Departamento de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

PRÁCTICA 6: Polimorfismo

OBJETIVO

La programación orientada a objetos es un paradigma de programación que usa objetos y sus interacciones para diseñar las aplicaciones. Se basa en varias técnicas, que incluyen abstracción, encapsulación, polimorfismo y herencia. Esta práctica se centra en el polimorfismo y en la forma de aplicarlo dentro del lenguaje de programación Java. Esta práctica también presenta la herramienta `javadoc`, que genera documentación HTML a partir de los archivos fuente (`.java`) de Java.

POLIMORFISMO

El polimorfismo es la habilidad de los objetos de responder con distintos comportamientos ante un mismo mensaje. En Java una forma de utilizar el polimorfismo es mediante la sobrecarga de métodos.

Sobrecarga de método

La sobrecarga es la posibilidad de tener dos o más funciones o métodos con el mismo nombre pero funcionalidad diferente. Es decir, dos o más operaciones con el mismo nombre realizan acciones diferentes. El compilador usará una u otra dependiendo de los parámetros usados.

Java permite definir dos o más métodos dentro de la misma clase que tengan el mismo nombre, pero con sus listas de parámetros distintas. Cuando ocurre esto, se dice que los métodos están **sobrecargados** y a este proceso se le denomina **sobrecarga de método**.

Cuando se invoca a un método sobrecargado, Java utiliza el tipo y/o el número de argumentos como guía para determinar la versión del método sobrecargado que realmente debe llamar. Por eso, los métodos sobrecargados deben diferenciarse en el tipo y/o en el número de parámetros. Aunque los métodos sobrecargados puedan devolver diferentes tipos de valores, el tipo devuelto por sí solo, es insuficiente para distinguir dos versiones del método. Cuando Java encuentra una llamada a un método sobrecargado, simplemente ejecuta la versión del método cuyos parámetros coinciden con los argumentos utilizados en la llamada al método.

Veamos un ejemplo de sobrecarga de métodos:

```
package fp2.poo.practica6;

public class Practica6Ejercicio01 {

    public void metodoSobrecargado() {
        System.out.println("Sin parametros");
    }

    public void metodoSobrecargado(int a) {
        System.out.println("Con un parametro entero: " +a);
    }

    public void metodoSobrecargado(int a, int b) {
        System.out.println("Con dos parametros a y b: practica6 "  + a + " " + b );
    }

    public double metodoSobrecargado(double a) {
        System.out.println("Con un parametro double: " + a);
        return a * a;
    }

    //public int metodoSobrecargado(int a) {
    //    System.out.println("Con un parametro entero: " +a);
    //}
}
class Main {
    public static void main( String args[ ]) {
        Practica6Ejercicio01 obj = new Practica6Ejercicio01 ();
        double result = 0.0d;
        obj.metodoSobrecargado();
        obj.metodoSobrecargado(10);
        obj.metodoSobrecargado(10, 20);
        result = obj.metodoSobrecargado(123.2d);
        System.out.println(" valor devuelto al invocar a obj.metodoSobrecargado(123.2d): "
                           + result );
    }
 }
```

Practica6Ejercicio01.java

Descargue y descomprima el código de la plataforma para realizar la práctica 6.

Observe que el fichero `Practica6Ejercicio01.java`, contiene dos clases `Practica6Ejercicio01` y `Main`. Una de ellas es pública y otra visible a nivel de paquete. Este será el formato general para las pruebas en esta práctica, aunque se recomienda utilizar un fichero por clase de forma habitual. Y recuerde que la compilación en este caso genera dos archivos `.class`, uno por clase.

Use el *makefile* proporcionado y compile y ejecute el programa mediante el siguiente comando:

`make -f makeP6 prueba01`

Cambie el nivel de acceso de la clase **`Main`** a nivel **`public`** y compílelo de nuevo con el comando anterior. Observe que produce un error en compilación debido a que aparecen dos clases públicas en el mismo fichero.

Obsérvese que **`metodoSobrecargado()`** ha sido sobrecargado cuatro veces (y además una versión con comentarios). La primera versión no tiene parámetros; la segunda tiene un parámetro entero; la tercera tiene dos parámetros enteros y la cuarta tiene un parámetro **`double`**. El hecho

de que la cuarta versión de **metodoSobrecargado()** devuelva un valor de tipo `double` no está relacionado con la sobrecarga, ya que los tipos devueltos no juegan ningún papel en la resolución de la sobrecarga.
Quite los comentarios al quinto método sobrecargado y ejecute de nuevo el comando anterior e interprete el resultado.

