

## **Tema 4**

# **Programación orientada a objetos en Java**

**Programación (Desarrollo de aplicaciones web)    Carlos Alberto Cortijo Bon**



Esta obra está bajo una [Licencia Creative Commons Atribución-NoComercial-SinDerivadas 4.0 Internacional](https://creativecommons.org/licenses/by-nc-nd/4.0/)

## Índice

1. <a href="#">Principios básicos de la programación orientada a objetos</a> .....	1
2. <a href="#">Características de los objetos</a> .....	2
3. <a href="#">Elementos de una clase</a> .....	3
4. <a href="#">Atributos</a> .....	6
5. <a href="#">Constructores</a> .....	8
6. <a href="#">Métodos</a> .....	9
7. <a href="#">Visibilidad</a> .....	11
8. <a href="#">Métodos de la clase Object</a> .....	12
8.1. <a href="#">Método toString()</a> .....	12
8.2. <a href="#">Método equals (Object obj)</a> .....	13
9. <a href="#">Creación de clases</a> .....	14
10. <a href="#">Tipos enumerados</a> .....	21
10.1. <a href="#">Tipos enumerados básicos</a> .....	21
10.2. <a href="#">Uso de tipos enumerados básicos</a> .....	22
10.3. <a href="#">Iteración sobre todos los valores de un tipo enumerado</a> .....	24
10.4. <a href="#">Tipos enumerados avanzados</a> .....	24
11. <a href="#">Herencia</a> .....	26
11.1. <a href="#">Clases abstractas</a> .....	29
11.2. <a href="#">Atributos y métodos protegidos (protected)</a> .....	33
12. <a href="#">Interfaces</a> .....	33

Hasta el momento se han utilizado las clases disponibles en la biblioteca estándar de clases de Java. Esta contiene una gran cantidad de clases diversas de uso general. También se han creado clases que contenían el método estático `main`, como punto de entrada de la aplicación, y como mucho algún otro método estático más.

Ahora ha llegado el momento de sacar todo el partido de la programación orientada a objetos. En este tema se aprenderá a desarrollar nuevas clases de utilidad general que puedan ser utilizadas para diversas aplicaciones.

## 1. Principios básicos de la programación orientada a objetos

La programación orientada a objetos (POO) comprende una serie de técnicas que facilitan el diseño, implementación y mantenimiento de programas de ordenador.

Los datos de tipo elemental, como por ejemplo entero (`int`), carácter (`char`) o real (`float` o `double`) tienen un valor sencillo y con el que solo se puede operar mediante un conjunto cerrado de operaciones implementadas en el propio lenguaje.

Frente a los datos elementales están los objetos, pertenecientes a una clase. Los conceptos fundamentales en POO son los de **clase** y **objeto**. Cada objeto pertenece a una clase. También se dice que es una instancia de una clase a la que pertenece. Una clase incluye:


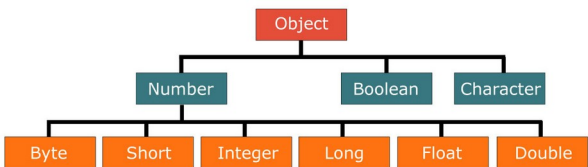

- Un conjunto de datos o **atributos**, que contienen información relativa al objeto.
- Un conjunto de **métodos**, que se pueden ejecutar para un objeto en particular. Estos tienen acceso a los atributos de la clase, tanto para leer su valor como para modificarlos.

Tanto el conjunto de atributos como el de métodos de una clase son abiertos. Se pueden añadir a una clase nuevos atributos y nuevos métodos.

Las clases se pueden estructurar en una **jerarquía de clases**. Una clase se puede definir como subclase o clase hija de otra, su superclase o clase madre, de la que hereda propiedades y métodos. Pero una subclase puede añadir nuevas propiedades y métodos, así como redefinir los métodos existentes en su superclase.

Por ejemplo, una publicación tiene un título y uno o varios autores, así como un número de páginas y un precio. Está además publicada por una editorial en un año. Se puede crear una clase `Publicacion` para representar una publicación. Esta tendría atributos para los datos antes mencionados. Podría tener un método `getCita` para obtener una cita o breve descripción de la publicación, que incluye alguna de esta información. Puede haber varios tipos de publicaciones. Por ejemplo, un libro (clase `Libro`), que además tiene un identificador único universal, el ISBN, y un año de publicación. Y una revista (clase `Revista`), que tiene una fecha de publicación, así como un identificador único universal, el ISSN. Pueden crearse instancias de las clases anteriores para representar libros y revistas. Pero también pueden crearse instancias de la clase `Publicacion` para representar publicaciones que no son un libro ni son una revista. En las clases `Libro` y `Revista` se puede redefinir (*override* en inglés) el método `getDescripcion` para que se presente la descripción de una manera distinta, incluyendo información específica de estas clases. Por ejemplo, para un libro la editorial, y para una revista el número.

Esta forma de modelar el mundo resulta muy natural y conveniente. Todos los lenguajes de programación importantes hoy en día permiten la programación orientada a objetos.

	<p>En Java, la clase <code>Object</code> es la clase madre de todas las clases, el origen de la jerarquía de clases. Por ejemplo, las clases <code>wrapper</code> de Java son, como cualquier clase que se defina con Java, descendientes de <code>Object</code>.</p> <div data-bbox="550 1697 1141 1863">  <pre> graph TD     Object[Object] --&gt; Number[Number]     Object --&gt; Boolean[Boolean]     Object --&gt; Character[Character]     Number --&gt; Byte[Byte]     Number --&gt; Short[Short]     Number --&gt; Integer[Integer]     Number --&gt; Long[Long]     Number --&gt; Float[Float]     Number --&gt; Double[Double] </pre> </div>
	<p>En otros lenguajes de programación como por ejemplo C++, no existe una clase de la que todas las demás son descendientes.</p>

## 2. Características de los objetos

Cabe distinguir, en relación a los objetos, las siguientes características:

- **Identidad.** Es una característica que permite distinguir cada objeto individual de cualquier otro objeto. Dos objetos son iguales si tienen igual valor para todos sus atributos. Pero aún así pueden no ser idénticos. Lo que determina la identidad de un objeto puede variar según el lenguaje de programación o el entorno en el que se utiliza. Puede ser la dirección de memoria en que se almacena el objeto, o un identificador único que se asigna a cada objeto en el momento de su creación. La identidad de un objeto no cambia en ningún momento, es siempre la misma desde el momento en que se crea hasta el momento en que deja de existir.

La identidad en Java se comprueba con el operador `==`.

Dos objetos son idénticos si están situados en la misma posición de memoria. Una variable cuyo tipo es una clase es en realidad un puntero o referencia a una dirección de memoria en la que está almacenado el objeto.

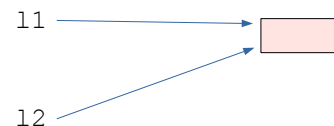
Cuando se asigna a un objeto otro objeto ya existente, no se crea un nuevo objeto, sino que se le asigna un puntero o referencia a la misma dirección de memoria.

La igualdad en Java se comprueba con el método `equals` de la clase `Object`. Esta se puede redefinir en cualquier clase. Este método devuelve `true` cuando los valores de todos los atributos son iguales.

```
Libro l1 = new Libro("La Odisea", "Homero");  
Libro l2 = l1;
```



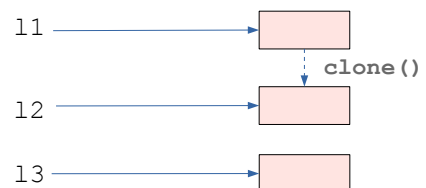
`l1 == l2` devuelve `true`.  
`l1.equals(l2)` devuelve `true`.



Se puede crear una copia de un objeto con el método `clone` de la clase `Object`. El objeto así creado será igual, pero no idéntico, porque es otro objeto.

```
Libro l1 = new Libro("La Odisea", "Homero");  
Libro l2 = l1.clone();  
Libro l3 = new Libro("La Odisea", "Homero");
```

`l1 == l2`, `l1 == l3`, `l2 == l3` devuelven `false`.  
`l1.equals(l2)`, `l1.equals(l3)`, `l2.equals(l3)`, devuelven `true`.



- **Estado.** Viene determinado por el valor de sus atributos. El estado de un objeto normalmente cambia a lo largo del tiempo. Normalmente existen métodos que permiten conocer el estado de un objeto. Pueden devolver el valor de un atributo determinado, o bien proporcionar información acerca del estado que depende de los valores de varios atributos. Por ejemplo, podrían haber métodos que devuelvan el título, y el autor o autores de una publicación. Podría haber también un método que devuelva `true` si una publicación está descatalogada (es decir, que ya no se publica), si la clase tuviera atributos a partir de los cuales se pudiera saber.
- **Comportamiento.** Son las acciones que se pueden realizar sobre el objeto. Corresponden a los métodos definidos para la clase del objeto que permiten obtener información acerca de su estado, y también cambiarlo.

### El valor `null`

Ya se ha comentado que una variable definida como perteneciente a una clase es en realidad una referencia a un objeto. Existe un valor especial, `null`, que no hace referencia a ningún objeto. Se puede asignar este valor especial a una variable de cualquier clase. El valor de la expresión `v == null` es `true` si `v` tiene asignado el valor `null`, y `false` en caso contrario.

**Actividad 4.1**

En un programa en Java se tienen las siguientes declaraciones de variables.

```
Object v1 = null;  
Object v2 = null;
```

¿Qué valor piensas que devolverá la expresión `v1 == v2`? Verifícalo.

### 3. Elementos de una clase

Ya se han utilizado clases de la biblioteca estándar de clases de Java. Estas proporcionan funcionalidad básica y de utilidad general para todas las aplicaciones que se puedan desarrollar en Java. Un ejemplo es la clase `String`. Ahora se verá cómo un programador puede crear sus propias clases. Para ello, se empezará comentando un ejemplo de clase sencillo.

La definición de la una clase de nombre `Personaje` se situaría dentro de un bloque con la siguiente forma.

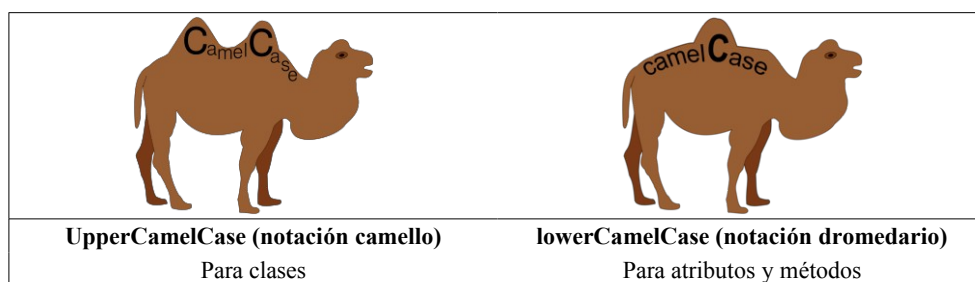
```
class Personaje {  
    (...)  
}
```

Dentro de este bloque se definen:

- Atributos. Son definiciones de variables. En una definición de variable se indica primero el tipo de la variable y después su nombre. El estado del objeto viene dado por los valores para sus variables.
- Métodos. Son bloques de código que contienen instrucciones. En los métodos se pueden utilizar los atributos o variables definidas en la clase.

A continuación se muestra la definición de una clase `Personaje` que representa un personaje que se puede mover por un tablero de juego de dimensiones fijas. Al crear un personaje se le asignan un nombre y una posición dentro del tablero. El nombre no cambia una vez creado el objeto. Su posición sí puede cambiar a lo largo del tiempo.

Se sigue el convenio de notación en forma de joroba de camello (*Camel Case*) para todos los nombres: de clases, de atributos y de métodos. Las iniciales de las palabras están en mayúsculas. Con la excepción de la inicial de la primera palabra. Con `UpperCamelCase` o notación camello está en mayúsculas. Con `lowerCamelCase` o notación dromedario está en minúsculas. El propio lenguaje no obliga a ello, pero es una práctica universalmente extendida.



En la definición de la clase y de sus distintos elementos se utilizan determinadas opciones (como por ejemplo `public`, `private`, `static`, o `final`). Se irán explicando todas, una tras otra. Entretanto, se pueden ignorar para centrarse en lo que se está explicando en cada momento.

```
package com.tusjuegos.juego;  
  
public class Personaje {  
  
    // Tipo enumerado para uso con método avanza. Por eso se declara dentro de la clase.
```

```
public enum Direccion {
    IZQUIERDA, DERECHA, ARRIBA, ABAJO
}

// Atributos
public static final int MAX_X = 9;
public static final int MAX_Y = 9;
private static int numObj = 0;

private final String nomPers;
private int x;
private int y;

// Métodos
public Personaje(String nombre, int x, int y) { // Método constructor
    this.nomPers = nombre;
    this.x = x;
    this.y = y;
    Personaje.numObj++;
}

public String getNomPers() {
    return nomPers;
}

public int getX() {
    return x;
}

public int getY() {
    return y;
}

public static int getNumObj() {
    return numObj;
}

public void avanza(Direccion mov) {
    switch (mov) {
        case IZQUIERDA:
            if (x > 0) {
                x--;
            }
            break;
        case DERECHA:
            if (x < MAX_X) {
                x++;
            }
            break;
        case ARRIBA:
            if (y > 0) {
                y--;
            }
            break;
    }
}
```

```
        }
        break;
    case ABAJO:
        if (y < MAX_Y) {
            y++;
        }
        break;
    }
}

public static void main(String[] args) { // Método main

    Personaje p = new Personaje("Bicho", 2, 3);

    System.out.printf("Posicion inicial: (%d, %d)\n", p.getX(), p.getY());

    p.avanza(Direccion.ARRIBA);
    p.avanza(Direccion.IZQUIERDA);
    p.avanza(Direccion.IZQUIERDA);
    p.avanza(Direccion.ARRIBA);
    p.avanza(Direccion.IZQUIERDA);
    p.avanza(Direccion.DERECHA);
    p.avanza(Direccion.ABAJO);

    System.out.printf("Posicion final: (%d, %d)\n", p.getX(), p.getY());

    Personaje q = new Personaje("Cazabichos", 6, 5);

    System.out.printf("Número de personajes: %d\n", Personaje.getNumObj());

}
```

El tipo enumerado `Direccion` se utiliza en el método `avanza` de la clase `Personaje`. Al haberse definido dentro de esta clase como `public`, estará disponible, junto con este método, desde cualquier otra clase. Se podrá hacer referencia a él con `Personaje.Direccion`. De esta manera, además, se evita cualquier posible ambigüedad con cualquier otra clase con el mismo nombre que se pueda haber definido en cualquier otra parte. Como ya se comentó, los tipos enumerados son, en realidad, clases. Lo mismo que los *arrays*.

