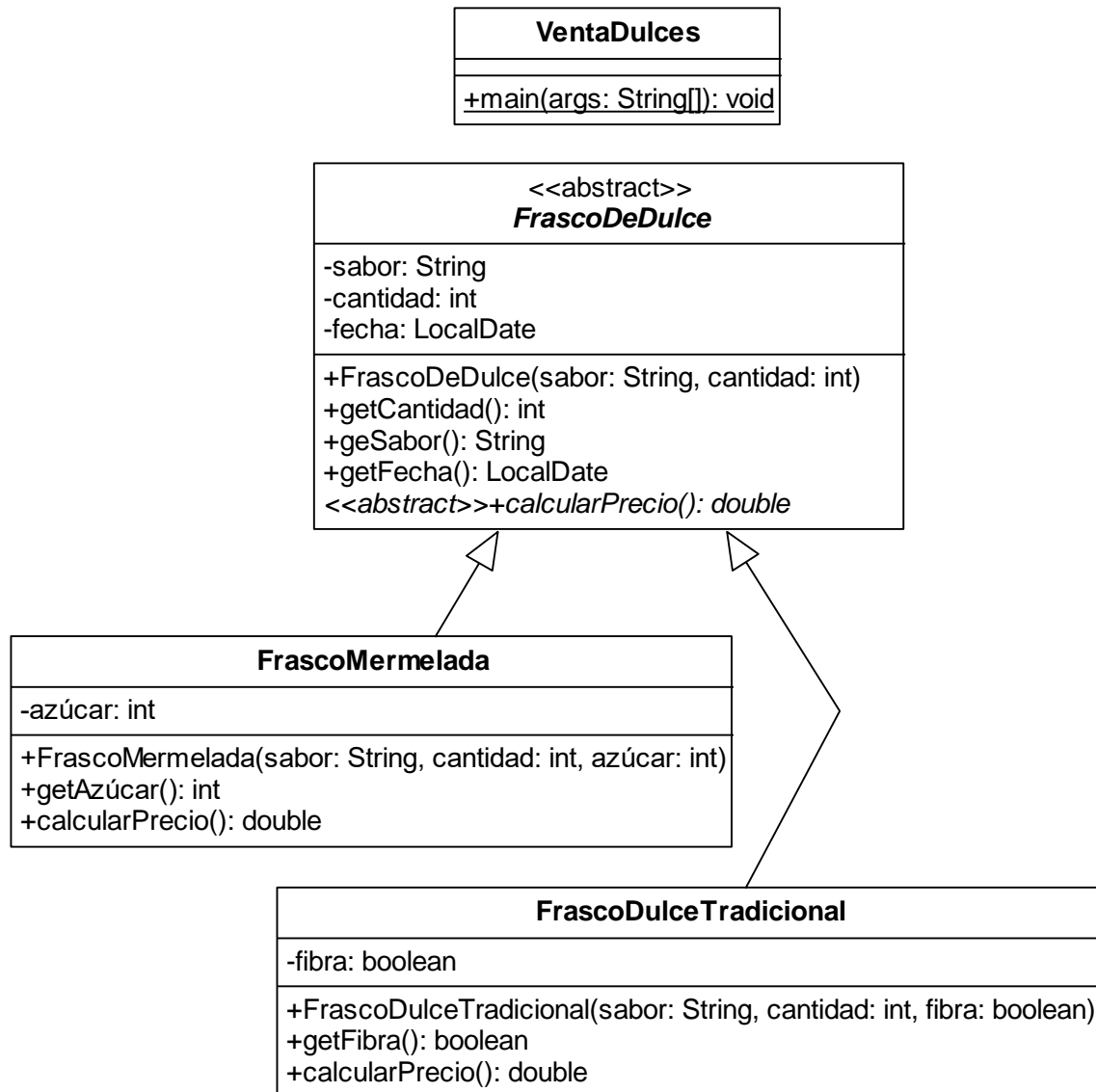
Cuando se crea el frasco de dulce se debe definir inmediatamente su sabor y la cantidad (máximo 500 gramos), y se toma la fecha del sistema como la fecha de envasado.

Ahora que ya tiene bien diferenciados sus dulces, la abuela desea crear frascos de mermelada o frascos de dulces tradicionales, no simplemente dulces.

Para lograr lo que desea la abuelita, se definirá la clase *FrascoDeDulce* como una clase abstracta, pues ya no se crearán objetos de esta clase sino de las clases hijas. Además, se tendrá el método “*calcularPrecio*”, que también será abstracto, para que cada una de sus hijas pueda hacer el cálculo diferente.

También se tendrá una clase para hacer una prueba, donde se crearán un frasco de mermelada y un frasco de dulce tradicional, y se mostrará el precio de cada uno, para que la abuela Pata confirme que el programa funciona apropiadamente.

Se presentan a continuación el diagrama de clases y el código Java correspondiente.



```
import java.time.LocalDate;
```

```
/**
 * Un frasco con delicioso dulce, elaborado por la abuela pata.
 * Por ejemplo, un frasco de dulce de guayaba.
 * @author Sandra V. Hurtado
 * @version 4.5
 */
```

```
public abstract class FrascoDeDulce
{
```

//continúa

```

private String sabor;
private int cantidad;
private LocalDate fecha;

/**
 * Constructor de nuevos objetos FrascoDeDulce,
 * tomando como fecha de envasado la fecha del sistema
 * @param sabor el sabor del dulce, por ejemplo, "brevas"
 * @param cantidad la cantidad, en gramos, para envasar
 *                (debe ser máximo 500 gramos)
 */
public FrascoDeDulce(String sabor, int cantidad)
{
    this.sabor = sabor;
    this.cantidad = (cantidad > 500)?500:cantidad;
    this.fecha = LocalDate.now();
}

public int getCantidad()
{
    return this.cantidad;
}

public String getSabor()
{
    return sabor;
}

public LocalDate getFecha()
{
    return fecha;
}

/**
 * Calcula el precio del frasco de dulce,
 * dependiendo de la cantidad y el sabor.
 * @return el precio, en pesos, del frasco.
 */
public abstract double calcularPrecio();

```

método abstracto

```

} // fin clase FrascoDeDulce

```

```

/**
 * Un frasco de mermelada, que es uno de los tipos de dulces
 * que elabora la abuela Pata.
 * @author Sandra V. Hurtado
 * @version 1.5
 */
public class FrascoMermelada extends FrascoDeDulce
{
    private int azucar;

    /**
     * Constructor de un frasco de mermelada
     * @param sabor    el sabor de la mermelada, por ejemplo: piña
     * @param cantidad la cantidad, en gramos, para envasar
     * @param azucar   la cantidad (en gramos) de azúcar
     */
    public FrascoMermelada(String sabor, int cantidad, int azucar)
    {
        super(sabor,cantidad);
        this.azucar = azucar;
    }

    public int getAzucar()
    {
        return this.azucar;
    }

    /**
     * Calcula el precio del frasco de dulce,
     * dependiendo de la cantidad y el azúcar que tiene.
     * @return el precio, en pesos, del frasco.
     */
    @Override
    public double calcularPrecio()
    {
        double precio = 10 * this.getCantidad();
        if (this.azucar > 80) {
            precio += precio*0.1;
        }
        return precio;
    }
}

```

Aquí se define el método

```

/**
 * Un frasco de dulce tradicional, que es la especialidad
 * de dulces elaborados por la abuela Pata.
 * @author Sandra V. Hurtado
 * @version 1.5
 */
public class FrascoDulceTradicional extends FrascoDeDulce
{
    private boolean fibra;

    /**
     * Constructor de un frasco de dulce tradicional.
     * @param sabor el sabor del dulce, por ejemplo: brevas.
     * @param cantidad la cantidad, en gramos, para envasar
     * @param fibra indicación de si tiene o no fibra natural
     */
    public FrascoDulceTradicional(String sabor,
                                   int cantidad, boolean fibra)
    {
        super(sabor,cantidad);
        this.fibra = fibra;
    }

    public boolean getFibra()
    {
        return fibra;
    }

    /**
     * Calcula el precio del frasco de dulce,
     * dependiendo de la cantidad y el sabor.
     * @return el precio, en pesos, del frasco.
     */
    @Override
    public double calcularPrecio()
    {
        int cantidad = this.getCantidad();
        String sabor = this.getSabor();
        double precio = 0;
        if (cantidad < 300)
        {
            precio = 15 * cantidad;
        }
        else
        {
            precio = 12 * cantidad;
        }
        if (sabor.equalsIgnoreCase("brevas"))
        {
            precio += precio*0.05;
        }
        return precio;
    }
}

```

Aquí se define el método

```

import javax.swing.JOptionPane;

/**
 * Se crean un frasco de dulce de mermelada
 * y un frasco de dulce tradicional,
 * y se muestran los precios de cada uno
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class VentaDulces
{
    public static void main(String[] args)
    {
        FrascoDeDulce frascoDulce;
        double precioFrasco = 0;


        // Se crea un frasco de mermelada
        String sabor =
            JOptionPane.showInputDialog("Sabor mermelada:");
        String cantidadCadena =
            JOptionPane.showInputDialog("Cantidad(gr.):");
        int cantidad = Integer.parseInt(cantidadCadena);
        String azucarCadena =
            JOptionPane.showInputDialog("Azúcar(gr.) mermelada:");
        int azucar = Integer.parseInt(azucarCadena);
        frascoDulce = new FrascoMermelada(sabor,cantidad,azucar);
        precioFrasco = frascoDulce.calcularPrecio();
        JOptionPane.showMessageDialog(null, "Precio: $" + precioFrasco);

        // Se crea un frasco de dulce tradicional
        sabor =
            JOptionPane.showInputDialog("Sabor dulce tradicional:");
        cantidadCadena =
            JOptionPane.showInputDialog("Cantidad(gr.):");
        cantidad = Integer.parseInt(cantidadCadena);
        String fibraCadena =
            JOptionPane.showInputDialog("¿Tiene fibra (S/N)?");
        boolean fibra = (fibraCadena.charAt(0) == 'S')?true:false;
        frascoDulce = new FrascoDulceTradicional(sabor,cantidad,fibra);
        precioFrasco = frascoDulce.calcularPrecio();
        JOptionPane.showMessageDialog(null, "Precio: $" + precioFrasco);
    }
}

```

Un ejemplo de ejecución de este programa, con una mermelada de fresa de 200 gramos y 50 gramos de azúcar, y un dulce tradicional de brevas de 150 gramos y con fibra:


Entrada

 Sabor mermelada:

fresa

Aceptar Cancelar


Entrada

 Cantidad(gr.):

200

Aceptar Cancelar


Entrada

 Azúcar(gr.) mermelada:

50


Aceptar Cancelar

Mensaje

 Precio: \$2000.0

Aceptar


Entrada

 Sabor dulce tradicional:

brevas

Aceptar Cancelar

Entrada

 Cantidad(gr.):

150

Aceptar Cancelar

Entrada

¿Tiene fibra (S/N)?

S

Aceptar Cancelar

Mensaje

Precio: \$2362.5

Aceptar



## 20. Interfaces

En este capítulo se presentará el concepto de interfaces en Java, que representan un nivel de abstracción incluso mayor al de las clases abstractas, y que por lo tanto permiten la construcción de programas mucho más flexibles y completos.

A finalizar este capítulo usted debe ser capaz de:

- Representar una interfaz en un diagrama de clases.
- Escribir el código Java correspondiente a una interfaz y una clase que la implementa.

### 20.1. Un nuevo reto

La abuela Pata sigue creciendo su negocio, pues le ha ido muy bien con las ventas. Ahora desea complementar las ventas de dulces con venta de bordados, pues es otra de las aficiones que tiene y piensa que puede diversificar su mercado.

Aunque los bordados también tienen una forma de calcular su precio, en lo cual se parecen en los dulces, en todo lo demás son diferentes, pues no tienen cantidad, sabor o fecha de elaboración.

Es claro que debemos crear una nueva clase para los bordados, pero no puede heredar de Dulce porque son productos diferentes. Entonces, ¿cómo manejar las ventas, considerando que tanto los dulces como los bordados tienen la forma de calcular su precio? Es aquí donde nos ayuda una interfaz.

### 20.2. Definición

Una interfaz en Java **define un comportamiento**, es decir, un conjunto de métodos o servicios que posteriormente tendrá alguna clase. A diferencia de una clase abstracta, una interfaz no tiene una estructura, es decir, no tiene atributos ni métodos que puedan ser heredados.

La sintaxis para definir una interfaz en Java es:

```
/**
 * Comentario que describe la interfaz
 */
public interface NombreInterfaz
{
    // Métodos abstractos
}
```

Los nombres de las interfaces deben comenzar con mayúscula, como las clases. A veces los nombres de las interfaces incluyen la terminación “able” o “ible” (aunque no es obligatorio), como “Adivinable”, “Comparable”, “Traducible”, etc., para indicar que es un comportamiento.

En una interfaz no pueden definirse variables de instancia, pero sí pueden definirse constantes.

Es importante tener en cuenta que todos los métodos definidos en una interfaz son **públicos**, incluso si no se coloca la palabra *public*. Por lo tanto, no pueden tenerse niveles *protected* ni *private*.

También es importante considerar que los métodos definidos en una interfaz **son abstractos**<sup>4</sup>, así no se indique con la palabra *abstract*.

Por ejemplo:

```
/**
 * Comportamiento de elementos que pueden ser alquilados, es decir,
 * que pueden darse a otro para su uso por un tiempo a cambio de un pago.
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public interface Alquilable
{
    /**
     * Método que permite conocer el valor mensual del alquiler
     * @return el valor, en pesos, del alquiler del objeto por un mes
     */
    public abstract double valorAlquilerMensual();

    /**
     * Método que permite alquilar el objeto por un tiempo dado
     * @param meses la cantidad de meses que se alquilará el objeto
     * @return si se pudo alquilar o no,
     *         pues no se puede alquilar dos veces en el mismo tiempo
     */
    boolean alquilar(int meses);
}
```

Este método es **público y abstracto**, así no se indique explícitamente.

En el ejemplo anterior se está creando una interfaz llamada *Alquilable*, con dos métodos, que deben estar **-obligatoriamente-** en las clases que implementen esta interfaz. Estos métodos son: “*valorAlquilerMensual*” y “*alquilar*”, cada uno con el tipo de retorno y los parámetros indicados.

## 20.3. Implementación de una interfaz

Como una interfaz solo define un comportamiento, para ser utilizadas es necesario que alguna clase **implemente** ese comportamiento, es decir, que tenga los métodos definidos en la interfaz, pero con cuerpo.

---

<sup>4</sup> Existe una excepción a esta regla, pues una interfaz puede tener métodos por defecto y métodos estáticos; pero es algo que va más allá del alcance de este libro.

En Java, para decir que una clase implementa una interfaz se utiliza la palabra reservada *implements*. Una clase puede implementar varias interfaces, y esto se indica colocando los nombres de las interfaces separados por comas.

La sintaxis es:

```
class NombreClase extends ClasePadre implements Interfaz1, Interfaz2
{
    // código clase
}
```

Las clases en Java sólo pueden heredar de una clase, pero pueden implementar varias interfaces, por lo que se dice que las interfaces son una forma de herencia múltiple.

Cuando una clase implementa una interfaz está indicando que se compromete a tener el comportamiento definido en ella. Como todos los métodos de la interfaz son abstractos la clase está obligada a definirlos.

A continuación, se presenta un ejemplo de un apartamento y de una lavadora que pueden ser alquilados. Por simplicidad no se tendrá en cuenta algunas validaciones con respecto al tiempo de alquiler.

```
/**
 * Un lugar para habitar, por lo general como parte independiente
 * de un edificio o casa. Puede ser alquilado.
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Apartamento implements Alquilable
{
    private String direccion;
    private double area;
    private char estado;

    /**
     * Constructor de un objeto apartamento,
     * que queda en estado disponible ('d')
     * @param direccion la ubicación del apartamento.
     * @param area el área, en metros cuadrados, del apartamento.
     * Permite definir el valor del alquiler.
     */
    public Apartamento(String direccion, double area)
    {
        this.direccion = direccion;
        this.area = area;
        this.estado = 'd';
    }
}
```

*//continúa*

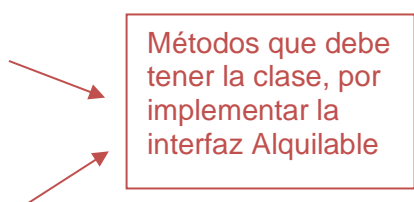
```

public double getArea()
{
    return area;
}

/**
 * Calcula el valor del alquiler del apartamento por su área.
 * Se cobran $10.000 por cada metro cuadrado (mensualmente).
 * @return el valor del alquiler mensual del apartamento, en pesos.
 */
@Override
public double valorAlquilerMensual()
{
    final double VALOR_M2 = 10000;
    return (area * VALOR_M2);
}

/**
 * Para alquilar un apartamento:
 * Si está disponible (estado 'd') cambia su estado a
 * alquilado ('a'). Si no está disponible entonces
 * no se puede alquilar (no cambia el estado y retorna false).
 * @return un valor true o false indicando si se pudo alquilar o no.
 */
@Override
public boolean alquilar(int meses)
{
    if (this.estado == 'd')
    {
        this.estado = 'a';
        return true;
    }
    return false;
}
} // fin clase Apartamento