### Recuerda

En Java todo son clases, y nada puede existir que no pertenezca a una clase. Con la excepción de los tipos básicos `int`, `long`, `short`, `float`, `double`, `byte`, `char` y `boolean`. Y para ellos, además, existen las correspondientes clases *wrapper* `Integer`, `Long`, `Short`, `Float`, `Double`, `Byte`, `Character`, `Boolean`.

Cuando se ejecuta el programa anterior, se ejecuta el método `main`, y su salida es la siguiente. Se escribe la posición inicial del personaje y la final después de realizar todos los desplazamientos con el método `avanza`. Después se crea otro objeto y se muestra el número de objetos de la clase `Personaje` que se han creado.

```
Posicion inicial: (2, 3)
Posicion final: (1, 2)
Número de personajes: 2
```

Toda clase pertenece a un **paquete** (*package*), que se especifica al principio con `package`.

## Paquetes

Los paquetes se estructuran en jerarquías. El nombre del paquete debe incluir la jerarquía completa. `com.tusjuegos.juego` significa el paquete `juego` dentro del paquete `tusjuegos` dentro del paquete `com`.

Esta jerarquía de paquetes se refleja en una jerarquía de directorios que contienen las clases del proyecto. Se muestra a continuación la vista de proyectos y la vista de ficheros en NetBeans para el proyecto creado para el programa anterior.

Vista de proyectos	Vista de ficheros
<ul style="list-style-type: none"> <li>Personaje           <ul style="list-style-type: none"> <li>Source Packages               <ul style="list-style-type: none"> <li>com.tusjuegos.juego                   <ul style="list-style-type: none"> <li>Personaje.java</li> </ul> </li> </ul> </li> <li>Test Packages</li> <li>Libraries</li> <li>Test Libraries</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>Personaje           <ul style="list-style-type: none"> <li>nbproject</li> <li>src               <ul style="list-style-type: none"> <li>com                   <ul style="list-style-type: none"> <li>tusjuegos                       <ul style="list-style-type: none"> <li>juego                           <ul style="list-style-type: none"> <li>Personaje.java</li> </ul> </li> </ul> </li> </ul> </li> <li>test</li> <li>build.xml</li> <li>manifest.mf</li> </ul> </li> </ul> </li></ul>

También existen jerarquías de paquetes para las clases de la biblioteca estándar de clases de Java. La clase `String`, por ejemplo, pertenece al paquete `java.lang`, cuyas clases están empaquetadas en un JDK incluido en el proyecto que se crea para el programa. Otros paquetes incluidos en el JDK que contienen clases estándares de Java son, por ejemplo, `java.io`.

Las jerarquía de paquetes del JDK, y sus clases, se pueden ver en el apartado “Libraries”.

En los ficheros de tipo jar se empaquetan clases, que se pueden añadir a proyectos en los apartados “Libraries” o “Test Libraries”. También contienen las clases en una jerarquía de paquetes.

<ul style="list-style-type: none"> <li>Personaje           <ul style="list-style-type: none"> <li>Source Packages</li> <li>Test Packages</li> <li>Libraries               <ul style="list-style-type: none"> <li>JDK 17 (Default)                   <ul style="list-style-type: none"> <li>java.base                       <ul style="list-style-type: none"> <li>&lt;default package&gt;</li> <li>META-INF.services</li> <li>com.sun.crypto.provider</li> <li>com.sun.security.ntlm</li> <li>java.io</li> </ul> </li> <li>java.lang                       <ul style="list-style-type: none"> <li>AbstractMethodError.class</li> <li>AbstractStringBuilder.class</li> <li>Appendable.class</li> <li>ApplicationShutdownHooks\$1.class</li> </ul> </li> </ul> </li> </ul> </li> </ul> </li></ul>	<ul style="list-style-type: none"> <li>Personaje           <ul style="list-style-type: none"> <li>Source Packages</li> <li>Test Packages</li> <li>Libraries               <ul style="list-style-type: none"> <li>JDK 17 (Default)</li> <li>Test Libraries                   <ul style="list-style-type: none"> <li>JUnit 5.6.0 - junit-jupiter-api-5.6.0.jar</li> <li>JUnit 5.6.0 - junit-jupiter-params-5.6.0.jar</li> <li>JUnit 5.6.0 - junit-jupiter-engine-5.6.0.jar</li> <li>hamcrest-core-1.3.jar</li> <li>junit-4.13.2.jar                       <ul style="list-style-type: none"> <li>&lt;default package&gt;</li> <li>META-INF</li> <li>junit.extensions</li> <li>junit.framework</li> <li>junit.runner</li> </ul> </li> </ul> </li> </ul> </li> </ul> </li></ul>
---	---

El método `main` es un método especial. Este es el punto de entrada del programa. Se ejecuta cuando se ejecuta el programa. En él se crea un objeto de la clase `Personaje`.

```
Personaje p = new Personaje("Bicho", 2, 3);
```

Una vez creado, se puede mover por el tablero con el método `avanza`. En cualquier momento se pueden obtener las coordenadas `x` e `y` de su posición con `getX()` y `getY()`.

A continuación se explican los distintos elementos de una clase, tomando como ejemplo la clase `Personaje`.

## 4. Atributos

Los atributos son variables que se definen para una clase.

```
public static final int MAX_X = 9;
public static final int MAX_Y = 9;
```



```
private static int numObj = 0;

private final String nomPers;
private int x;
private int y;
```

Cabe distinguir dos tipos de atributos:

- **Estáticos.** Estos se definen con la opción **static**. Estos tienen un valor que es único y global para toda la clase. Es decir, son atributos o variables de clase. Para acceder a su valor se antepone a su nombre el nombre de la clase (Personaje) y un punto: Personaje.MAX\_X, Personaje.MAX\_Y, Personaje.numObj. Se pueden escribir sus valores con el siguiente código.

```
System.out.println(Personaje.MAX_X);
System.out.println(Personaje.MAX_Y);
System.out.println(Personaje.MnumObj);
```

- En la clase hay dos atributos estáticos finales (**static final**): MAX\_X y MAX\_Y. Ambos son de tipo **int** y se les asigna un valor inicial de 9. Estos son los valores máximos de los atributos x e y, respectivamente. El mínimo valor es 0. Por tanto, el tablero tiene unas dimensiones de 10x10.

```
public static final int MAX_X = 9;
public static final int MAX_Y = 9;
```

- Hay además otro atributo estático pero no final (**static**), numObj. Este sirve para llevar la cuenta del número de objetos de la clase que se han creado. Inicialmente tiene valor 0.

```
private static int numObj = 0;
```

- **No estáticos.** El resto de atributos. Estos tienen un valor que es distinto para cada objeto o instancia de la clase. Es decir, son atributos o variables de instancia. Por ello, para acceder a su valor se antepone el nombre del objeto y un punto.

En la clase hay tres atributos no estáticos: nombre, x e y.

```
private final String nomPers;
private int x;
private int y;
```

Se puede crear una instancia de la clase Persona y escribir el valor de su atributo nombre con el siguiente código.

```
Personaje p = new Personaje("Bicho", 0, 3);
System.out.printf("Nombre del personaje: %s\n", p.nombre);
```

**NOTA:** En realidad, esto último NO se puede hacer, porque este atributo se han definido como privado, con la opción **private**. Si se hubiera definido como **public**, como se ha hecho con MAX\_X y MAX\_Y (lo que, por otra parte, no sería una buena idea, como se explicará en breve), sí se podría. En breve se explicará más acerca de las opciones de visibilidad, entre las que están **public** y **private**.

Sus nombres siguen el convenio de la joroba de camello (*camel case notation*). La primera letra es en minúscula, y la inicial de cada palabra en mayúscula.

También se ha utilizado la opción **final** para algunos atributos. El valor de un atributo o variable definida como final no se puede cambiar una vez que se le ha asignado un valor. Se define con esta opción el atributo nomPers. Este es el nombre del personaje, y no se contempla que cambie su valor una vez que se asigna al crear el objeto.

Los atributos estáticos y finales (**static final**) son constantes de clase. Son atributos de la clase, y su valor no puede cambiar. En la clase Personaje son global MAX\_X y MAX\_Y. Para ellas se sigue la notación *Snake Case*: las palabras se separan mediante guiones bajos. Y además todas sus letras están en mayúsculas. Son la única excepción a la notación *Camel Case*. El atributo nomPers, en cambio, es estático pero no constante (**final**), y para el se utiliza la notación lowerCamelCase.



## 5. Constructores

Un constructor es un tipo particular de método que se llama cuando se crea un objeto. Tiene el mismo nombre que la clase. Normalmente, en un constructor se asignan los valores iniciales de los atributos del nuevo objeto que se está creando.

Existe un constructor en la clase de ejemplo `Personaje`.

```
public Personaje(String nombre, int x, int y) { // Método constructor
    this.nomPers = nombre;
    this.x = x;
    this.y = y;
    Personaje.numObj++;
}
```

Este constructor asigna valores iniciales a los atributos del objeto que se está creando. Los valores son los que se pasan en los parámetros del constructor `x`, `y`, `nombre`. Como son atributos de instancia y no de clase, están accesibles en **this**, que hace referencia al objeto sobre el que se ejecuta el método. Por ello se antepone **this** a su nombre.

Los atributos de clase (**static**) están accesibles en `Personaje`, es decir, en la clase en la que están definidos. Por ello se antepone `Personaje.` a su nombre. En este constructor se incrementa su valor, para reflejar el hecho de que se ha creado una nueva instancia de la clase.

### El identificador **this**

**this** representa el objeto sobre el que se ejecuta un método no estático de una clase. No está disponible en métodos estáticos. Por ejemplo, **this.x** representa el valor del atributo o variable `x` para el objeto sobre el que se ejecuta el método. O en otras palabras, el valor de la variable de instancia `x`.

Normalmente no es necesario su uso. Como tampoco es normalmente necesario anteponer el nombre de la clase al nombre de variables estáticas. El constructor de la clase `Personaje` podría haberse escrito también de la siguiente forma, y haría exactamente lo mismo.

```
public Personaje(String nombre, int posX, int posY) { // Método constructor
    nomPers = nombre;
    x = posX;
    y = posY;
    numObj++;
}
```

En este código, `x` no puede representar más que el valor de la variable de instancia `x`, es decir, **this.x**. Y `numObj` no puede representar más que el valor de la variable de clase `numObj`.

Por supuesto, se podría utilizar **this.x**, **this.y**, **this.nomPers** y `Personaje.numObj`.

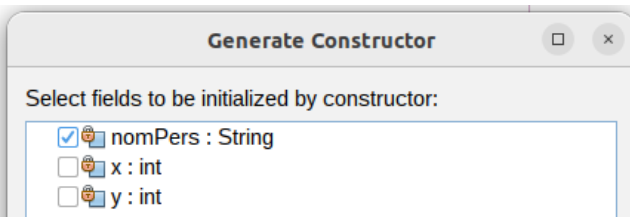
Pero si el nombre del parámetro del constructor fuera `x` en lugar de `posX`, entonces `x` representaría el valor del parámetro del constructor con ese nombre. Y para referirse al valor de la variable de instancia con ese nombre, la única posibilidad sería **this.x**.

### Ayudas para la creación de constructores

Los entornos de desarrollo suelen proporcionar herramientas para facilitar la creación de los constructores de una clase, una vez que se han definido sus atributos. En Netbeans, por ejemplo, se pueden crear pulsando con el botón derecho del ratón sobre el editor, seleccionando la opción "Insert Code" y después "Constructor...". Con ello se muestra un cuadro de diálogo donde se pueden elegir los atributos para los que se quiere asignar valor en el constructor. Por defecto solo aparecen los finales (**final**), pero se puede seleccionar el resto. No aparecen en la lista los estáticos (**static**).

```
public static final int MAX_X = 10;
public static final int MAX_Y = 10;
private static int numObj = 0;

private final String nomPers;
private int x;
private int y;
```



Si se seleccionan todos los atributos de la lista, se genera el siguiente constructor, que luego se puede adaptar a conveniencia.

```
public Personaje(String nomPers, int x, int y) {
    this.nomPers = nombre;
    this.x = x;
    this.y = y;
}
```

#### Actividad 4.2

Añade a la clase un constructor al que solo se le pasa el nombre del personaje, y que lo sitúa en la posición (0, 0).

#### Actividad 4.3

Al constructor de la clase `Personaje` se le pueden pasar coordenadas iniciales que están fuera de los límites del tablero. Cambia el constructor para que, en este caso, se sitúe al personaje en la posición más cercana dentro del tablero. Solo hay que modificar la coordenada o las coordenadas que están por debajo del valor mínimo (0) o por encima del valor máximo (`MAX_X` para `x` o `MAX_Y` para `y`).