```



Métodos que debe tener la clase, por implementar la interfaz Alquilerable

Clase Lavadora, que también implementa la interfaz Alquilerable:

```

/**
 * Electrodoméstico usado para lavar la ropa.
 * Se puede alquilar para su uso por otras personas.
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Lavadora implements Alquilerable
{
    private int capacidad;
    private String marca;

```

//continúa

```

/**
 * Constructor de objetos lavadoras.
 * @param capacidad la capacidad, en litros, de ropa para lavar
 * @param marca la marca o fabricante de la lavadora
 */
public Lavadora(int capacidad, String marca)
{
    this.capacidad = capacidad;
    this.marca = marca;
}

public int getCapacidad()
{
    return this.capacidad;
}

/**
 * Calcula el valor del alquiler de la lavadora.
 * Se cobran un valor fijo por mes.
 * @return el valor, en pesos, del alquiler mensual de la lavadora.
 */
@Override
public double valorAlquilerMensual()
{
    return 60000;
}

/**
 * Se define que siempre se puede alquilar la lavadora.
 * @return true, indicando que se puede alquilar.
 */
@Override
public boolean alquilar(int meses)
{
    return true;
}
} // fin clase Lavadora

```

En el ejemplo anterior puede observarse que ambas clases (Apartamento y Lavadora) implementan los métodos “valorAlquilerMensual” y “alquilar”, que están incluidos en la interfaz Alquilerable, aunque cada clase tiene un código diferente para estos métodos.

Es decir, que la interfaz define el comportamiento o servicio, pero cada clase decide cómo lo hace, es decir, puede tener formas diferentes de implementar o llevar a cabo ese servicio.

### **Ejercicio 20-1**

Identifique el error que se presenta a continuación y explique por qué se presenta.

```

/**
 * Elementos con medidas, a los que se les puede pedir su área.
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public interface Medible
{
    public double area();
}

```

```

/**
 * Cuarto o habitación en una casa, con medidas
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Habitacion implements Medible
{
    private int largo;
    private int ancho;

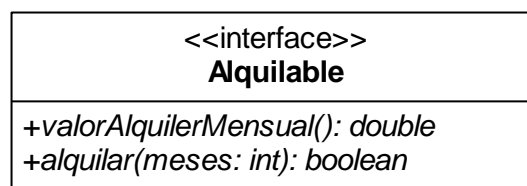
    /**
     * Obtiene el área de la habitación
     * @return el área en metros cuadrados
     */
    @Override
    public int area()
    {
        return (largo * ancho);
    }
}

```

## 20.4. Representación

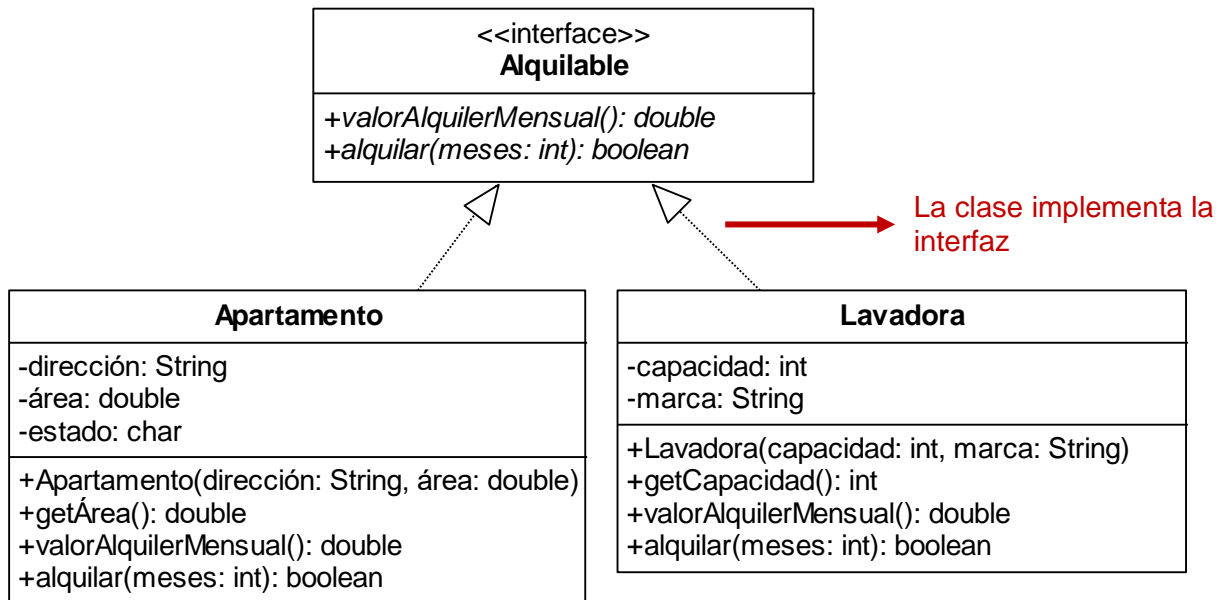
Una interfaz se representa gráficamente como una clase, pero tiene la palabra “*interface*”, encerrada entre los símbolos: << y >> antes del nombre de la interfaz.

Por ejemplo, la interfaz Alquilable:



Para indicar que una clase implementa una interfaz se representa de manera similar a la herencia, pero con la línea punteada o discontinua.

Por ejemplo, el diagrama correspondiente a las clases Apartamento y Lavadora, que implementan la interfaz Alquilable, es:



## 20.5. Uso de las interfaces

Las interfaces pueden ser utilizadas en un programa de la misma forma en que se utilizan las clases abstractas, es decir, aunque no se pueden tener instancias de ellas, sí se pueden tener variables. Estas variables harán referencia a objetos de clases que implementan la interfaz. Ejemplo:

```

/**
 * Clase donde se crean objetos "alquilables",
 * y se muestra su valor de alquiler mensual
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class ConsultaAlquileres
{
    public static void main(String[] args)
    {
        Alquilable alquilable1 = new Apartamento("Cra 1 #3-2",50);
        Alquilable alquilable2 = new Lavadora(20,"LavaMas");

        System.out.println("Valor alquiler apartamento: $" +
            alquilable1.valorAlquilerMensual());
        System.out.println("Valor alquiler lavadora: $" +
            alquilable2.valorAlquilerMensual());
    }
}

```

*//continúa*

```

        // Esto sería un error:
        // System.out.println(alquilable2.getCapacidad());
        // Con esa variable solo pueden usarse
        // los métodos que están en la interfaz Alquilable
    }
}

```

La salida de este programa es:

```

Valor alquiler apartamento: $500000.0
Valor alquiler lavadora: $60000.0

```

Al igual que con las clases abstractas, se puede saber si un objeto implementa o no una interfaz, utilizando el operador *instanceof*:

```

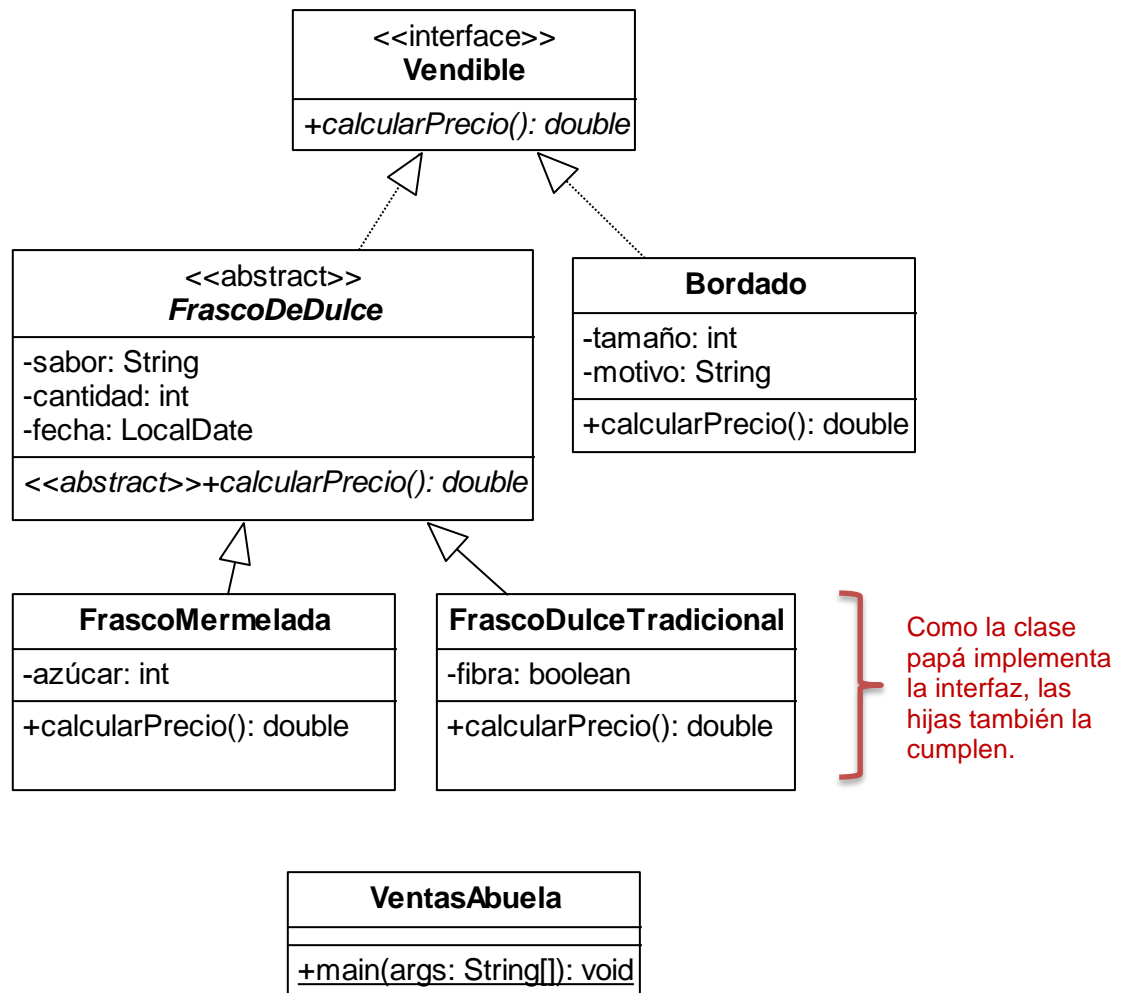
if (electrodomestico instanceof Alquilable)
{
    //código
}

```

En el caso de la abuela Pata, ahora puede tener una interfaz para todo lo que ella pueda vender, que debe tener alguna forma de calcular el precio. Esto incluye los dulces y también los bordados, e incluso cualquier otro producto que desee vender más adelante. La interfaz, como es para todo lo que la abuela desee vender, se llamará “Vendible”.

A continuación, se muestra el diagrama de clases propuesto, donde se omiten los constructores y los métodos “get” para hacer énfasis en los métodos propios de la interfaz:





Como ejemplo, se muestra el código de la clase `VentasAbuela`, donde se crean dos objetos vendibles: un dulce y un bordado, y luego se muestran los precios de cada uno y el precio total (la suma de los dos). Puede observarse como ambos objetos se guardan en una lista de objetos `Vendibles`, simplificando así el programa y permitiendo que luego puedan crearse y venderse nuevos objetos sin hacer muchos cambios.

```

import java.util.ArrayList;
import javax.swing.JOptionPane;

/**
 * Se crean un frasco de dulce y un bordado,
 * y se muestran los precios de cada uno.
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class VentasAbuela
{
    public static void main(String[] args)
    {

```

*//continúa*

```

// Se crea una lista de objetos vendibles
ArrayList<Vendible> vendibles = new ArrayList<>();

// Se crea un frasco de mermelada y se adiciona a la lista
String sabor =
    JOptionPane.showInputDialog("Sabor mermelada:");
String cantidadCadena =
    JOptionPane.showInputDialog("Cantidad(gr.):");
int cantidad = Integer.parseInt(cantidadCadena);
String azucarCadena =
    JOptionPane.showInputDialog("Azúcar(gr.):");
int azucar = Integer.parseInt(azucarCadena);
Vendible dulce = new FrascoMermelada(sabor,cantidad,azucar);
vendibles.add(dulce);

// Se crea un bordado y se adiciona a la lista
String tamanoCadena =
    JOptionPane.showInputDialog("Tamaño bordado:");
int tamano = Integer.parseInt(tamanoCadena);
String motivo =
    JOptionPane.showInputDialog("Motivo:");
Vendible bordado = new Bordado(tamano, motivo);
vendibles.add(bordado);

// Se muestran los precios de cada objeto y el total
double precioTotal = 0;
for (Vendible vendible : vendibles)
{
    double precio = vendible.calcularPrecio();
    precioTotal+=precio;
    JOptionPane.showMessageDialog(null, "Precio: $" + precio);
}
JOptionPane.showMessageDialog(null,
    "Precio Total: $" + precioTotal);
}
} // fin clase VentasAbuela

```

### **Ejercicio 20-2**

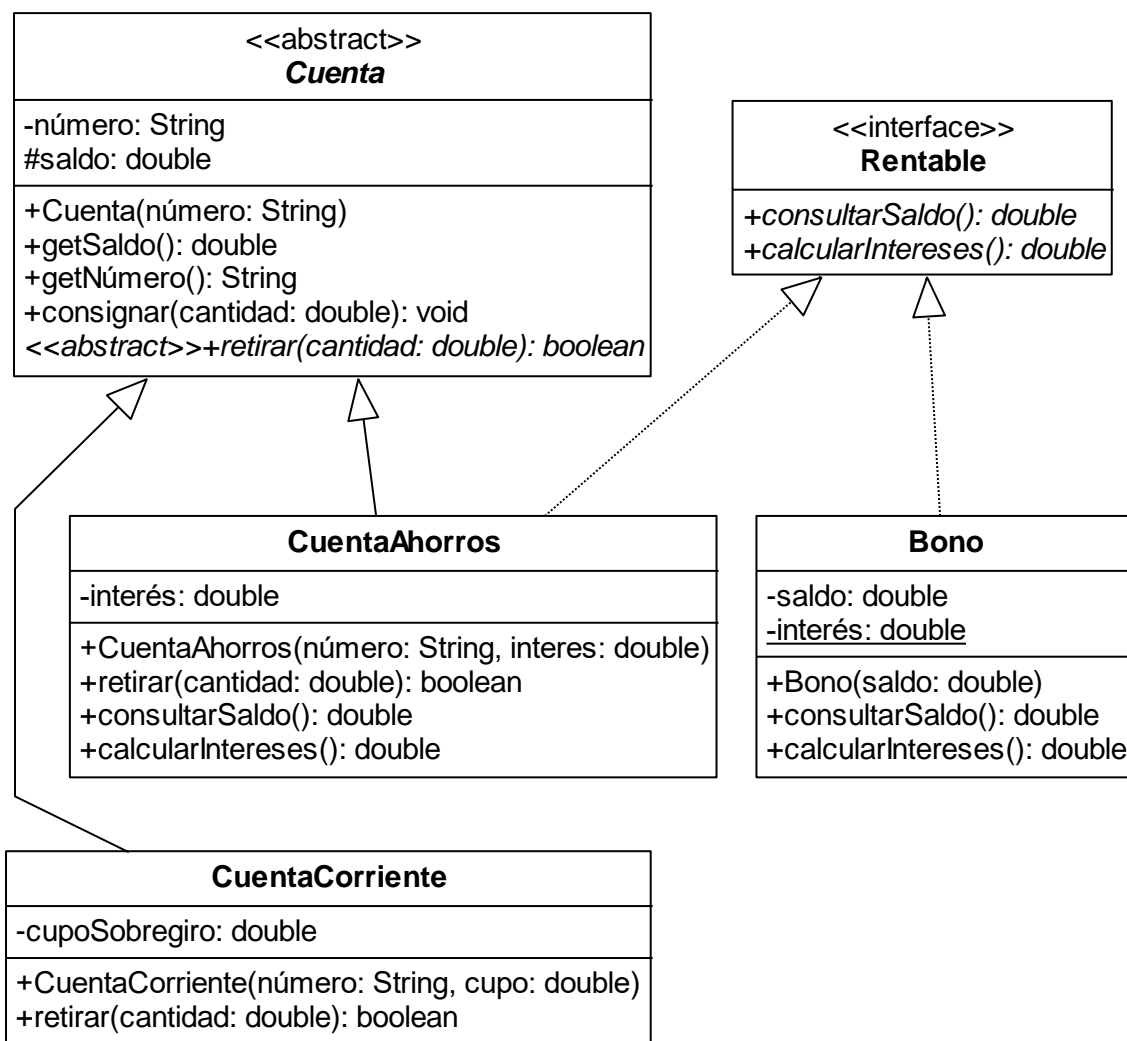
Dada la siguiente descripción de un problema elabore el diagrama de clases correspondiente (incluyendo interfaces si es necesario):

Se necesita tener información de los diferentes productos que se venden en una tienda. Esta tienda en particular vende: Comida (de la cual se desea saber las calorías que tiene), Juguetes (de los cuales se desea saber la edad mínima requerida) y Libros (de los cuales se desea conocer su autor). A los juguetes y los libros se les aplica un impuesto (IVA), pero no a la comida. Hay otros elementos a los cuales se les aplica el IVA, incluyendo algunos servicios, de manera que el concepto de impuesto está separado del concepto "Producto". Se sabe que el IVA, para todos los elementos a los cuales se aplica, es del 16 %, y que cualquier elemento al que se le pueda aplicar este impuesto debe tener un método "calcularValorIva", para poder saber el valor del IVA que le corresponde.