Nota. El inconveniente de este planteamiento es que no hay manera de detectar el error en los parámetros de entrada, como no sea comprobando la posición una vez creado el `Personaje`. Más adelante se verá una manera mejor de gestionar problemas de este tipo, mediante el lanzamiento de una excepción.

## 6. Métodos

Un método tiene un nombre, unos parámetros y un tipo de valor de retorno.

Sus nombres siguen el convenio de la joroba de camello (*camelback notation*). La primera letra es en minúscula, y la inicial de cada palabra en mayúscula.

Hay varios métodos de diversos tipos en la clase de ejemplo `Personaje`, dependiendo de algunas opciones que se pueden incluir al principio de su definición. Dejando de lado el constructor, que ya se ha visto, los métodos son los siguientes.

Opciones	Tipo de valor de retorno	nombre	parámetros
<code>public</code>	<code>String</code>	<code>getNomPers</code>	(Ninguno)
<code>public</code>	<code>int</code>	<code>getX</code>	(Ninguno)
<code>public</code>	<code>int</code>	<code>getY</code>	(Ninguno)
<code>public static</code>	<code>int</code>	<code>getNumObj</code>	(Ninguno)
<code>public</code>	<code>void</code>	<code>avanza</code>	Direccion mov
<code>public static</code>	<code>void</code>	<code>main</code>	<code>String[] args</code>

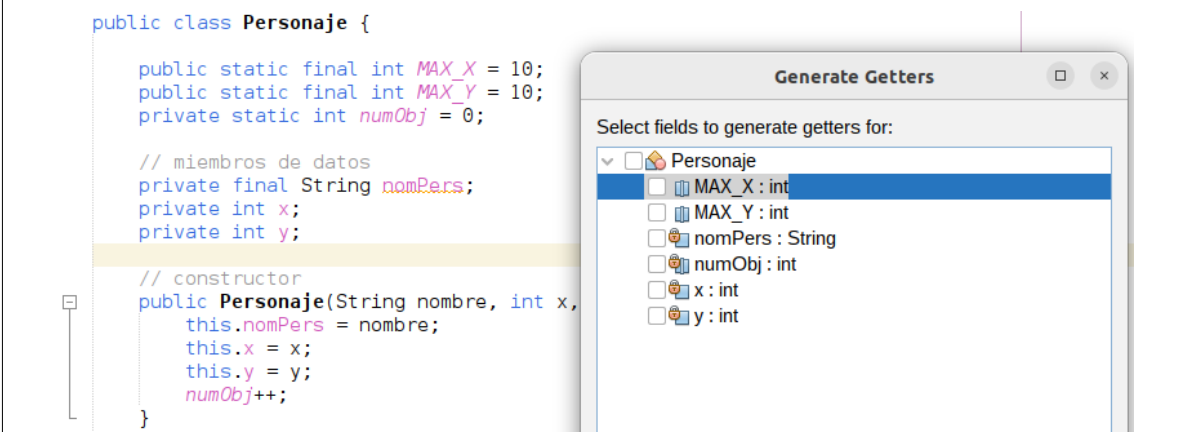
- **Opción `public`.** Indica que el método es accesible desde cualquier clase. Hay otras opciones que se verán más adelante. En esta clase, todos los métodos son públicos.

- **Opción `static`.** Ya se han explicado los atributos estáticos. Son atributos de la clase, y no de sus instancias particulares. Para acceder a su valor, hay que anteponer el nombre de la clase. Por ejemplo: `Personaje.MAX_X`. Los métodos estáticos son métodos que no actúan sobre instancias particulares de la clase. Para referirse a ellos hay también que anteponerles el nombre de la clase seguido de un punto. Los métodos estáticos no tienen acceso directo a los atributos de la clase.
- **Tipo de valor de retorno.** Es el tipo de valor que devuelve el método cuando se ejecuta. El método debe devolver un valor del tipo indicado, con la sentencia `return`. Si el tipo de valor de retorno es `void`, el método no devuelve ningún valor.
- **Métodos `getter`.** Son métodos públicos (`public`) cuyo nombre empieza por `get` y termina con el nombre de un atributo. Devuelven el valor del atributo. Su nombre debe seguir también el convenio de la notación de joroba de camello. Por lo tanto, la primera letra después de `get` debe estar en mayúsculas.

Si el atributo del que devuelven el valor es estático (`static`), el método `getter` debe ser también estático. Es el caso de `getNumObj`.

#### Ayudas para la creación de métodos `getter`

Se suelen definir `getters` para todos o casi todos los atributos. Esta es una tarea repetitiva y mecánica, los entornos de desarrollo suelen proporcionar opciones para crearlos. En Netbeans, por ejemplo, se pueden crear pulsando con el botón derecho del ratón sobre el editor, seleccionando la opción “Insert Code” y después “Getter...”. Con ello se muestra un cuadro de diálogo donde se pueden elegir los atributos para los que se quiere crear un método `getter`. No siempre es aconsejable crearlos para todos los atributos. Algunos pueden ser para uso interno y puede no ser necesario, o no ser conveniente, o incluso ser inconveniente, que se pueda conocer su valor fuera del propio objeto.



- **Método `main`.** Es un método especial. Es el punto de entrada del programa. Cuando se ejecuta el programa, se ejecuta el método `main`. Este método debe tener siempre las siguientes características:
  - Es estático (`static`). Esto evita la necesidad de crear un objeto de la propia clase solo para que el programa se pueda ejecutar.
  - Tiene un parámetro `String[] args`, de donde se pueden obtener los argumentos de línea de comandos. En estos, el usuario puede pasar datos al programa cuando lo ejecuta. Por ejemplo, números cuya suma programa debe mostrar, nombres de ficheros que debe utilizar, o credenciales de usuario (nombre y contraseña) que debe utilizar para acceder a determinados servicios.
  - El valor de retorno es `void`. Este es el método que se ejecuta al ejecutar el programa y no tiene ningún método al que devolver ningún valor.



En Java solo existen métodos que pertenecen a clases. El método `main`, que es el punto de entrada del programa, debe pertenecer a una clase en Java. En C++ pueden existir funciones que no pertenecen a ninguna clase, además de funciones o métodos que pertenecen a una clase. En C++ el método `main` es también el punto de entrada del programa, pero no pertenece a ninguna clase. No podría ser de otra forma, con el planteamiento de C++ como un superconjunto de C o como un C con clases. Cualquier programa válido en C es válido en C++, pero además en C++ se pueden usar clases. Si en C++ el método `main` tuviera que pertenecer a una clase, ningún programa válido en C lo sería en C++.

- **Otros métodos no estáticos.** En general, modifican el estado del objeto. El único es `avanza`, que modifica la posición (dada por los atributos `x` e `y`) del objeto.
- **Otros métodos estáticos (`static`).** No hay, aparte de los ya vistos `main` y `getNumObj`.

#### Actividad 4.4

Cambia el método `avanza` para que devuelva `true` si el movimiento se ha podido realizar, y `false` en caso contrario. Por ejemplo, si el personaje está en el extremo izquierdo del tablero y se intenta avanzar hacia la izquierda.

## 7. Visibilidad

Antes de la palabra reservada `class` se puede especificar la visibilidad de la clase. Las posibles opciones son:

<code>public</code>	Esta clase la puede utilizar cualquier otra clase.
<i>(Sin especificar)</i>	Esta clase la pueden utilizar clases del mismo paquete.

En general, se definirán como `public` solo las clases que se vayan a utilizar en otras clases. Y también cualquiera que contenga un método `main` y que, por tanto, se pueda ejecutar cuando se ejecuta el programa al que pertenece. Para el resto, se omite la opción de visibilidad.

También se pueden especificar la visibilidad de los elementos de una clase, es decir, de sus atributos y métodos. La siguiente tabla muestra la visibilidad en distintos ámbitos para cada opción.

	Misma clase	Clases del mismo paquete	Subclases	Otras clases
<code>public</code>	✓	✓	✓	✓
<code>protected</code>	✓	✓	✓	✗
<i>(Sin especificar)</i>	✓	✓	✗	✗
<code>private</code>	✓	✗	✗	✗

En general se deben definir como `private` los atributos de una clase, de manera que no se puede acceder a ellos desde fuera de la clase. De no ser así, otras clases del mismo paquete podrían cambiar su valor sin restricciones para, por ejemplo, situar el personaje fuera de los límites del tablero, o moverlo arbitrariamente desde cualquier posición a cualquier otra posición. En lugar de ello, se quiere que solo se pueda cambiar su posición utilizando los métodos apropiados, como por ejemplo el método `avanza`.

Si un atributo de una clase se va a utilizar solo desde clases del mismo paquete, se omite la opción de visibilidad.

Para comprender el significado de la opción `protected`, debe conocerse antes la herencia, que se explicará en breve.



En C++, si un atributo o método se define como `private`, solo se puede acceder a él desde el propio objeto, y no desde ningún otro objeto, aunque pertenezca a la misma clase. En Java, en cambio, sí se podrá acceder desde otros objetos de la misma clase.

**Actividad 4.5**

Se quiere mejorar la clase `Personaje` de manera que no esté limitada a un tablero de dimensiones con valores fijos asignados en la propia clase (es decir, *hardcoded*). En lugar de ello, se quiere poder especificar las dimensiones del tablero cuando se crea el personaje. Sustituye las variables estáticas finales `MAX_X` y `MAX_Y` por variables no estáticas pero sí finales `maxX` y `maxY`. A estas variables debe asignárseles un valor que se pasa al constructor cuando se crea un objeto, de manera análoga a como se hace con `nomPers`, `x` e `y`. Estas nuevas variables deben ser privadas (**private**) y debe crearse para ellas su correspondiente método *getter*.

**Actividad 4.6**

Añade a la clase anterior un método `avanzaExt`, al que se le pasa un *array* de tipo `Direccion`, es decir, `Direccion[]`. Este método debe ejecutar los movimientos, uno tras otro.

## 8. Métodos de la clase Object

La clase `Object` tiene varios métodos que es muy importante conocer. Aunque la herencia se explicará en breve, por ahora basta con decir que todas las clases de Java son descendientes de la clase `Object` y que, por tanto, todos los métodos de la clase `Object` están disponibles en cualquier clase de Java. No solo es importante conocerlos y saberlos utilizar correctamente, sino también redefinirlos de manera apropiada, si es conveniente o necesario, en las nuevas clases que se creen.

<code>Object clone()</code>	Crea y devuelve una copia del objeto.
<code>String toString()</code>	Devuelve una representación del objeto en forma de <code>String</code> . Es muy útil para mostrar el estado de un objeto en un formato legible para las personas.
<code>boolean equals(Object obj)</code>	Indica si otro objeto es igual al objeto.
<code>public final Class&lt;?&gt; getClass()</code>	Devuelve la clase a la que pertenece el objeto. El objeto pertenece a una subclase de <code>Object</code> . Las instancias de la clase <code>Class</code> representan clases que se han cargado en el entorno de ejecución ( <i>runtime</i> ) de la máquina virtual de Java.

### 8.1. Método `toString()`

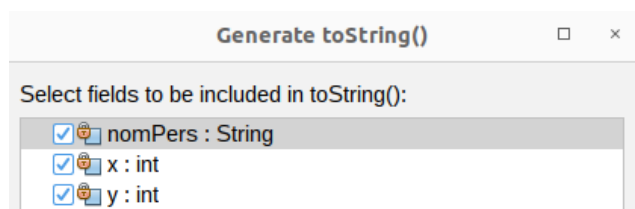
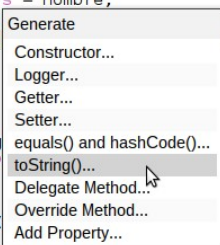
El método `toString` proporciona una representación en forma de texto de un objeto.

Los entornos de desarrollo suelen ofrecer ayudas para redefinir este método. En NetBeans, por ejemplo, está dentro de las opciones del menú “Insert Code” que se muestra al pulsar con el botón derecho sobre el editor de código fuente.

```
// constructor
public Personaje(String nombre, int x, int y) {
    this.nomPers = nombre;
    this.x = x;
    this.y = y;
    numObj++;
}

// métodos
public String toString() {
    return nomPers + " en (" + x + ", " + y + ")";
}

public int getX() {
    return x;
}
```



El código que se crea para el método `toString` es el siguiente, y se puede adaptar a conveniencia.

La anotación `@Override` significa que este método redefine el mismo método de la clase madre (con el mismo nombre, valor de retorno y atributos).

```
@Override
public String toString() {
    return "Personaje{" + "nomPers=" + nomPers + ", x=" + x + ", y=" + y + '}';
}
```

#### Actividad 4.7

Redefine el método `toString` para la clase `Personaje`, de manera que devuelva un `String` de la forma:

`nombre(x, y)`.

Cambia el método `main` del programa de ejemplo para que use el método `toString` cada vez que se quiera mostrar información de un personaje, sea el nombre o la posición.

No hace falta utilizar directamente el método `toString`. Si donde se espera un `String` aparece un objeto, se invoca automáticamente el método `toString` y se utiliza el `String` que devuelve.

## 8.2. Método `equals` (`Object obj`)

El método `equals(Object obj)` debe devolver `true` si el objeto `obj` es igual al objeto sobre el que se aplica, y `false` en caso contrario. Cuando se crea una nueva clase, hay que tener claro, o decidir, cuándo dos objetos de esa clase “son iguales”. Y por supuesto, para decidir si es conveniente redefinir este método en la nueva clase, antes que nada hay que saber cómo está implementado este método en la clase `Object`.

Un objeto es siempre igual a sí mismo. De esta forma, si se cumple que `a == b`, entonces `a.equals(b)` debe devolver `true` porque, como ya se ha visto, `a` y `b` son referencias al mismo objeto. Es decir, la identidad entre dos objetos (el valor de la expresión `a == b` es `true`) implica su igualdad (el valor de la expresión `a.equals(b)` es `true`).

Cuando dos objetos no son idénticos (el valor de la expresión `a == b` no es `true`), se pueden dar casos distintos.

- En una clase `IntervaloEnt`, cuyas instancias representan un intervalo de números enteros entre un número entero  $a$  y otro  $b$ , de forma que  $a \leq b$ , parece lógico considerar que dos instancias de esta clase son iguales cuando el valor del atributo  $a$  es igual para ambos y también lo es el valor del atributo  $b$ .
- Pero una clase puede tener atributos que son irrelevantes en lo que respecta a la igualdad entre dos objetos. Considérese el caso de una clase `Persona` que tenga un atributo `id`. El valor de este atributo podría ser, en España, el DNI o NIE, que es diferente para cada persona. Se puede considerar que dos objetos de esta clase son iguales, es decir, que representan a la misma persona, si el valor de estos atributos es igual, sin importar el valor de cualquier otro atributo, incluyendo el de un atributo `nombre`. El valor de `nombre` debería ser igual si lo es el valor de `id`, por lo que solo es necesario verificar la igualdad del valor de `id`. Podría darse incluso el caso en que el valor de `id` sea igual pero no el de `nombre`. Una instancia de esta clase podría haberse creado a partir de una consulta en una base de datos, y otra a partir de un DNI introducido en una interfaz gráfica de usuario. En esta misma interfaz podría haberse pedido el nombre de la persona a efectos de comprobación, y este podría no coincidir exactamente con el nombre existente en la base de datos.
- Pero no siempre es un criterio válido la igualdad del valor de los atributos. Considérese la clase `Fraccion` con atributos `num` y `den`, cuyas instancias representan una fracción  $\frac{num}{den}$ . Dos fracciones  $\frac{a}{b}$  y  $\frac{c}{d}$  son iguales cuando  $a \cdot d = b \cdot c$ , de manera que el caso en que  $a = c$  y  $b = d$  es solo un caso particular, y dos fracciones pueden ser iguales aunque no coincida el valor de ningún atributo.
- En la clase `Personaje` anteriormente vista, no tiene sentido plantearse que dos instancias no idénticas (`p1` y `p2` cuando `p1 != p2`) puedan ser iguales. Si el valor del atributo `nombre` es igual, la posición (atributos `x` e `y`) de uno de ellos podría cambiar. No tiene sentido plantear que dos instancias de esta clase son iguales cuando están en la misma posición y distintas cuando no. Dos instancias no idénticas de la clase `Personaje` nunca son iguales.



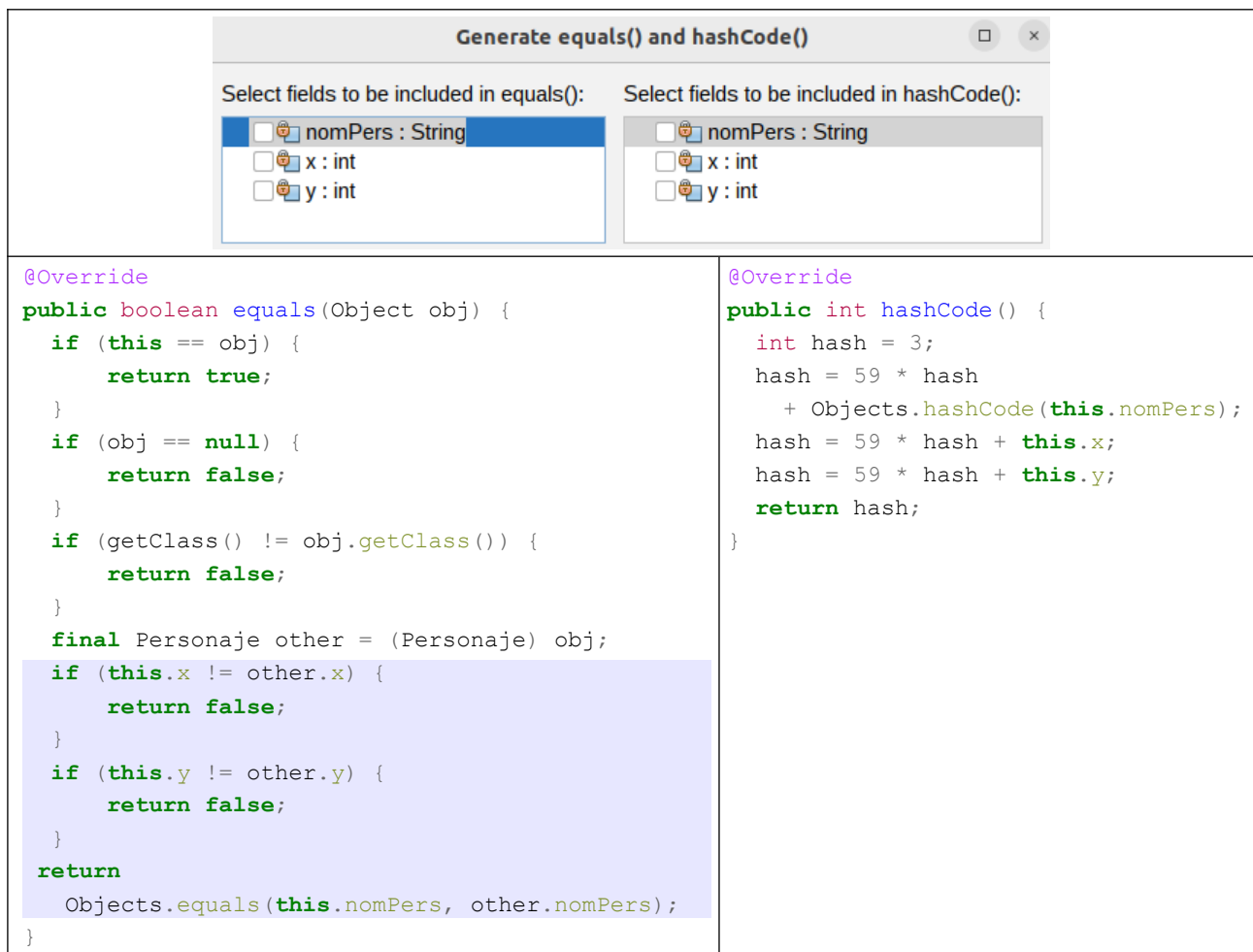
Y una vez que se ha decidido cuándo dos instancias de una clase son iguales, hay que considerar si la implementación del método `equals` para la clase `Object` es válida para la clase.

La implementación del método `equals` de la clase `Object` es la más restrictiva posible. Dos instancias `obj1` y `obj2` de la clase `Object` solo son iguales si ninguna es igual a `null` y además son idénticas, es decir, si se cumple:

```
obj1 != null && obj2 != null && obj1 == obj2.
```

Es decir, la interpretación de `equals` en la clase `Object` es la más restrictiva posible.

Los entornos de desarrollo de NetBeans proporcionan herramientas para ayudar en la definición de este método cuando se crea una nueva clase. Pero este método debería crearse en conjunto con el método `hashCode`. Por ahora no vale la pena preocuparse del método `hashCode`. Baste decir que devuelve un valor de tipo `int`, y que este debe ser igual cuando dos objetos `obj1` y `obj2` son iguales (es decir, cuando `obj1.equals(obj2)`) y distinto en caso contrario.



The screenshot shows the 'Generate equals() and hashCode()' dialog in NetBeans. It has two sections: 'Select fields to be included in equals():' and 'Select fields to be included in hashCode():'. Both sections have checkboxes for 'nomPers : String', 'x : int', and 'y : int'. The 'nomPers : String' checkbox is selected in both. Below the dialog, the generated code is shown in two panels. The left panel shows the `@Override` method `public boolean equals(Object obj)`. The right panel shows the `@Override` method `public int hashCode()`.

```
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final Personaje other = (Personaje) obj;
    if (this.x != other.x) {
        return false;
    }
    if (this.y != other.y) {
        return false;
    }
    return
        Objects.equals(this.nomPers, other.nomPers);
}

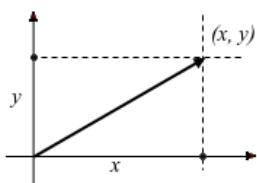
@Override
public int hashCode() {
    int hash = 3;
    hash = 59 * hash
        + Objects.hashCode(this.nomPers);
    hash = 59 * hash + this.x;
    hash = 59 * hash + this.y;
    return hash;
}
```

La primera parte, en general, se puede dejar tal como está. La segunda parte (resaltada) implementa el criterio de que dos instancias de la clase son iguales cuando el valor de todos sus atributos es igual. Si no es un criterio válido para la clase, hay que revisarla para implementar la igualdad de manera apropiada para la clase.

## 9. Creación de clases

En este apartado se muestran varias clases de ejemplo adicionales y se propone como ejercicio la creación de varias más.





El siguiente ejemplo muestra una clase **Vector**. Las instancias de esta clase representan vectores que tienen dos componentes  $x$  e  $y$ .

La suma de dos vectores es otro vector cuyas respectivas componentes  $x$  e  $y$  son la suma de los de ambos vectores.

$$(x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2).$$

En la clase `Vector` están implementados los siguientes métodos que se pueden utilizar para la suma de vectores. Ambos son métodos no estáticos. Por tanto, se ejecutan sobre una instancia de esta clase a la que, en el código fuente, se puede hacer referencia con `this`.

- `Vector suma(Vector v)`. Devuelve la suma del vector sobre el que se aplica y del vector `v`. El vector sobre el que se aplica permanece inalterado.
- `void sumar(Vector v)`. Cambia el vector sobre el que se aplica, sumándole el vector `v`. No devuelve ningún valor.

Aparte de estos métodos, la clase `Vector` tiene los típicos métodos que suele tener cualquier clase: un constructor, métodos *getter* y un método `toString` que devuelve un texto con la representación habitual de un vector: con las coordenadas entre paréntesis y separadas por una coma. El código de todos estos métodos se puede crear con las herramientas que proporciona el entorno de desarrollo. En NetBeans, son las opciones disponibles pulsando el botón derecho del ratón y seleccionando la opción “Insert Code”.

En el método `main` se muestran varios escenarios de uso de los métodos `void sumar(Vector v)` y `Vector suma(Vector v)`.

```
package vectores;

public class Vector {

    private double compX;
    private double compY;

    public Vector(double compX, double compY) {
        this.compX = compX;
        this.compY = compY;
    }

    public double getCompX() {
        return compX;
    }

    public double getCompY() {
        return compY;
    }

    @Override
    public String toString() {
        return "(" + compX + ", " + compY + ")";
    }

    // Ejemplo de suma de vectores:
    // (5, 3) + (4, -2) = (5 + 4, 3 + (-2)) = (9, 1)

    void sumar(Vector v) { // Cambia el estado del objeto, no devuelve ningún valor
        this.compX += v.getCompX();
        this.compY += v.getCompY();
    }
}
```

```

    }

    Vector suma(Vector v) {    // No cambia el estado del objeto, devuelve la suma
        return new Vector(this.compX + v.getCompX(), this.compY + v.getCompY());
    }

    public static void main(String[] args) {

        Vector v1 = new Vector(5, 3);
        Vector v2 = new Vector(4, -2);

        System.out.printf("v1=%s\n", v1);    // Se muestra v1.toString()
        System.out.printf("v2=%s\n", v2);    // Se muestra v2.toString()

        v1.sumar(v2);

        System.out.println("Después de v1.sumar(v2)");

        System.out.printf("v1=%s\n", v1);
        System.out.printf("v2=%s\n", v2);

        Vector suma = v1.suma(v2);

        System.out.printf("Suma de v1 y v2: %s\n", suma);

        System.out.printf("v1=%s\n", v1);
        System.out.printf("v2=%s\n", v2);
    }
}

```

**Actividad 4.8**

Crea la clase anterior y añade el método `equals`. Para esto último puedes utilizar la herramienta que ofrece el entorno de desarrollo para añadir código. Revisa el código que se crea y, en su caso, haz los cambios que sean necesarios.

**Actividad 4.9**

El producto escalar de dos vectores  $(x_1, y_1)$  y  $(x_2, y_2)$  es  $x_1 x_2 + y_1 y_2$ .

El producto de un vector  $(x_1, y_1)$  por un número  $a$  es un vector:  $a(x_1, y_1) = (ax_1, ay_1)$ .

Añade los siguientes métodos a la clase `Vector`:

- `int productoEscalar(Vector v)`. Devuelve el producto escalar del vector por un vector  $v$ .
- `Vector producto(double a)`. Devuelve el producto del vector por  $a$ , y no altera el vector.
- `void multiplicar(double a)`. Multiplica el vector por  $a$ , y no devuelve ningún valor.

La clase **Intervalo** representa un intervalo de números reales entre un valor mínimo y uno máximo. El intervalo  $[a_{\min}, a_{\max}]$  incluye todos los números  $x$  que cumplen que  $a_{\min} \leq x \leq a_{\max}$ , es decir, que  $a_{\min} \leq x$  y  $x \leq a_{\max}$ . Se trata de un intervalo cerrado, porque incluye sus extremos.



```
package intervalos;

public class Intervalo {

    private final double min;
    private final double max;

    public Intervalo(double min, double max) {
        if (min > max) {
            throw new IllegalArgumentException(
                "Extremo superior debe ser mayor o igual que extremo inferior"
            );
        }
        this.min = min;
        this.max = max;
    }

    public double getMax() {
        return max;
    }

    public double getMin() {
        return max;
    }

    boolean numeroEnIntervalo(double num) {
        return num >= min && num <= max;
    }

    boolean seSolapaCon(Intervalo otro) {
        return otro != null && this.max >= otro.min && otro.max >= this.min;
    }

    Intervalo interseccionCon(Intervalo otro) {
        if (otro == null) {
            return null;
        }
        if (this.max < otro.min) {
            return null;
        } else if (otro.max < this.min) {
            return null;
        }
        return new Intervalo(
            Math.max(this.min, otro.min), Math.min(this.max, otro.max)
        );
    }
}
```

Todos los atributos de esta clase son **final**. Su valor no puede cambiar una vez se ha asignado en el constructor. Esta se trata, por tanto, de una clase inmutable.