## 20.6. Ejercicio resuelto

En el banco MuchoDinero, además de ofrecer cuentas corrientes y cuentas de ahorro, ahora tendrán también bonos. Los bonos son un servicio de ahorro a largo plazo, donde las personas consignan un dinero para luego recibirlo con algunos intereses, pero después de dejarlo un tiempo en el banco. El interés que se ha determinado para los bonos es del 10 % anual. A diferencia de las cuentas, en los bonos no se puede retirar ni consignar dinero, pero sí se puede consultar el saldo. Además, para hacer más atractivos sus productos, el banco también ofrecerá diferentes intereses en las cuentas de ahorro. Por lo tanto, tanto las cuentas de ahorro como los bonos se consideran productos Rentables, mientras que las cuentas corrientes no. A los productos rentables, además de consultar el saldo básico, también se les podrá consultar los intereses que se pueden obtener con ese saldo.

El diagrama de clases para representar esto es:



Seguidamente se presenta el código de la interfaz **Rentable**, y de las clases **CuentaAhorros** y **Bono** (el código de las otras dos clases no se presenta, por no tener una relación directa con la interfaz).

```

/**
 * Comportamiento de los productos bancarios
 * que ofrecen intereses (rentabilidad).
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public interface Rentable
{
    /**
     * Permite consultar el saldo, es decir, la cantidad
     * de dinero (sin intereses) que tiene el producto.
     * @return el saldo actual, en pesos.
     */
    public double consultarSaldo();

    /**
     * Permite obtener el valor de los intereses,
     * considerando el saldo actual del producto.
     * @return el valor de los intereses, en pesos.
     */
    public double calcularIntereses();
}

```

```

/**
 * Un producto bancario para guardar dinero por un tiempo,
 * y que genera algunos intereses al final.
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Bono implements Rentable
{
    private double saldo;
    private static double interes = 0.1;

    /**
     * Construye un objeto Bono, con un saldo base
     * @param saldo la cantidad de dinero del bono, en pesos
     */
    public Bono(double saldo)
    {
        this.saldo = saldo;
    }
}

```

*//continúa*

```

/**
 * Permite consultar el saldo, es decir, la cantidad
 * de dinero (sin intereses) que tiene el bono.
 * @return el saldo del bono, en pesos.
 */
@Override
public double consultarSaldo()
{
    return saldo;
}

/**
 * Permite obtener el valor de los intereses,
 * considerando el saldo actual del producto.
 * ESTE ES UN VALOR SIMULADO (solo para el ejemplo),
 * NO ES LA FORMA REAL DE CALCULAR INTERESES.
 * @return el valor de los intereses, en pesos.
 */
@Override
public double calcularIntereses()
{
    return this.saldo * interes;
}
} // fin clase Bono

```

```

/**
 * Cuenta de ahorros en una entidad financiera,
 * donde solo se puede retirar dinero hasta
 * un mínimo que debe quedar como saldo.
 * Además, tiene unos intereses que se pueden consultar.
 * @author Sandra V. Hurtado
 * @version 2.0
 */
public class CuentaAhorros extends Cuenta implements Rentable
{
    private double interes;

    /**
     * Constructor de objetos cuenta de ahorros
     * @param numero    el número de la cuenta
     * @param interes    el valor de interés anual para la cuenta
     */
    public CuentaAhorros(String numero, double interes)
    {
        super(numero);
        this.interes = interes;
    }
}

```

*//continúa*

```

/**
 * Retira o saca una cantidad de dinero de la cuenta,
 * dejando un saldo de mínimo $20.000.
 * @param cantidad la cantidad que se desea retirar, en pesos.
 * @return si se pudo hacer el retiro porque tenía dinero
 * suficiente o no (true o false)
 */
@Override
public boolean retirar(double cantidad)
{
    if ((this.getSaldo()-20000) >= cantidad)
    {
        this.saldo = this.saldo - cantidad;
        return true;
    }
    return false;
}

/**
 * Permite consultar el saldo, es decir, la cantidad
 * de dinero (sin intereses) que tiene la cuenta.
 * @return el saldo actual, en pesos, de la cuenta.
 */
public double consultarSaldo()
{
    return super.getSaldo();
}

/**
 * Permite obtener el valor de los intereses,
 * considerando el saldo actual del producto,
 * y el interés anual que tienen.
 * ESTE ES UN VALOR SIMULADO (solo para el ejemplo),
 * NO ES LA FORMA REAL DE CALCULAR INTERESES.
 * @return el valor de los intereses, en pesos.
 */
public double calcularIntereses()
{
    double intereses= (saldo * interes) / 12;
    return intereses;
}
} // fin clase CuentaAhorros

```

También se incluye el código de una clase de prueba, donde se crean y muestran dos productos rentables del banco.

```

/**
 * Programa donde se crean dos productos rentables
 * del banco y se muestran sus intereses
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class RentablesBanco
{
    public static void main(String args[])
    {
        // Se crea un bono y una cuenta de ahorros,
        // ambos con variables de tipo Rentable
        Rentable rentable1 = new Bono(500000);
        Rentable rentable2 = new CuentaAhorros("123-a",12);

        // Se muestran los intereses de cada uno
        System.out.println("Intereses bono: "+
                           rentable1.calcularIntereses());
        System.out.println("Intereses cuenta: "+
                           rentable2.calcularIntereses());
    }
}

```

## 21. Ejemplo

En este capítulo se presentará un ejemplo de un programa sencillo, que incluye varios de los conceptos vistos en el libro.

Como un elemento nuevo, en el programa usa una clase del API llamada *ImageIcon* para guardar y mostrar una imagen en una ventana de diálogo.

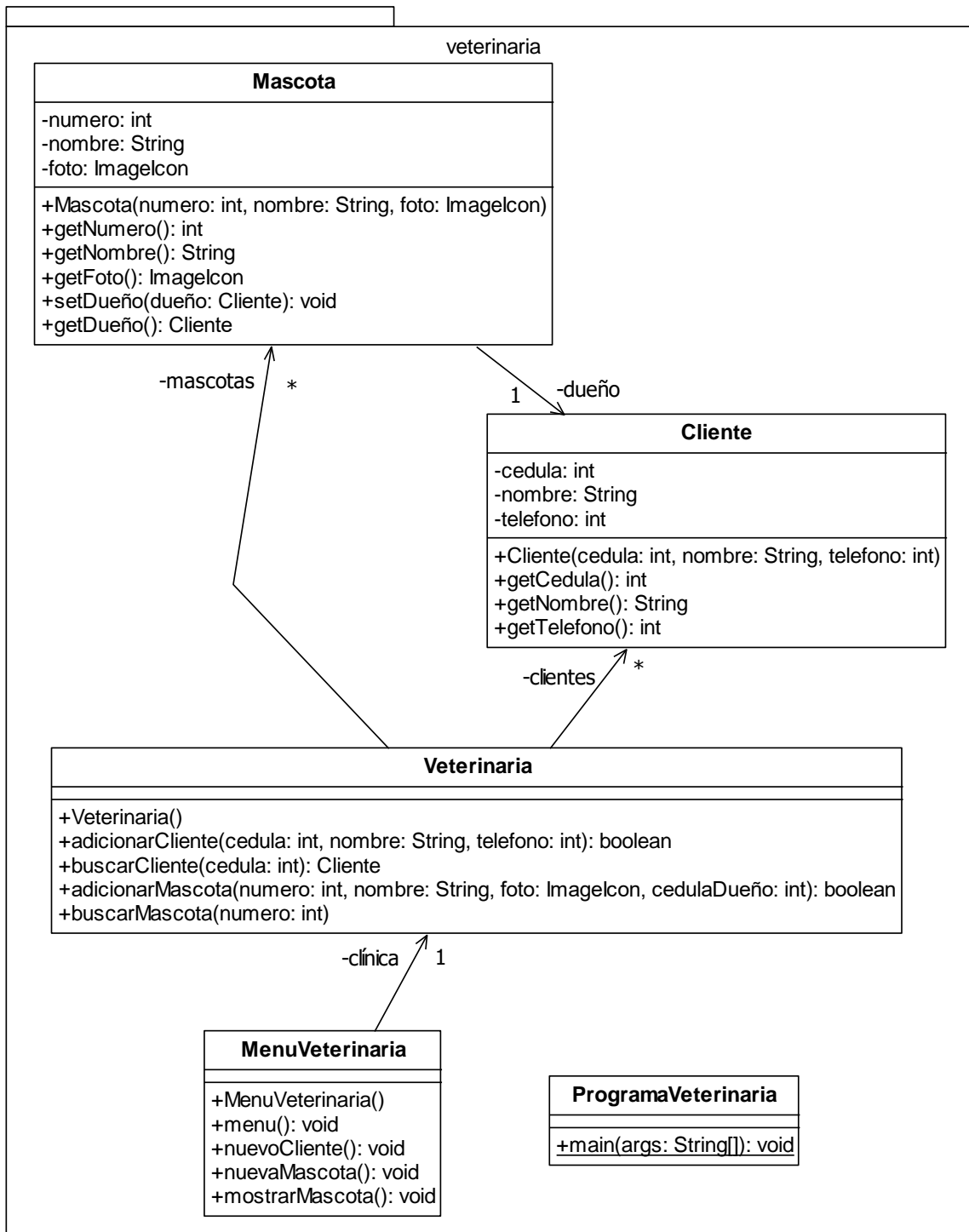
### 21.1. Enunciado

Se desea un programa para una clínica veterinaria, donde tienen información de las mascotas que atienden y de sus dueños, quienes son los clientes. El programa debe permitir:

- Registrar un nuevo cliente. Para esto se piden sus datos: cédula, nombre y teléfono. Si el cliente ya está registrado no es necesario adicionarlo de nuevo, es decir, primero se busca por la cédula para saber si es necesario registrarlo o no.
- Registrar una nueva mascota. Para esto se piden sus datos: número (para identificarla de allí en adelante) nombre, foto y la cédula del dueño. Si la cédula no corresponde a un cliente registrado no se puede registrar la mascota. Antes de registrar la mascota se busca que ya no esté otra registrada con ese número. Si ya hay una mascota con ese número entonces tampoco se puede registrar. Después de crear la mascota se debe asociar con el correspondiente dueño.
- Mostrar los datos de una mascota. Para esto se debe solicitar el número que la identifica, y después de buscarla con este número, si se encuentra se mostrará su nombre, la foto, el nombre del dueño y el teléfono de contacto.



## 21.2. Diagrama de clases



## 21.3. Código Java

Clase Cliente:

```
package veterinaria;

/**
 * Persona cliente de la veterinaria, pues es el dueño
 * de alguna mascota (o de varias)
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Cliente
{
    private int cedula;
    private String nombre;
    private int telefono;

    /**
     * Constructor de clientes, con sus datos básicos
     * @param cedula el número de cédula del cliente
     * @param nombre el nombre completo del cliente
     * @param telefono el teléfono de contacto del cliente
     */
    public Cliente(int cedula, String nombre, int telefono)
    {
        this.cedula = cedula;
        this.nombre = nombre;
        this.telefono = telefono;
    }

    public int getCedula()
    {
        return cedula;
    }

    public String getNombre()
    {
        return nombre;
    }

    public int getTelefono()
    {
        return telefono;
    }
}
```

Uso de paquetes

Atributos privados

Constructor

Métodos "get"

Clase Mascota:

```
package veterinaria;
import javax.swing.ImageIcon;
```

Clase del API de Java

```
/**
 * Mascotas que van a una clínica veterinaria
 * @author Sandra V. Hurtado
 * @version 1.0
 */
```

```
public class Mascota
{
```

```
    private int numero;
    private String nombre;
    private ImageIcon foto;
    private Cliente dueño;
```

Atributo que corresponde a una asociación

```
/**
 * Constructor de objeto Mascota, con sus datos básicos
 * @param numero un número que identifica la mascota
 * @param nombre el nombre de la mascota
 * @param foto la imagen o foto de la mascota
 */
```

```
public Mascota(int numero,String nombre,ImageIcon foto)
{
    this.numero = numero;
    this.nombre = nombre;
    this.foto = foto;
    this.dueño = null;
}
```

```
public int getNumero()
{
    return numero;
}
```

```
public String getNombre()
{
    return nombre;
}
```

```
public ImageIcon getFoto()
{
    return foto;
}
```

```
public void setDuenho(Cliente dueño)
{
    this.dueño = dueño;
}
```

Referencia *this*

```
public Cliente getDuenho()
{
    return dueño;
}
```

```
}
```

## Clase Veterinaria:

```
package veterinaria;
import java.util.ArrayList;
import javax.swing.ImageIcon;

/**
 * Clínica veterinaria con información de mascotas
 * y de sus dueños (los clientes)
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class Veterinaria
{
    private ArrayList<Cliente> clientes;
    private ArrayList<Mascota> mascotas;

    /**
     * Constructor de un objeto Veterinaria
     */
    public Veterinaria ()
    {
        this.clientes = new ArrayList<Cliente>();
        this.mascotas = new ArrayList<Mascota>();
    }

    /**
     * Permite registrar un nuevo cliente en la clínica veterinaria.
     * Antes de adicionarlo se verifica que no exista un cliente
     * ya registrado con esa cédula.
     * @param cedula el número de cédula del Cliente
     * @param nombre el nombre completo del Cliente
     * @param telefono el número de teléfono de contacto
     * @return un valor verdadero si se pudo registrar el cliente,
     * o false si no se pudo registrar
     * porque ya se tenía un cliente con esa cédula.
     */
    public boolean adicionarCliente(int cedula, String nombre,
                                     int telefono)
    {
        Cliente clienteBuscado = this.buscarCliente(cedula);
        if (clienteBuscado != null)
        {
            return false;
        }
        Cliente nuevoCliente = new Cliente(cedula, nombre, telefono);
        return clientes.add(nuevoCliente);
    }
}
```

Atributos *ArrayList*, que corresponden a asociaciones

Crear objeto con *new*

//continúa

```

/**
 * Permite buscar un cliente en la veterinaria,
 * para saber si ya está registrado
 * @param cedula el número de cédula del cliente que se buscará.
 * @return el objeto Cliente que corresponda a la cédula,
 *         o null si no se encuentra registrado.
 */
public Cliente buscarCliente(int cedula)
{
    for (Cliente cliente : clientes)
    {
        if (cliente.getCedula() == cedula)
        {
            return cliente;
        }
    }
    return null;
}

/**
 * Permite registrar una nueva mascota en la clínica veterinaria.
 * Primero se busca el dueño para saber si ya está registrado y
 * luego se verifica que no exista ya una mascota con el
 * mismo número. Si cumple las condiciones, entonces
 * se crea la mascota y se adiciona a la lista.
 * @param numero el número que identifica la mascota
 * @param nombre el nombre de la mascota
 * @param foto un objeto ImageIcon con la foto de la mascota.
 * @param cedulaDuenho la cédula del dueño
 * @return true si se pudo registrar la mascota, false si no.
 */
public boolean adicionarMascota(int numero, String nombre,
                                ImageIcon foto, int cedulaDuenho)
{
    // Busca el dueño y luego busca otra mascota con ese número
    Cliente duenho = this.buscarCliente(cedulaDuenho);
    if (duenho == null)
    {
        return false;
    }
    Mascota mascotaBuscada = this.buscarMascota(numero);
    if (mascotaBuscada != null)
    {
        return false;
    }

    // Se crea la mascota, se le asigna el dueño y se adiciona
    Mascota mascotaNueva = new Mascota(numero, nombre, foto);
    mascotaNueva.setDuenho(duenho);
    return mascotas.add(mascotaNueva);
}

```

Ciclo "foreach"

//continúa

```

/**
 * Permite buscar una mascota en la veterinaria,
 * para saber si ya está registrada
 * @param numero el número de la mascota que se buscará.
 * @return el objeto Mascota que corresponda al número,
 *         o null si no se encuentra.
 */
public Mascota buscarMascota(int numero)
{
    for (Mascota mascota : mascotas)
    {
        if (mascota.getNumero() == numero)
        {
            return mascota;
        }
    }
    return null;
}
} // fin clase Veterinaria