**Clases inmutables**

Todos los atributos de una clase inmutable se declaran como **final**, por lo que su valor no puede cambiar, una vez que se asigna en el constructor. Las instancias de una clase inmutable son objetos inmutables.

La biblioteca estándar de clases de Java incluye muchas clases inmutables. Por ejemplo, la clase `String` y todas las clases *wrapper* para los tipos básicos: `Integer`, `Double`, etc.

No tiene sentido que el extremo inferior del intervalo sea mayor que el extremo superior. Y no se puede evitar que, cuando se crea un objeto, se pasen valores para sus parámetros que violen esta restricción. En ese caso, el constructor lanza una excepción de tipo `IllegalArgumentException`.

**Lanzamiento de excepción `IllegalArgumentException` en el constructor**

Nunca debería crearse un objeto con datos incorrectos o inconsistentes. En este caso particular, no debería crearse un intervalo cuyo extremo inferior sea mayor que su extremo superior.

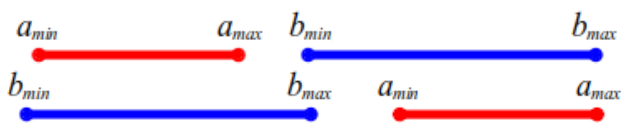
Para ello, se puede lanzar una excepción de tipo `IllegalArgumentException`. Esto impide la creación del objeto.

Los métodos `seSolapaCon` e `interseccionCon` tienen un parámetro de tipo `Intervalo`. Antes de acceder a cualquier atributo o método suyo, hay que comprobar que es distinto de **null**. No tiene sentido acceder a un atributo o método de un objeto con valor **null**, porque esta es una referencia o puntero que no hace referencia a ningún objeto. Si se hace, se produce una excepción de tipo `NullPointerException`. Cabe reseñar aquí que **this** nunca puede tener valor **null**, por el hecho mismo de ser una referencia a un objeto sobre el que se está ejecutando un método.

A continuación se explica la implementación de los métodos `seSolapaCon` e `interseccionCon`.

**Intersección de dos intervalos**

Es más fácil obtener la condición para cuando dos intervalos NO se solapan, porque solo están los dos casos que se muestran a la derecha. En el primer caso, se tiene que  $a_{\max} < b_{\min}$ , y en el segundo se tiene que  $b_{\max} < a_{\min}$ .



Por tanto, dos intervalos  $[a_{\min}, a_{\max}]$  y  $[b_{\min}, b_{\max}]$  no se solapan cuando:  $a_{\max} < b_{\min}$  o  $b_{\max} < a_{\min}$ .

Es decir, que se solapan en caso contrario, cuando: no  $(a_{\max} < b_{\min}$  o  $b_{\max} < a_{\min})$

Lo que equivale a no  $(a_{\max} < b_{\min})$  y no  $(b_{\max} < a_{\min})$ .

Es decir, dos intervalos se solapan cuando  $a_{\max} \geq b_{\min}$  y  $b_{\max} \geq a_{\min}$

Y en ese caso, el extremo inferior de la intersección debe ser uno de los dos extremos inferiores  $a_{\min}$  o  $b_{\min}$ , y es el máximo de ellos, es decir,  $\max(a_{\min}, b_{\min})$ . Y el extremo superior, por un razonamiento análogo, es  $\min(b_{\max}, a_{\max})$ .

Es decir, la intersección de dos intervalos  $[a_{\min}, a_{\max}]$  y  $[b_{\min}, b_{\max}]$  que se solapan es  $[\max(a_{\min}, b_{\min}), \min(b_{\max}, a_{\max})]$

Por lo demás, en esta clase se utiliza el valor **null** para representar un intervalo que no contiene ningún punto, es decir, el conjunto vacío  $\emptyset$ . El método `interseccionCon` devuelve **null** si el intervalo que se le pasa como argumento para el parámetro `otro` no se solapa con el propio intervalo (**this**), porque en ese caso la intersección es el conjunto vacío. La intersección con **null** es **null**, porque la intersección de cualquier intervalo con el conjunto vacío  $\emptyset$  es el conjunto vacío  $\emptyset$ .

Es de reseñar que, como ya se explicó anteriormente, se puede acceder a los atributos privados (**private**) de otro objeto de la misma clase (`otro.min` y `otro.max`). Se puede acceder, en general, a los de objetos de clases del mismo paquete (**package**).



En C++ nunca se puede acceder a atributos privados (**private**) si no es desde el propio objeto. No se puede desde otros objetos, ni aunque pertenezcan a la misma clase. Y además solo se puede hacer desde métodos de la propia clase y no desde métodos de ninguna subclase suya. A los atributos protegidos (**protected**) solo se puede acceder desde el propio objeto, bien sea desde métodos de la propia clase o desde métodos de una subclase suya. En C++, por defecto, si no se especifica la visibilidad de un atributo o clase, es de tipo **private**. C++ es, por tanto, más restrictivo que Java en lo que respecta a visibilidad de atributos y métodos de los objetos.

**Actividad 4.10**

Redefine de manera apropiada los métodos `toString` y `equals` para la clase `Intervalo`.

**Actividad 4.11**

Modifica la clase `Personaje` de manera que lance una excepción de tipo `IllegalArgumentException` si se le pasan argumentos inválidos.

**Actividad 4.12**

Crear una clase `FichaDomino` que viene dada por dos números de 0 a 6, que pueden ser iguales. El creador de la clase toma estos dos números como parámetros. Se pueden dar en cualquier orden. Es decir, `FichaDomino(3, 4)` y `FichaDomino(4, 3)` son la misma ficha, y el constructor admite que se dé primero el menor o el mayor de los números.

En la clase se deben redefinir de forma apropiada los métodos `toString` y `equals` de `Object`. El método `toString` debe dibujar la pieza (3, 4) de la siguiente manera: [3|4].

La clase debe tener un método `String combina(FichaDomino f)` que devuelva un `String` que será `null` si ambas fichas no se pueden combinar porque no tienen ningún número en común. En caso contrario, se mostrará la forma en la que ambas pueden combinar, pero siempre debe aparecer primero la ficha sobre la que se ejecuta el método. Una ficha no se puede combinar consigo misma, porque en el dominó no hay ninguna pieza duplicada, todas son distintas.

Por ejemplo, tras crear varias fichas de la siguiente manera:

```
FichaDomino f1 = new FichaDomino(3,4);
FichaDomino f2 = new FichaDomino(3,5);
FichaDomino f3 = new FichaDomino(4,6);
```

```
f1.combina(f2) devuelve [4|3][3|5]
f1.combina(f3) devuelve [3|4][4|6]
f2.combina(f3) devuelve null
f2.combina(f1) devuelve [5|3][3|4]
f3.combina(f1) devuelve [6|4][4|3]
f3.combina(f2) devuelve null
```

**Actividad 4.13**

Crear una clase `Cuenta` que tiene como atributos un número de cuenta (un `String`), un saldo y una moneda, que por defecto es €. Esto significa que, si no se indica la moneda cuando se crea una instancia de la clase, es €. Pero se puede especificar otra moneda. Para crear una cuenta hay que indicar siempre el saldo inicial. Hay que crear un constructor para cada forma de crear una cuenta.

Se puede hacer un ingreso en una cuenta. Se puede retirar dinero de una cuenta, siempre que el saldo de la cuenta no sea menor que la cantidad que se quiera retirar. Se puede hacer una transferencia de una cantidad determinada de una cuenta a otra, pero en principio solo si ambas tienen la misma moneda. Deben crearse métodos no estáticos para realizar cada una de estas operaciones. Deben existir métodos *getter* para todos los atributos de la cuenta.

**Actividad 4.14**

Crear una clase `Tiempo` cuyas instancias representan un intervalo de tiempo en horas (un número entero), minutos (un número entero) y segundos (un número con decimales). El tiempo debe estar normalizado. Es decir, que no pueden haber más de 60 minutos (60 minutos serían una hora) ni más de 60 segundos (60 segundos serían un minuto). El constructor de la clase puede admitir un intervalo de tiempo no normalizado, pero debe normalizarlo para asignar valores correctos a sus atributos.

Debe tener un método para sumar un intervalo de tiempo a otro. En realidad deben ser dos: uno que sume el intervalo al objeto sobre el que se ejecuta, y otro que devuelva la suma del intervalo sobre el que se ejecuta y del otro intervalo.

Cuando se suma un intervalo de tiempo a otro con el primero de los dos métodos anteriores, se debe normalizar el resultado. Dado que puede ser necesario normalizar el intervalo de tiempo en distintas situaciones, es conveniente crear un método para ello. Este método debería ser privado (`private`), porque solo será necesario utilizarlo en métodos de la propia clase.

Esta clase debe tener un método para obtener la duración del intervalo en segundos.

Debe tener también un método `compara`, al que se le pase otro intervalo de tiempo, y que devuelva -1, 0 o 1 dependiendo de si el otro intervalo es menor, igual o mayor.

**Actividad 4.15**

Crea dos clases `Punto` y `Rectangulo`, ambas inmutables. Las instancias de la clase `Punto` representan puntos con coordenadas `x` y `y`, ambas de tipo `double`. Las instancias de la clase `Rectangulo` representan un rectángulo, definido por dos puntos `P1` (esquina superior izquierda) y `P2` (esquina inferior derecha).

En el eje de las `x` aumentan los valores hacia la derecha, y en el eje de las `y` aumentan los valores hacia abajo. Por tanto, el valor de la coordenada `x` de `P2` debe ser mayor o igual que la correspondiente coordenada `x` de `P1`, y lo mismo para la coordenada `y`.

Usa la clase `Punto` en la clase `Rectangulo`.

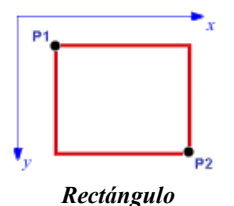
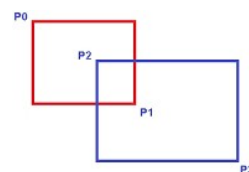
Implementa métodos `base`, `altura`, `superficie` y `perimetro`, que devuelven todos un `double`, en la clase `Rectangulo`.

Redefine el método `toString` para ambas clases. Para un `Punto` debe devolver "`(x, y)`", donde `x` e `y` son las coordenadas del punto. Para un `Rectangulo` debe devolver "`rect(x, y, ancho, alto)`", donde `(x, y)` es la esquina superior izquierda y `ancho` y `alto` son la anchura y la altura del rectángulo, respectivamente.

Redefine el método `equals` para ambas clases.

Implementa un método boolean `enRectangulo(Punto p)`, que devuelva `true` si el punto está dentro del rectángulo y `false` en caso contrario.

Implementa un método `Rectangulo interseccionCon(Rectangulo r)`, que devuelva la intersección del rectángulo con el otro dado, o `null` si no se solapan. Puedes utilizar la clase `Intervalo` y, en particular, su método `interseccionCon`.

**Rectángulo****Intersección de rectángulos****Actividad 4.16**

Crear una clase `Barquito` que representa un barco del juego de los barquitos.

Los barcos se sitúan en un tablero de 10 filas x 10 columnas. Cada posición viene especificada por sus coordenadas, que son una fila (de A a la J) y una columna (de 1 a 10).

Un barco viene definido por las coordenadas (fila y columna) de su posición inicial y de su posición final. Internamente ambas son números de 0 a 9, almacenados en un atributo de tipo `int`.

El número de filas y columnas deben estar en constantes de clase `NUM_FILAS`,

	1	2	3	4	5	6	7	8	9	10
A										
B										
C										
D										
E										
F										
G										
H										
I										
J										

NUM\_COLUMNS.

La longitud de un barquito está entre 1 y 5.

Al constructor se le debe pasar la especificación de un barco, que es distinta dependiendo de si es horizontal o vertical. Para un barco horizontal podría ser C6:8, y para uno vertical podría ser F:G6. Deben utilizarse expresiones regulares para validar el formato y obtener la información relevante, es decir, fila y columna inicial y final. Una vez extraída esta información, se debe verificar, además, que la fila final no es anterior a la fila inicial (pueden ser, por ejemplo, C y F pero no F y C) y que la columna final no es anterior a la columna inicial (pueden ser, por ejemplo, 3 y 4 pero no 4 y 3).

Crea antes una clase auxiliar `Coordenada` que represente las coordenadas de una posición. Ambas deben ser un número entre 0 y 9. Su creador debe admitir un `String`. Por ejemplo: "G8". Se deben utilizar expresiones regulares para validar el `String` y obtener la información relevante, y posteriormente se debe validar.

Añadir a la clase `Barquito` métodos para:

- Determinar si una posición (`Coordenada`) pertenece al barquito.
- Determinar si una posición (`Coordenada`) es contigua al barquito.
- Determinar si un barco toca a otro (esto no está permitido). Quizá puedas basarte en código de la clase `Intervalo`.

#### Actividad 4.17

Cambia la clase `Intervalo` para que pueda ser abierto, opcionalmente, por cada uno de los dos extremos. De esta forma, podría representar no solo intervalos de tipo  $[a, b]$ , sino también de tipo  $(a, b)$ ,  $[a, b)$  y  $(a, b]$ . La manera más sencilla es, seguramente, añadir dos atributos booleanos, uno para el extremo inferior y otro para el superior, que indican si el intervalo es abierto o cerrado por el extremo. Revisa todos los métodos y haz todos los cambios que sean necesarios.

#### Actividad 4.18

Cambia la clase `Intervalo` para que pueda representar intervalos infinitos. Es decir, intervalos cuyo extremo inferior es  $-\infty$  o cuyo extremo superior es  $+\infty$ . Incluso el intervalo  $(-\infty, +\infty)$  que incluye todos los números. La clase `Double` tiene dos constantes de clase que representan  $-\infty$  y  $+\infty$ . Prueba bien todos los métodos para que devuelvan resultados correctos con intervalos que tengan un extremo infinito (o los dos).

## 10. Tipos enumerados

Un tipo enumerado es un tipo de datos especial que admite un conjunto cerrado de posibles valores predefinidos y constantes. Por ejemplo:

- Las direcciones IZQUIERDA, DERECHA, ARRIBA y ABAJO.
- Los días de la semana LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO y DOMINGO.
- Los reinos de los seres vivos eucariotas: ANIMALES, VEGETALES, HONGOS, PROTOZOOS, ALGAS.

### 10.1. Tipos enumerados básicos

@PEND. Revisar, se han pasado los tipos enumerados desde otro lugar, hacia este tema de programación orientada a objetos. Repasar encaje en este contexto. Pasar a tema 4, porque aquí se explica cómo crear tipos enumerados a medida. Aquí se podría poner algún ejemplo de cómo utilizar algún tipo enumerado de la biblioteca de clases de Java (habría que buscar uno útil, apropiado e ilustrativo). Es importante, por ejemplo, el hecho de que, normalmente, habría que definirlo como public si se quiere que se pueda utilizar externamente. Por ejemplo, tipo enumerado `Direccion` en clase

Personaje en el tema siguiente, para que lo puedan utilizar los clientes de la clase y pasar valores de este tipo a métodos de esta clase. El lugar apropiado es una vez se hayan explicado las clases, y entonces se puede explicar el constructor, por ejemplo, y explicar que son un tipo de clase especial con un conjunto fijo y cerrado de posibles instancias.

Un tipo enumerado se define con la palabra reservada `enum`. Un tipo enumerado para las direcciones se podría definir de la siguiente forma.

```
enum Direccion {
    IZQUIERDA, DERECHA, ARRIBA, ABAJO
}
```

## 10.2. Uso de tipos enumerados básicos

Se puede declarar una variable de este tipo y asignarle un valor de la siguiente forma:

```
Direccion dir = Direccion.IZQUIERDA;
```

Los tipos enumerados se prestan mucho a su uso con sentencias `switch` en las que se hace una cosa distinta para cada uno de sus posibles valores. Para ellas hay una sintaxis más tradicional (que de hecho es la misma que existe en los lenguajes de programación C y C++), y una nueva (*rule switch*), que se introdujo a partir de una determinada versión de Java. Se muestra un ejemplo a continuación, expresado con ambas variantes.

Sintaxis clásica	Sintaxis <i>rule switch</i>
<pre>Direccion dir = Direccion.ABAJO; int x = 0; int y = 0;</pre>	
<pre>switch (dir) {     case IZQUIERDA:         x--;         break;     case DERECHA:         x++;         break;     case ARRIBA:         y--;         break;     case ABAJO:         y++;         break; }</pre>	<pre>switch (dir) {     case IZQUIERDA -&gt;         x--;     case DERECHA -&gt;         x++;     case ARRIBA -&gt;         y--;     case ABAJO -&gt;         y++; }</pre>
<pre>System.out.printf("Fin: (%d,%d)\n", x, y);</pre>	

### Actividad 4.19

Crea un programa con un *array* con tipo base `Direccion`, que contenga varias direcciones (elementos de tipo `Direccion`) que representen sucesivos desplazamientos desde una posición inicial. Por ejemplo:

```
Direccion[] desplazamientos = {
    Direccion.ARRIBA,
    Direccion.DERECHA,
    Direccion.DERECHA,
    Direccion.ABAJO,
```



```
Direccion.IZQUIERDA
```

```
}
```

Define dos variables `x` e `y` con un valor inicial cualquiera. Utiliza un bucle `for` para obtener una a uno a uno los desplazamientos contenidos en el *array*, y dentro de él una sentencia `switch` como la anterior para calcular la nueva posición, a partir de la anterior, después de cada desplazamiento. Haz que el programa escriba al principio la posición inicial y al final la posición final.

Pueden utilizarse cláusulas `case` con más de un valor, y cláusulas `default`, como se muestra en los siguientes ejemplos.

El primero es con el tipo `Direccion` ya visto.

Sintaxis clásica	Sintaxis <i>rule switch</i>
<pre>Direccion dir = Direccion.IZQUIERDA;  switch (dir) {     case IZQUIERDA:     case DERECHA:         System.out.println("Horizontal.");         break;     case ARRIBA:     case ABAJO:         System.out.println("Vertical.");         break; }</pre>	<pre>switch (dir) {     case IZQUIERDA, DERECHA -&gt;         System.out.println("Horizontal.");     case ARRIBA, ABAJO -&gt;         System.out.println("Vertical."); }</pre>

El siguiente es con un nuevo tipo `DiaSemana` para los días de la semana.

Sintaxis clásica	Sintaxis <i>rule switch</i>
<pre>package dias;  public class Dias {      enum DiaSemana {         LUNES, MARTES, MIERCOLES,         JUEVES, VIERNES, SABADO, DOMINGO     }      public static void main(String[] args) {          DiaSemana dia = DiaSemana.VIERNES;          switch (dia) {             case SABADO:             case DOMINGO:                 System.out.println("Festivo");                 break;             default:                 System.out.println("Laborable");         }     } }</pre>	<pre>switch (dia) {     case SABADO, DOMINGO -&gt;         System.out.println("Festivo");     default -&gt;         System.out.println("Laborable"); }</pre>

```
}  
}
```

### 10.3. Iteración sobre todos los valores de un tipo enumerado

El método `values()` devuelve un *array* con todos los posibles valores del tipo enumerado (`Direccion`). El siguiente programa itera sobre este *array* para mostrar todos los valores. Para cada uno, el método `name()` devuelve una descripción textual, y el método `ordinal()` devuelve un número de orden.

```
enum Direccion {  
    IZQUIERDA, DERECHA, ARRIBA, ABAJO  
}  
  
for (Direccion dir : Direccion.values()) {  
    System.out.printf("%d: %s\n", dir.ordinal(), dir.name());  
}
```

La salida del programa anterior es la siguiente.

```
0: IZQUIERDA  
1: DERECHA  
2: ARRIBA  
3: ABAJO
```

### 10.4. Tipos enumerados avanzados

Los tipos enumerados sencillos vistos hasta ahora son equivalentes a los existentes en lenguajes anteriormente existentes como C y C++. Pero en Java tienen funcionalidades adicionales.

Para empezar, cada posible valor del tipo enumerado puede tener atributos adicionales. Por ejemplo, se puede definir el siguiente tipo enumerado para los planetas del sistema solar. Para cada planeta se tiene la masa y el radio, que se asignan al crear cada objeto de tipo planeta. El programa escribe la masa de todos los planetas.

```
package planetas;  
  
enum Planeta {  
    MERCURIO(3.303e+23, 2.4397e6),  
    VENUS(4.869e+24, 6.0518e6),  
    TIERRA(5.976e+24, 6.37814e6),  
    MARTE(6.421e+23, 3.3972e6),  
    JUPITER(1.9e+27, 7.1492e7),  
    SATURNO(5.688e+26, 6.0268e7),  
    URANO(8.686e+25, 2.5559e7),  
    NEPTUNO(1.024e+26, 2.4746e7);  
  
    public final double masa;    // en Kg  
    public final double radio;  // en m  
  
    Planeta(double masa, double radio) {
```

```

        this.masa = masa;
        this.radio = radio;
    }
}

public class Planetas {
    public static void main(String[] args) {
        for (Planeta p : Planeta.values()) {
            System.out.printf("%s, masa=%f\n", p.name(), p.masa);
        }
    }
}

```

**Actividad 4.20**

Añade el símbolo de cada planeta, además de la masa y el radio. Haz que en el método `main` se escriba el nombre y el símbolo, la masa y el radio de cada planeta.

Los símbolos de cada planeta se pueden encontrar en [https://es.wikipedia.org/wiki/S%C3%Admbolos\\_planetarios](https://es.wikipedia.org/wiki/S%C3%Admbolos_planetarios). El de Mercurio, por ejemplo, está en este fragmento de texto. Los puedes copiar y pegar.

La salida del programa debe ser esta:

```

MERCURIO(☿), masa: 3303000000000000000000,000000, radio: 2439700,000000
VENUS(♀), masa: 4869000000000000000000,000000, radio: 6051800,000000
TIERRA(□), masa: 5976000000000000000000,000000, radio: 6378140,000000
MARTE(♂), masa: 6421000000000000000000,000000, radio: 3397200,000000
JUPITER(♃), masa: 1900000000000000000000,000000, radio: 71492000,000000
SATURNO(♄), masa: 5688000000000000000000,000000, radio: 60268000,000000
URANO(♅), masa: 8686000000000000000000,000000, radio: 25559000,000000
NEPTUNO(♆), masa: 1024000000000000000000,000000, radio: 24746000,000000

```

**Mercurio** [editar]

El símbolo ☿ de Mercurio es un **caduceo** (un bastón entrelazado con de la antigüedad.<sup>9</sup> Algún tiempo después del siglo XI, se añadió un más cristiano.<sup>4</sup> Su punto de código Unicode es U+263F ☿ MERCURY.

Para el tipo enumerado `Direccion` anterior se podrían añadir atributos para el desplazamiento horizontal y vertical. De esta manera, para calcular la nueva posición a partir de una posición inicial y una dirección de desplazamiento, se podrían sumar a las coordenadas `x` y `y` de la posición actual. En la siguiente actividad se propone hacer esto, solo que cambiando el nombre por `PuntoCardinal`. Las direcciones arriba, derecha, abajo e izquierda se corresponderían con los puntos cardinales norte, este, sur y oeste.

**Actividad 4.21**

Crea un tipo enumerado `PuntoCardinal` con los cuatro puntos cardinales básicos: norte, este, sur, oeste, y además los cuatro intermedios: noreste, sureste, suroeste, noroeste.

A cada uno hay que asociarle un desplazamiento horizontal `deltaX` y uno vertical `deltaY`. Por ejemplo, para norte hay que decrementar `y` (es decir, `deltaY=-1`), y para este hay que incrementar `x` (es decir, `deltaX=1`). Para noreste hay que combinar los desplazamientos de norte y de este (es decir, `deltaY=-1` y `deltaX=1`).

Crea un *array* con varios desplazamientos según los puntos cardinales, y un programa que, dados esos desplazamientos, finalmente escriba la posición final, si la inicial es (0, 0). El programa debe contener al menos una vez cada punto cardinal.

**Actividad 4.22**

Crea un programa a partir del desarrollado para la *Actividad 4.21* que anote las posiciones que se van recorriendo en un *array* bidimensional de 10x10 posiciones `tablero boolean[10][10]`. En este *array* deben anotarse las posiciones por las que se pasa, empezando por la posición inicial. Para terminar, el programa debe mostrar el tablero, marcando con caracteres distintos las posiciones por las que se ha pasado y las posiciones por las que no se ha pasado. Si no se ha hecho la actividad anterior, se puede crear a partir del desarrollado para la *Actividad 4.19*.

**Actividad 4.23**

Crea un programa a partir del desarrollado para la *Actividad 4.22* que anote en cada posición el desplazamiento a partir de esa posición, con un carácter distinto. El tipo base para el *array* debe cambiar a `char`. Los distintos desplazamientos se deben especificar con flechas. Puedes buscar los caracteres para las distintas flechas en <https://www.fileformat.info/info/unicode/char/search.htm>, especificando como texto para la búsqueda *arrow* (flecha, en inglés).

## 11. Herencia

Una clase se declara como clase hija incluyendo la opción `extends` en su declaración. En una clase hija están disponibles todos los atributos y métodos de la clase madre. Pero además en ella se puede:

- Añadir nuevos atributos y métodos.
- Redefinir (*override*) métodos de la clase madre.

A continuación se muestra el código de una clase `PersonajeConEnergia`, que es hija de `Personaje`. Esta clase tiene algunas particularidades con respecto a la clase `Personaje`. Tiene una cantidad de energía acumulada y gasta una cantidad de ella con cada desplazamiento.

Los métodos de la clase madre siguen estando disponibles. Para utilizarlos, se accede a ellos mediante `super`.

```
package com.tusjuegos.juego;

public class PersonajeConEnergia extends Personaje {

    int energia;
    int consumoPorPaso;

    PersonajeConEnergia(String nombre, int x, int y, int energiaInic, int consumoPorPaso) {
        super(nombre, x, y);
        if(energiaInic < 0) {
            throw new IllegalArgumentException("Valor inicial de energía incorrecto");
        }
        if(consumoPorPaso < 0) {
            throw new IllegalArgumentException("Consumo por paso incorrecto");
        }
        this.energia = energiaInic;
        this.consumoPorPaso = consumoPorPaso;
    }

    @Override
    public void avanza(Direccion mov) {
        if (energia > consumoPorPaso) {
            int xPrev = this.getX();
```

```
        int yPrev = this.getY();
        super.avanza(mov);
        if (this.getX() != xPrev || this.getY() != yPrev) {
            energia -= consumoPorPaso;
        }
    }

    public int reponEnergia(int carga) {
        this.energia += carga;
        return energia;
    }

    public static void main(String[] args) {
        PersonajeConEnergia pce = new PersonajeConEnergia("superhormiga", 5, 1, 100, 2);
        System.out.print(pce);
    }
}
```

En el constructor de esta nueva clase, se llama primero al constructor de la clase madre, `Personaje`. Después se asigna valor a los atributos que añade la clase `PersonajeConEnergia`.