```

Valor *null*

Clase MenuVeterinaria:

```

package veterinaria;
import java.io.File;
import javax.swing.ImageIcon;
import javax.swing.JFileChooser;
import javax.swing.JOptionPane;

/**
 * Menú de opciones del programa de la clínica veterinaria
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class MenuVeterinaria
{
    private Veterinaria clinica;

    public MenuVeterinaria()
    {
        this.clinica = new Veterinaria();
    }
}

```

Uso de *import* para clases del API

*//continúa*

```

/**
 * Menú principal de opciones con las funciones del programa
 */
public void menu()
{
    int opcion=-1;
    do {
        String valorSeleccionado =
        JOptionPane.showInputDialog("--- CLÍNICA VETERINARIA --- \n" +
            "1.Registrar nuevo cliente    \n" +
            "2.Registrar nueva mascota    \n" +
            "3.Mostrar datos de mascota    \n" +
            "0.Terminar        \n\n" +
            "Escriba la opción seleccionada: ");

        opcion = Integer.parseInt(valorSeleccionado);
        switch (opcion)
        {
            case 1:
                nuevoCliente();
                break;

            case 2:
                nuevaMascota();
                break;

            case 3:
                mostrarMascota();
                break;

            case 0:
                break;

            default:
                JOptionPane.showMessageDialog(null,
                    "Opción no disponible");
        }
    }
    while (opcion != 0);
    JOptionPane.showMessageDialog(null," - Terminación exitosa -");
}

```

Pedir datos con *JOptionPane*

*//continúa*

```

/**
 * Pide los datos de un nuevo cliente
 * para adicionarlo a la veterinaria
 */
void nuevoCliente()
{
    String lecturaCedula = JOptionPane.showInputDialog("Cédula: ");
    int cedula = Integer.parseInt(lecturaCedula);
    String nombre = JOptionPane.showInputDialog("Nombre: ");
    String lecturaTelefono =
        JOptionPane.showInputDialog("Teléfono: ");
    int telefono = Integer.parseInt(lecturaTelefono);

    boolean pudoAdicionarCliente =
        clinica.adicionarCliente(cedula, nombre, telefono);
    if (pudoAdicionarCliente)
    {
        JOptionPane.showMessageDialog(null,
            "Cliente adicionado");
    }
    else
    {
        JOptionPane.showMessageDialog(null,
            "Ya existe un cliente con esa cédula");
    }
}

/**
 * Pide los datos de una nueva mascota
 * para adicionarla a la veterinaria
 */
void nuevaMascota()
{
    String lecturaNumero = JOptionPane.showInputDialog("Número: ");
    int numero = Integer.parseInt(lecturaNumero);
    String nombre = JOptionPane.showInputDialog("Nombre: ");

    JFileChooser selectorArchivo = new JFileChooser();
    selectorArchivo.showOpenDialog(null);
    File rutaFoto = selectorArchivo.getSelectedFile();
    ImageIcon foto = new ImageIcon(rutaFoto.getAbsolutePath());

    String lecturaCedula =
        JOptionPane.showInputDialog("Cédula del Dueño: ");
    int cedulaDuenho = Integer.parseInt(lecturaCedula);

    boolean pudoAdicionarMascota =
        clinica.adicionarMascota(numero, nombre, foto, cedulaDuenho);
}

```

Operador punto

Uso de  
*JFileChooser*  
y *File* para  
obtener una  
imagen  
(*ImageIcon*)

//continúa



```

        if (pudoAdicionarMascota)
        {
            JOptionPane.showMessageDialog(null,
                "Mascota adicionada");
        }
        else
        {
            JOptionPane.showMessageDialog(null,
                "No se encontró la cédula del dueño,"
                + " o ya existe una mascota con ese número");
        }
    }

    /**
     * Muestra los datos de una mascota
     */
    void mostrarMascota()
    {
        String lecturaNumero =
            JOptionPane.showInputDialog("Número de la mascota: ");
        int numero = Integer.parseInt(lecturaNumero);

        Mascota mascota = clinica.buscarMascota(numero);
        if (mascota == null)
        {
            JOptionPane.showMessageDialog(null,
                "No se encontró una mascota con ese número");
        }
        else
        {
            String datosDuenho = ", sin dueño conocido";
            Cliente duenho = mascota.getDuenho();
            if (duenho != null)
            {
                datosDuenho = ", dueño:" + duenho.getNombre();
            }
            JOptionPane.showMessageDialog(null,
                "Mascota "+mascota.getNombre()+datosDuenho,
                "Datos de la mascota "+
                    mascota.getNumero(),1,mascota.getFoto());
        }
    }
} // fin clase MenuVeterinaria

```

Clase ProgramaVeterinaria:

```

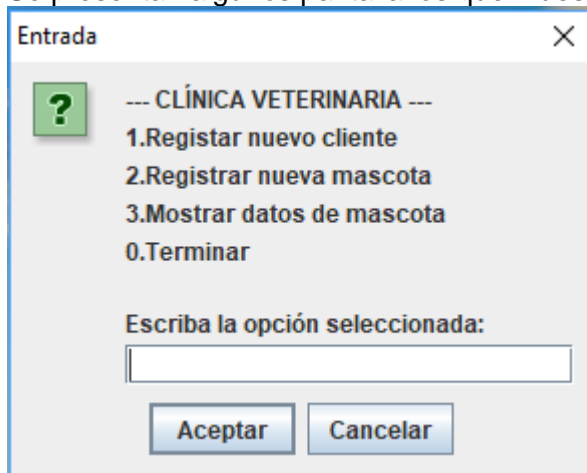
package veterinaria;

/**
 * Ejecuta el programa de la clínica veterinaria
 * @author Sandra V. Hurtado
 * @version 1.0
 */
public class ProgramaVeterinaria
{
    public static void main(String[] args)
    {
        MenuVeterinaria menuPrincipal = new MenuVeterinaria();
        menuPrincipal.menu();
    }
}

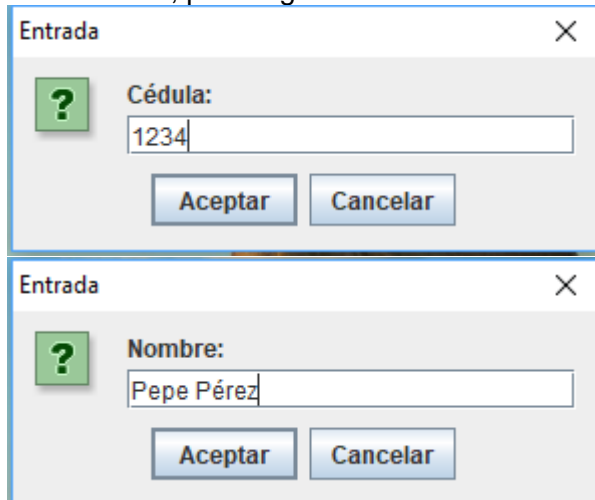
```

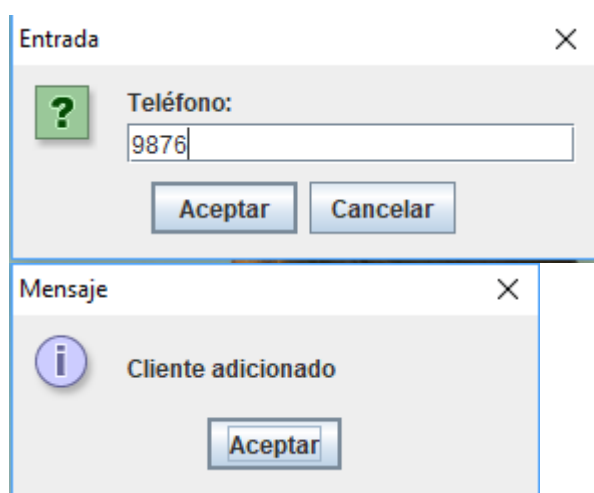
## 21.4. Ejecución del programa

Se presentan algunos pantallazos que muestran la ejecución del programa.