El método `avanza` se redefine (eso significa la anotación `@Override`). Sus parámetros son los mismos. Desde el método `avanza` de `PersonajeConEnergia` se ejecuta el de la clase madre `PersonajeConEnergia` con `super.avanza(mov)`. Si la nueva posición es distinta a la antigua, eso significa que se ha realizado el movimiento y se consume la energía de un paso (`consumoPorPaso`).

Aparte, se añade un método `reponEnergia` para reponer una cantidad de energía.

La siguiente clase, `PersonajeMovLim`, es un tipo de personaje que solo se puede mover en las direcciones que se le pasan en un *array* en el constructor.

```
package com.tusjuegos.juego;
import com.tusjuegos.juego.Personaje.Direccion;
import java.util.Arrays;

public class PersonajeMovLim extends Personaje {

    Personaje.Direccion[] movPermitidos;

    PersonajeMovLim(String nombre, int x, int y, Direccion[] movPermitidos) {
        super(nombre, x, y);
        this.movPermitidos = movPermitidos;
    }

    @Override
    public void avanza(Personaje.Direccion mov) {
        for(Direccion movPerm: this.movPermitidos) {
            if(mov.equals(movPerm)) {
                super.avanza(mov);
                break;
            }
        }
    }
}
```

```
public static void main(String[] args) {
    Direccion[] movPerm = {Direccion.DERECHA, Direccion.IZQUIERDA};
    PersonajeMovLim pml = new PersonajeMovLim("superhormiga", 5, 1, movPerm);
    System.out.print(pml);
}
}
```

#### Actividad 4.24

Crea las clases `PersonajeConEnergia` y `PersonajeMovLim` como clases hijas de tu clase `Personaje`. Seguramente has hecho algunos de los ejercicios propuestos para la clase `Personaje`. Posiblemente sea necesario o conveniente hacer algunas adaptaciones o mejoras. Por ejemplo: si has cambiado el método `avanza` para que devuelva `true` o `false` según se haya podido realizar el movimiento, puedes simplificar el método `avanza`. En cuanto al método `avanzaExt`, tendrás que decidir qué hacer cuando no hay energía suficiente para realizar la secuencia de movimientos. Podrías hacer que el personaje se quede donde está. O podrías añadir un parámetro de tipo `boolean` al método que indique si, en este caso, el personaje debe realizar todos los movimientos que pueda hasta quedarse sin energía (con valor `true`) o realizar todos los movimientos que pueda (con valor `false`). Si haces esto último, podrías crear un método `avanzaExt` sin este parámetro que haga lo mismo que si el parámetro tiene valor `false`.

#### Actividad 4.25

Añade métodos *getter* para todos los atributos de las clases anteriores. No hace falta que lo hagas manualmente, puedes utilizar las herramientas que proporciona el entorno de desarrollo para ello.

#### Actividad 4.26

Para acceder al valor del atributo `x` en el método `avanza` de `PersonajeConEnergia`, hace falta utilizar el método `getX`. ¿Por qué no se puede, sencillamente, acceder con `this.x` o con `x`? ¿Qué cambios se podrían hacer en la clase `Personaje` para que sí se pudiera? Haz cambios y pruébalos, pero después deja la clase `Personaje` como estaba.

#### Actividad 4.27

¿Qué pasa si en el constructor de la clase `PersonajeMovLimitados` se pasa el valor `null` en el parámetro `movPermitidos` del constructor. Cambia la clase para que, si se pasa valor `null`, se pueda realizar cualquier movimiento.

#### Actividad 4.28

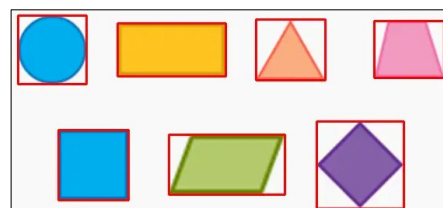
Crea una clase `PersonajeQueRebota` como subclase de la clase `PersonajeConEnergia`. Este personaje rebota con los bordes. Si cuando está en el borde izquierdo intenta realizar un movimiento hacia la izquierda, en lugar de quedarse donde está, rebota y realiza un movimiento en dirección contraria, es decir, hacia la derecha. Debes redefinir el método `avanza` y crear en el método `main` un programa de prueba que verifique el rebote con cada uno de los cuatro bordes del tablero. Mejor aún, puedes crear una clase de prueba con JUnit con un método de prueba para verificar cada uno de estos casos.

## 11.1. Clases abstractas

Una clase abstracta es una clase que especifica uno o varios métodos que no implementa la propia clase, sino que deben implementar sus subclases. No se puede crear ninguna instancia de una clase abstracta.

Podría ser el caso de una clase `Figura`, que representa una figura geométrica. Sobre cualquier instancia de esta clase deberían poderse ejecutar los siguientes métodos:

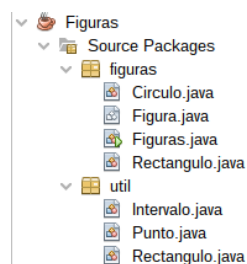
- Un método `double perimetro()` que devuelve el perímetro de la figura.
- Un método `double superficie()` que devuelve el área o superficie de la figura.
- Un método `void dibujar()` que dibuja la figura en la pantalla.
- Un método `Rectangulo cuadroDelimitador()` que devuelve un `Rectangulo` que delimita la `Figura`. En inglés se suele denominar *bounding box*.



Cuadro delimitador de diferentes figuras

Como subclases de `Figura` se podrían tener `Rectangulo`, `Circulo`, `Cuadrado` y `Triangulo`.

La clase `Rectangulo` hija de `Figura` es distinta de la clase `Rectangulo` que se ha propuesto como ejercicio en una sección previa. La segunda podría estar disponible previamente, y se podría querer utilizar tal como está, sin ningún cambio. La segunda podría ser una nueva que se quiera desarrollar para una aplicación de diseño gráfico. Que ambas tengan el mismo nombre no supone ningún problema, siempre que estén en distintos paquetes. En la ilustración se muestra la jerarquía de paquetes que podría haber para esta aplicación. En un paquete `util` aparte, están las clases auxiliares `Punto`, `Intervalo` y `Rectangulo`, que utilizan la clase `Figura` y sus clases hijas.



Desde cualquier clase del paquete `figuras` se puede hacer referencia a la clase `Rectangulo` dentro del paquete `util` con `util.Rectangulo`. Y se puede hacer referencia a la clase `Rectangulo` dentro del mismo paquete `figuras` con `Rectangulo`.

Los métodos `perimetro`, `superficie`, `dibujar` y `cuadroDelimitador` se pueden definir perfectamente en las subclases, pero ninguno en la superclase `Figura`. Porque no hay ninguna forma general de calcular el área o la superficie o de dibujar una figura cualquiera, se hace de forma completamente diferente dependiendo del tipo de figura.

Podría pensarse que `Figura` es una especie de clase vacía que solo especifica los métodos que deben tener sus subclases. Pero no tiene por qué ser así. La clase `Figura` podría tener atributos y métodos comunes a cualquier tipo de figura:

- Atributos. La clase `figura` podría tener un atributo `pos` de la clase `Punto`, para su posición. Este punto tendría un significado distinto para cada subclase de `Figura`. Para un `Circulo`, seguramente, en pero determina la `x`, `y` para las coordenadas de un punto distinguido. O mejor, un atributo `posicion` de tipo `Punto`. Para un círculo podría ser su centro. Pero este punto distinguido no tiene por qué ser el centro geométrico de la figura. Basta con que sirva para situarlo en el plano. Para un `Rectangulo` podría ser, por ejemplo, el extremo superior izquierdo.
- Métodos. Por ejemplo, un método booleano `posibleInterseccion(Figura otraFig)` que devuelve `true` si existe una posible intersección entre las dos figuras. Más exactamente, devolvería `true` cuando sus cuadros delimitadores, obtenidos con el método `cuadroDelimitador`, se solapan. Esto no implica siempre que las figuras se solapan. Pero si este método devuelve `false`, sí se puede descartar que se solapan. Lo interesante es que, aunque `cuadroDelimitador` está definido en las subclases, se puede utilizar en un método de la superclase `Figura`.

Un método abstracto se declara con la palabra reservada **abstract**.

Si una clase tiene algún método abstracto, es una clase abstracta.

Una clase abstracta debe declararse con la palabra reservada **abstract**.

No se pueden crear instancias de una clase abstracta, porque una clase abstracta tiene métodos que no están definidos. En todo caso, se podrán crear objetos de subclases cuyas cuyos métodos están todos definidos.

La clase Figura puede implementarse de la siguiente manera.

```
package figuras;

import util.Punto;

public abstract class Figura {

    private Punto pos;

    public Figura(Punto pos) {
        this.pos = pos;
    }

    public Punto getPos() {
        return pos;
    }

    abstract public double perimetro();
    abstract public double superficie();
    abstract public void dibujar();
    abstract public util.Rectangulo cuadroDelimitador();

    boolean posibleInterseccion(Figura otraFig) {
        util.Rectangulo r = this.cuadroDelimitador();
        util.Rectangulo rOtro = otraFig.cuadroDelimitador();
        return r.seSolapaCon(rOtro);
    }

}
```

Los métodos declarados como **abstract** en la clase Figura se pueden definir en sus subclases.

Las clases hijas podrían crearse de la siguiente manera.

```
package figuras;

import util.Punto;

public class Circulo extends Figura {

    private double radio;

    public Circulo(Punto pos, double radio) {
        super(pos);
        this.radio = radio;
    }

}
```



```
}

public double getRadio() {
    return radio;
}

@Override
public String toString() {
    return "Circulo en ("
        + getPos().getX() + ", " + getPos().getY()
        + ") con radio " + radio;
}

@Override
public double perimetro() {
    return 2 * Math.PI * radio;
}

@Override
public double superficie() {
    return Math.PI * radio * radio;
}

@Override
public void dibujar() {
    System.out.printf("Dibujo de un círculo: %s", this);
}

@Override
public util.Rectangulo cuadroDelimitador() {
    return new util.Rectangulo(
        new Punto(getPos().getX() - radio, getPos().getY() - radio),
        new Punto(getPos().getX() + radio, getPos().getY() + radio)
    );
}
}
```

```
package figuras;

import util.Punto;

public class Rectangulo extends Figura {

    private double base;
    private double altura;
```

```
public Rectangulo(Punto pos, double base, double altura) {
    super(pos);
    this.base = base;
    this.altura = altura;
}

public double getBase() {
    return base;
}

public double getAltura() {
    return altura;
}

@Override
public double perimetro() {
    return 2 * (base + altura);
}

@Override
public double superficie() {
    return base * altura;
}

@Override
public void dibujar() {
    System.out.printf("Dibujo de un rectángulo: %s", this);
}

@Override
public util.Rectangulo cuadroDelimitador() {
    return new util.Rectangulo(
        new Punto(getPos().getX(), getPos().getX() + base),
        new Punto(getPos().getY(), getPos().getY() + altura)
    );
}
}
```

#### Actividad 4.29

Crea las clases *Figura*, *Rectangulo* y *Circulo* con el código que se ha proporcionado. Escribe un programa que cree un *array* *Figura[] figuras* con cuatro objetos de subclases de *Figura*, y que utilice un bucle *for(Figura f: figuras)* para ejecutar los métodos *dibujar*, *cuadroDelimitador*, *perimetro* y *superficie* sobre todos ellos.

**Actividad 4.30**

¿Podrías crear la clase `Cuadrado` como subclase de `Rectangulo`? Inténtalo.

## 11.2. Atributos y métodos protegidos (**protected**)

Un método o atributo **protected** está disponible solo desde clases hijas. Es un nivel de protección intermedio entre **private** (solo está disponible para la propia clase) y **public** (esta disponible para cualquier clase).

**Actividad 4.31**

Define el atributo `pos` de la clase `Figura` como `protected`, y cambia el código de sus clases hijas para que accedan directamente a este atributo, sin necesidad de utilizar el método `getPos()`.

## 12. Interfaces

Para ilustrar el funcionamiento y la utilidad de las interfaces, se plantea el siguiente problema de programación.

Se quieren crear clases para representar distintos animales y sus características y capacidades. Algunas de estas clases serían `Mamifero`, `Ave` y `Pez`. Muchas características vienen dadas por el tipo de animales. Todos los mamíferos amamantan a sus crías. Todas las aves y todos los peces ponen huevos. Hasta ahí es sencillo. Aunque también hay mamíferos que ponen huevos (los ornitorrincos), pero estos son una excepción.

Pero ... ¿Qué pasa con capacidades como las de nadar (desplazarse sobre o bajo el agua, de la forma que sea), correr y volar? Estas son características que se dan en distintos tipos de animales.

Todos los peces nadan. Pero también algunos mamíferos como los humanos, y los cetáceos, como las ballenas y los delfines. E incluso algunas aves, como el pato.

Todos los mamíferos corren (incluso los perezosos, por lentamente que lo hagan). Pero también algunas aves como el avestruz y el pato. Ningún pez lo hace.

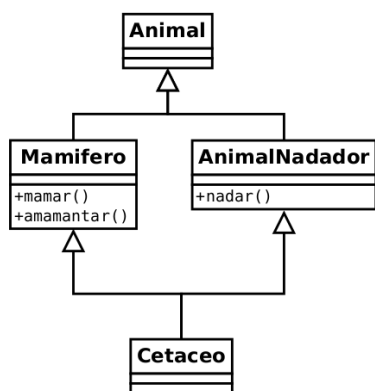
Casi todas las aves vuelan, pero no todas. No lo hace el avestruz ni lo hace la gallina. Algunos mamíferos, como los murciélagos, vuelan. E incluso hay peces voladores.

Parece claro que hay que crear una jerarquía de clases que corresponda a la taxonomía de los animales, es decir, a su división jerárquica en tipos y subtipos. Pero, ¿qué hacer con respecto a las características como volar, nadar, etc?

### Primera solución: Herencia múltiple

Crear clases `AnimalCorredor`, `AnimalVolador`, `AnimalNadador`.

Podría crearse, por ejemplo, la siguiente estructura de clases para los cetáceos, que son mamíferos.



Este es un ejemplo de herencia múltiple, en la que una clase es subclase de más de una. Las clases tienen entre ellas una relación de tipo “es un/una”. Una instancia de una subclase lo es también de la superclase, y por ello, hereda todos los atributos y métodos de la superclase.

La herencia múltiple plantea problemas. Por ejemplo, al crear un `Cetaceo` se podría ejecutar dos veces el constructor de `Animal`, lo que normalmente no es deseable. También, un método de `Animal` podría estar redefinido en `Mamifero` y `AnimalNadador`, pero no en `Cetaceo`. Entonces, para un `Cetaceo`, ¿cuál se ejecutaría?

Algunos lenguajes de programación, como por ejemplo C++, permiten la herencia múltiple. Pero la gran mayoría de los importantes en la actualidad no. **Java no permite la herencia múltiple.**

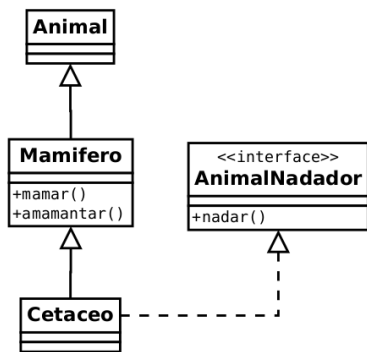
## Actividad 4.32

Crea un diagrama para una estructura de clases que incluya las clases Mamifero, Ave, Pez, AnimalCorredor, AnimalVolador, AnimalNadador, Cetaceo, PonedorDeHuevos, Perro, Murcielago, Humano, Pato, Avestruz, Delfin, PezVolador, Ornitorrinco. Puedes utilizar el programa dia (<https://sourceforge.net/projects/dia-installer/>). El tipo de diagrama que debes crear con él es un diagrama de UML.

## Segunda solución: Interfaces

Una interfaz (*interface* en inglés) define un conjunto de métodos. Una clase que implementa la interfaz debe implementar los métodos que define la interfaz.

La clase Cetaceo podría haberse creado dentro de la siguiente estructura de clases. En ella, AnimalNadador ya no es una clase, sino una interfaz que contiene el método nadar(). La clase Cetaceo implementa esta interfaz.



El uso de interfaces permite evitar la herencia múltiple. La relación entre una interfaz y la clase que la implementa ya no es una relación de tipo “es un/una”, sino de tipo “implementa”. Es decir, que un Cetaceo no “es un” AnimalNadador (que es una clase) sino que “implementa” AnimalNadador (que es una interfaz).

Bien pensado, este diseño es conceptualmente más elegante que el anterior. Los animales nadadores se diferencian enormemente entre ellos, tanto que es difícil pensar en un atributo común a todos ellos (un Humano, un Pato, un PezVolador). Lo único que tienen en común es que nadan.

La interfaz AnimalNadador contiene la cabecera de un método nadar.

```

package animales;

public interface AnimalNadador {

    public void nadar(int maxDist);

}
  
```

Se puede definir una clase con la opción **implements AnimalNadador**. Para que se puedan crear instancias de esta clase, debe implementar el método nadar. Si no lo hace, la debe definirse con la opción **abstract**, porque tiene un método que debe implementarse en clases descendientes.

Las clases Animal, Mamifero y Cetaceo se podrían definir de la siguiente manera.

```

package animales;

public class Animal {

    private final String nombre;

    public String getNombre() {
        return nombre;
    }

    public Animal(String nombre) {
        this.nombre = nombre;
    }
}
  
```

```
}  
  
}
```

```
package animales;  
  
public class Mamifero extends Animal {  
  
    Mamifero(String nombre) {  
        super(nombre);  
    }  
  
    protected void amamantar() {  
        System.out.printf("Mamífero %s amamantando.\n", this.getNombre());  
    }  
  
    protected void mamar() {  
        System.out.printf("Mamífero %s mamando.\n", this.getNombre());  
    }  
  
}
```

```
package animales;  
  
public class Cetaceo extends Mamifero implements AnimalNadador {  
  
    Cetaceo(String nombre) {  
        super(nombre);  
    }  
  
    @Override  
    public void nadar(int maxDist) {  
        System.out.printf("Cetáceo %s nadando.\n", this.getNombre());  
    }  
  
}
```

#### Actividad 4.33

Crema un diagrama para una estructura de clases que incluya las clases o interfaces Animal, Mamifero, Ave, Pez, AnimalCorredor, AnimalVolador, AnimalNadador, Cetaceo, PonedorDeHuevos, Perro, Murcielago, Humano, Pato, Avestruz, Delfin, PezVolador, Ornitorrinco. Debes decidir qué es una clase y qué es una interfaz. En general, las interfaces definen solo una capacidad o conjunto de capacidades de los animales.

#### Actividad 4.34

Implementa en Java la estructura de clases anterior. Cada clase debe incluir al menos un atributo y un método. Cada interfaz debe incluir al menos un método. Basta con que todas las clases tendrán un atributo `nombre` con su correspondiente `getter`, y que el nombre se pase en el constructor. Aparte del `getter` para el atributo `nombre`, debe haber un método correspondiente con algo que haga el animal, como correr, volar, poner huevos, etc. Si una clase implementa una interfaz, basta con que implemente el método o los métodos de la interfaz.

Las clases que corresponden a animales determinados pueden tener un método que devuelva una onomatopeya para el sonido que hace el animal. La clase `Perro` podría tener un método `String ladrar()`.

Para probar las clases, crea un `array` `AnimalNadador[] acuuario`, añade en él instancias de las clases `Persona` y `Delfin`, que solo tienen en común que implementan la interfaz `AnimalNadador`. Por último, para cada elemento del `array`, ejecuta el método `nadar()`. Si no has creado aún ningún método `nadar()` en la interfaz `AnimalNadador`, hazlo ahora.

Puedes hacer más pruebas similares, creando `arrays` de objetos que pertenecen a diferentes subclases de una misma clase o que implementan una misma interfaz.

#### Actividad 4.35

Crea una clase abstracta `Cantidad` a cuyo constructor se le pasa un número entero. Si se le pasa un número negativo o cero, el constructor debe lanzar una excepción de tipo `IllegalArgumentException`.

Debe tener un método abstracto `String representacion()` que devuelva un `String` que representa el número. Crea subclases tuyas que implementen el método `representacion()` de diferentes formas.

`CantidadPalotes` que devuelva una secuencia de palotes (caracteres `I`), tantos como el propio número. Pero para una cantidad mayor que 100, devuelve `null`.

`CantidadCuadradosYPalotes`, que devuelva un carácter `■` por cada cinco unidades, y palotes para el resto, hasta un máximo de 4. Pero para una cantidad mayor que 500, devuelve `null`.

`CantidadCuadradosRellenos`, que devuelva un carácter `■` por cada cinco unidades, y un carácter para el resto: `|` para 1, `└` para 2, `┐` para 3, `□` para 4. Pero para una cantidad mayor que 500, devuelve `null`.

`CantidadPseudoRomanos` que devuelva una cadena con una `M` por cada millar, una `C` por cada centenar, una `X` por cada decena, y una `I` por cada unidad. Pero para una cantidad mayor o igual que 3000, devuelve `null`.

(Opcional) `CantidadRomanos` que devuelva la representación del número como número romano. Pero solo para números hasta 3000. Para números mayores, debe devolver `null`.

Redefine el método `toString` en cada una de estas clases, de manera que muestre el nombre de la clase y la cantidad, como un número entero. Por ejemplo: `"CantidadCuadradosYPalotes(6)"`.

Para probar estas clases, crea un `array` de tipo `Cantidad[]`, que contenga varios elementos. Para cada uno de ellos, muestra el resultado de ejecutar el método `representacion`.

#### Actividad 4.36

Cambia el planteamiento de la estructura de clases anterior para que la clase `Cantidad` ya no tenga un método `representacion()`, sino que implemente una interfaz `Formateador` con un método `String representacion()`. Cada subclase de `Cantidad` debe tener una implementación distinta de este método.

En realidad, no se trata de escribir nuevo código, sino de reestructurar (refactorizar) la jerarquía de clases creada para el anterior ejercicio.

Ejecuta el programa de prueba anterior. ¿Has tenido que cambiar el código del programa de prueba?

Las interfaces permiten crear código genérico aplicable a cualquier clase, siempre que implemente determinados métodos definidos en una interfaz. Se explica con un ejemplo a continuación.

El siguiente programa contiene un método que ordena un `array` de números enteros. Compara el primer elemento (`i=0`) con todos los siguientes (de `j=i+1` a `nums.length`). Si el primer elemento `nums[i]` es mayor que otro `nums[j]`, intercambia ambos. A lo largo de la ejecución del bucle interior (de `j=i+1` a `nums.length`), se pueden intercambiar varias veces `nums[i]` y `nums[j]`, de manera que, al final, `nums[i]` no es mayor que ninguno de los

números en las siguientes posiciones. Cuando *i* es 0 y termina de ejecutarse el bucle interior, en `nums[0]` está el valor mínimo del *array*. En sucesivas ejecuciones del bucle interior, con valores de *i* que aumentan de uno en uno, se va situando en la posición *i* el valor mínimo de los elementos en las posiciones desde *i* inclusive hasta el final.

```
package ordenanumeros;

import java.util.Arrays;

public class OrdenaNumeros {

    public static void ordenaArrayInt(int[] nums) {
        for (int i = 0; i < nums.length - 1; i++) {
            for (int j = i + 1; j < nums.length; j++) {
                if (nums[j] < nums[i]) {
                    int aux = nums[i];
                    nums[i] = nums[j];
                    nums[j] = aux;
                }
            }
        }
    }

    public static void main(String[] args) {
        int[] arrNum = {3, 0, 1, 5, 6, 10, 5, 14};
        System.out.printf("Array desordenado: %s\n", Arrays.toString(arrNum));
        ordenaArrayInt(arrNum);
        System.out.printf("Array ordenado: %s\n", Arrays.toString(arrNum));
    }
}
```

Este algoritmo podría utilizarse para ordenar *arrays* de elementos de cualquier tipo que tengan un operador de comparación `<`, como por ejemplo `double`. Pero ... ¿Podría utilizarse para ordenar un *array* de `String`? Esta clase no tiene un operador de comparación `<`, pero sí un método `int compareTo(String anotherString)` que devuelve -1, 0 o 1 según el `String` `anotherString` sea menor, igual o mayor, en orden alfabético, que el `String` para el que se ejecuta. Así que podría utilizarse este método para ordenar un *array* de `String` (¡prueba a hacerlo!). De hecho, para ordenar un *array* de objetos de cualquier clase, bastaría con que la clase tuviera un método `comparaA(Objeto otroObjeto)`, donde *Clase* es la clase en cuestión. Esto se puede garantizar haciendo que la clase implemente la siguiente interfaz.

```
public interface Comparable {

    public int comparaA(Object otro);

}
```

El tipo del otro objeto, `otro`, es `Object`, para que el método se pueda utilizar con objetos de cualquier clase. De esta forma, el siguiente método permite ordenar *arrays* de objetos de cualquier clase, siempre que implemente la interfaz `Comparable` y por tanto, su método `comparaA`.

```
public static void ordena(Comparable[] objs) {
    for (int i = 0; i < objs.length; i++) {
        for (int j = i + 1; j < objs.length; j++) {
```

```
        if (objs[j].comparaA(objs[i]) < 0) {
            Comparable aux = objs[i];
            objs[i] = objs[j];
            objs[j] = aux;
        }
    }
}
```

**Actividad 4.37**

Utiliza la interfaz Comparable para ordenar objetos de una clase Persona, contenidos en un *array*, por su edad. Utiliza para ello el método *ordena*. Crea la clase Persona con atributos String nombre e int edad. Haz que Persona implemente la interfaz, y crea el método comparaA en Persona.

Nota: al método comparaA se le pasarán objetos de la clase Persona, pero su parámetro es Object otro. Se puede convertir a la clase Persona utilizando un *casting* de la siguiente manera:

```
Persona otraPers = (Persona) otro
```

Esto funcionará siempre que otro sea un objeto de la clase Persona o de una clase descendiente de ella.

**Actividad 4.38**

Ahora define el método comparaA en la clase Persona de manera que se ordenen las personas alfabéticamente por su nombre.

Después, haz los cambios necesarios para que se ordenen las personas por orden alfabético descendiente.

**Actividad 4.39**

Crea una clase Longitud que represente una medida en metros, centímetros y milímetros. Añade un atributo Longitud altura a la clase Persona para su altura en metros y centímetros. Crea un nuevo constructor (puedes dejar el que ya existe) al que se le pasa la altura en metros y centímetros en dos parámetros de tipo int.

Crea un *array* de objetos de clase Persona, de manera que cada una tenga una estatura asignada. Haz los cambios necesarios en la clase Persona para ordenar un *array* de Persona por estatura, utilizando el método *ordena*.