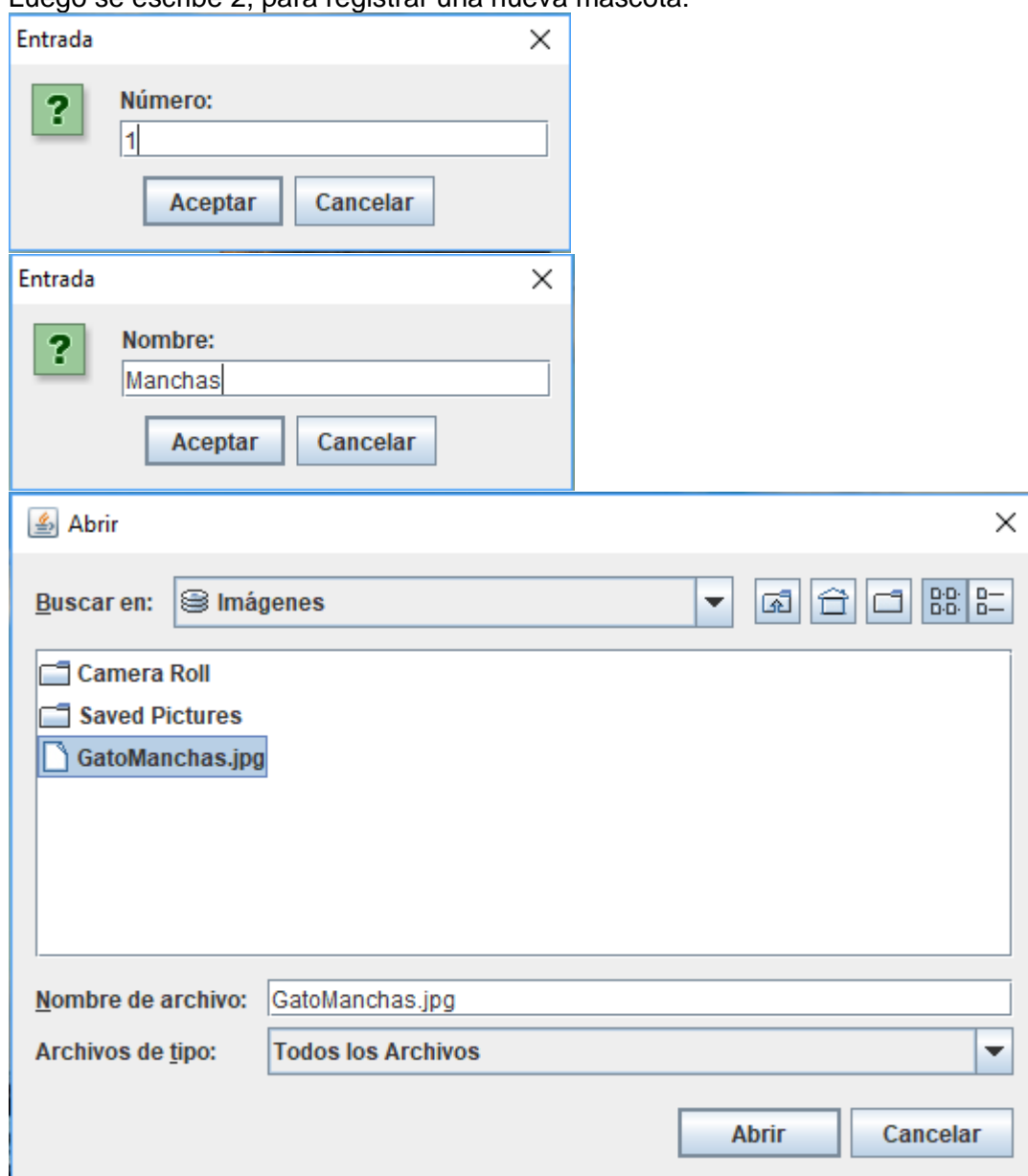


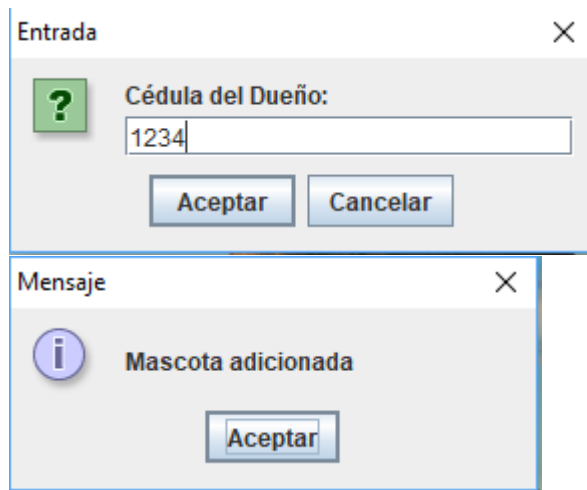
Se escribe 1, para registrar un nuevo cliente:



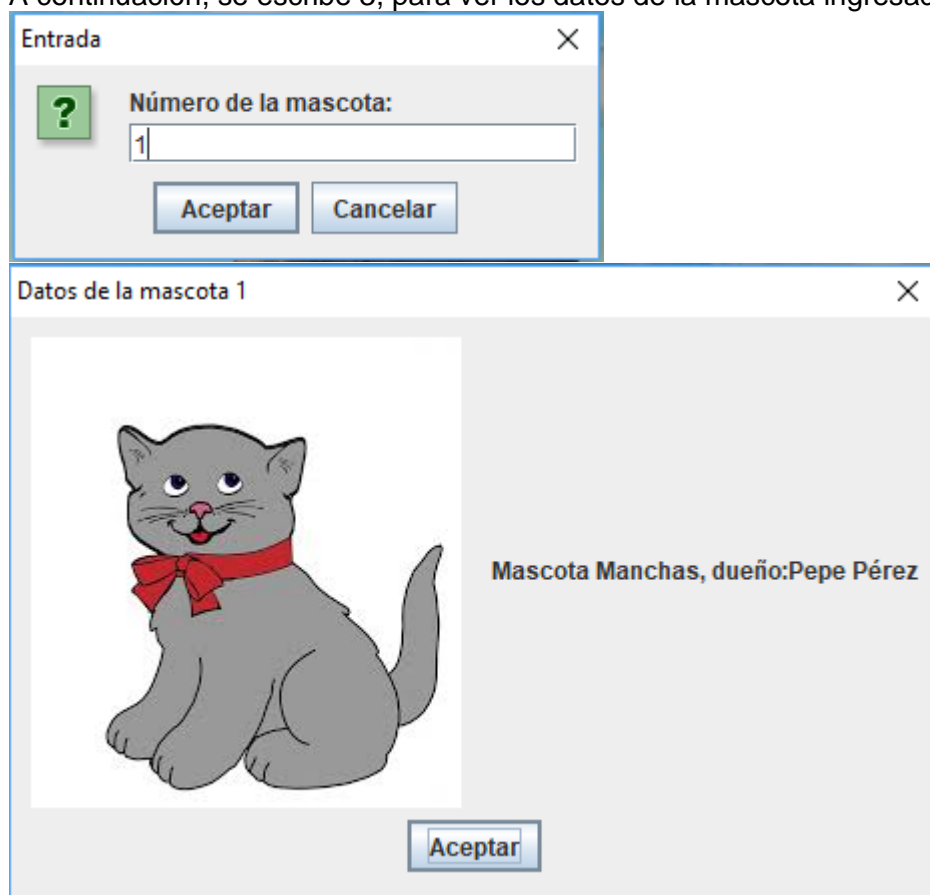


Luego se escribe 2, para registrar una nueva mascota:

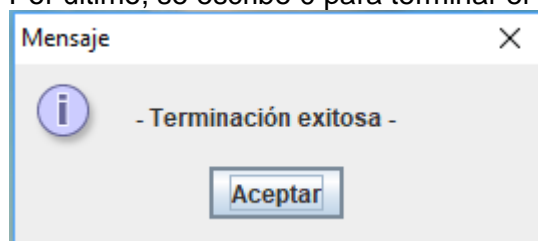




A continuación, se escribe 3, para ver los datos de la mascota ingresada:



Por último, se escribe 0 para terminar el programa:



## Bibliografía

- Gallardo, Raymond; Hommel, Scott; Kannan Sowmya; Gordon, Joni y Zakhour, Sharon Biocca. The Java Tutorial: A Short Course on the Basics (6th Edition). Oracle, 2014. Disponible en Internet: <https://docs.oracle.com/javase/tutorial/?sess=16e492aba137894101940f7f88d9f51f>
- Hurtado Gil, Sandra Victoria. Conceptos Avanzados de Programación con Java. Icesi, 2002.
- Jiménez Collazos, Luz Elena. Conceptos Básicos de Programación con Java. Icesi, 2002.
- Oracle. Java™ Platform, Standard Edition 8 API Specification. Oracle, 2018. Disponible en Internet: <https://docs.oracle.com/javase/8/docs/api/>