Promoción automática de tipo en la sobrecarga de método

Cuando se llama a un método sobrecargado, Java busca una versión del método cuyos parámetros coincidan con los argumentos utilizados en la llamada al método. Sin embargo, esta coincidencia no tiene por qué ser siempre exacta. En algunos casos, las conversiones de tipo automáticas de Java pueden jugar un papel importante en la resolución de la sobrecarga. Por ejemplo, consideremos el siguiente programa.

```
package fp2.poo.practica6;


public class Practica6Ejercicio02 {

    public void metodoSobrecargado() {
        System.out.println("Sin parámetros");
    }

    //public void metodoSobrecargado(int a) {
    //    System.out.println("Con un parametro entero: "+a);
    //}

    public double metodoSobrecargado(double a) {
        System.out.println("Con un parametro double: "+a);
        return a * a;
    }
}

class Main {
    public static void main( String args[ ]) {
        Practica6Ejercicio02 ob = new Practica6Ejercicio02();
        int i = 88;
        ob.metodoSobrecargado();
        ob.metodoSobrecargado(i);     //Esto llama a metodoSobrecargado (double)
        ob.metodoSobrecargado(123.2); //Esto llama a metodoSobrecargado (double)
    }
 }
```

Practica6Ejercicio02.java

Compile y ejecute el programa mediante el siguiente comando.

make -f makeP6 prueba02

Obsérvese que en esta versión no se define el método **metodoSobrecargado(int)**. Cuando se llama a **metodoSobrecargado()** con un argumento entero, no se encuentra ningún método cuyos parámetros coincidan exactamente con el argumento. Sin embargo, Java puede convertir automáticamente un entero en un **double** y esta conversión puede ser utilizada para resolver la llamada al método. Por eso, cuando no encuentra el método **metodoSobrecargado(int)**, Java convierte la variable **i** a tipo **double** y llama a **metodoSobrecargado(double)**. Por supuesto, si hubiese estado definido el método

`metodoSobrecargado(int)`, Java lo habría llamado. Java emplea su conversión de tipo automática sólo si no existe una coincidencia exacta entre parámetros y argumentos.

Quite los comentarios del método **`metodoSobrecargado(int)`**, vuelva a ejecutar el comando anterior, y observe el resultado.

Ejercicio:

En el siguiente ejemplo se prueban distintas promociones de tipos en la llamada a métodos sobrecargados.
Cada tipo de dato simple contiene su implementación de método sobrecargo. Se trata de ver a qué tipo promociona si no existe la versión de su método.
Sustituya XXX por cada uno de los tipos para los que está implementada la sobrecarga de método. Una vez que haya sustituido XXX por un tipo concreto (por ejemplo por `byte`), comente el método que contiene un parámetro del tipo que haya elegido (Si XXX lo ha sustituido por byte, comente las tres líneas del método `public void metodoSobrecargado(byte arg)`).

```
package fp2.poo.practica6;


public class Practica6Ejercicio03 {

    public void metodoSobrecargado(byte arg) {
        System.out.println("Metodo: metodoSobrecargado(byte  obj)");
    }

    public void metodoSobrecargado(short arg) {
        System.out.println("Metodo: metodoSobrecargado(short obj)");
    }

    public void metodoSobrecargado(int arg) {
        System.out.println("Metodo: metodoSobrecargado(boolean obj)");
    }

    public void metodoSobrecargado(char arg) {
        System.out.println("Metodo: metodoSobrecargado(char obj)");
    }

    public void metodoSobrecargado(long arg) {
        System.out.println("Metodo: metodoSobrecargado(long obj)");
    }

    public void metodoSobrecargado(float arg) {
        System.out.println("Metodo: metodoSobrecargado(float obj)");
    }

    public void metodoSobrecargado(double arg) {
        System.out.println("Metodo: metodoSobrecargado(double obj)");
    }

    public void metodoSobrecargado(boolean arg) {
        System.out.println("Metodo: metodoSobrecargado(boolean obj)");
    }
}


class Main {
    public static void main( String args[ ]) {
        DemoSobrecarga ob = new DemoSobrecarga ();
       /*
        * Ponga el tipo a la variable arg y elimine el metodo sobrecargado
        * al que se le asociaria.
        */
        XXX arg = 88;
        ob.metodoSobrecargado( arg );
    }
}
```

Practica6Ejercicio03.java

Este ejemplo necesita volver a compilarlo cada vez que se realiza la eliminación de un método para probar la promoción del tipo de dato. Pruébelo para todos los tipos de datos simple mediante:

`make -f makeP6 prueba03`

Sobrecarga de constructores

Además de sobrecargar métodos normales, también se pueden sobrecargar los constructores. De hecho, en la mayoría de las clases que se implementan en el mundo real, la sobrecarga de los constructores es la norma y no la excepción. Vamos a aplicarlo a la clase `Saldo()` desarrollada en la práctica anterior:

```
package fp2.poo.practica6;


public class Saldo  {

    Double saldo;

    public Saldo() {
        saldo = 0.0d;
    }

    public Saldo(Double d) {
        saldo = d;
    }

    public Saldo(double d) {
        this.saldo = new Double(d);
    }

    public double getSaldo() {
        return this.saldo.doubleValue();
    }

    public void setSaldo(double d) {
        this.saldo = new Double(d);
    }
}
```

Saldo.java

Observe los tres constructores que se proporcionan en la clase `Saldo`.

```
package fp2.poo.practica6;


public class Practica6Ejercicio04 {
    public static void main (String args[]){
        Saldo obj1 = new Saldo();
        Saldo obj2 = new Saldo(new Double(5e+10));
        Saldo obj3 = new Saldo(20.5d);
        System.out.println("obj1.getSaldo() = " + obj1.getSaldo());
        System.out.println("obj2.getSaldo() = " + obj2.getSaldo());
        System.out.println("obj3.getSaldo() = " + obj3.getSaldo());
    }
}
```

Practica6Ejercicio04.java

Compile y ejecute el programa con el siguiente comando:

make -f makeP6 prueba04

Objetos como parámetros

Hasta ahora sólo se han utilizado tipos simples en los parámetros de los métodos. Sin embargo, es correcto y habitual pasar objetos a los métodos. Consideremos la siguiente versión de la clase **`Saldo`** en la que se ha incluido un nuevo constructor y el método **`igual`**.

```
package fp2.poo.practica6;


public class Saldo  {

    Double saldo;

    public Saldo() {
        saldo = 0.0d;
    }

    /*
     * Ejemplo de objeto como parametro
     */
    public Saldo(Saldo obj) {
        this.saldo = ((obj == null) ? null : obj.getSaldo());
    }

    public Saldo(Double d) {
        saldo = d;
    }

    public Saldo(double d) {
        this.saldo = new Double(d);
    }

    public double getSaldo() {
        return this.saldo.doubleValue();
    }

    public void setSaldo(double d) {
        this.saldo = new Double(d);
    }

    /*
     * Ejemplo de objeto como parametro
     */
    public boolean igual (Saldo obj) {
        boolean resultado = false;

        if ( (obj != null) && (obj.getSaldo() == this.getSaldo() )) {
            resultado= true;
        } else {
            resultado= false;
        }
        return resultado;
    }

}
```

Saldo.java

Como se puede ver, el método **`igual()`** dentro de **`Saldo`** compara dos objetos para ver si son iguales y devuelve el resultado. Es decir, compara el objeto que invoca al método con uno que se pasa como parámetro. Si tiene los mismos valores, entonces el método devuelve **`true`**. En caso contrario, devuelve **`false.`** Observe que el parámetro **`obj`** de **`igual()`** es de tipo **`Saldo.`**

En el siguiente fichero se proporciona una clase en la que se usan los métodos añadidos a la clase **`Saldo`**.

```
package fp2.poo.practica6;

public class Practica6Ejercicio05 {
    public static void main (String args[]){
        Saldo saldo1 = new Saldo( 6000.33d );
        Saldo saldo2 = new Saldo(saldo1);
        System.out.println("saldo1 .getSaldo() = " + saldo1.getSaldo());
        System.out.println("saldo2 .getSaldo() = " + saldo2.getSaldo());
        System.out.println("saldo1 == saldo2   : " + saldo1.igual(saldo2));
    }
}
```

Practica6Ejercicio05.java

Compile y ejecute el programa con el siguiente comando:

`make -f makeP6 prueba05`

Paso de argumentos

En general, en los lenguajes de programación hay dos formas de pasar un argumento a un método.

1. La primera forma es el paso de parámetros **por valor**. En este caso se copia el **valor** del argumento en el parámetro formal del método. Los cambios que se realizan sobre el parámetro formal del método no tienen efecto sobre el argumento utilizado en la llamada. En Java cuando se pasa un tipo simple a un método, se pasa por valor.

2. La segunda forma es el paso de parámetros **por referencia**. En este caso, el parámetro formal recibe la referencia del argumento utilizado en la llamada. Dentro del método, esta referencia se utiliza para acceder al argumento real especificado en la llamada. Esto significa que los cambios realizados al parámetro afectarán al argumento utilizado en la llamada del método. En Java cuando se pasa un objeto a un método, se pasa por referencia.

Devolución de objetos

Un método puede devolver cualquier tipo de dato, incluyendo los tipos de clases definidos por el programador. Se le ha añadido a la clase **`Saldo`** un método llamado **`crearCopia()`** que devuelve un nuevo objeto de tipo **`Saldo`**.

En el siguiente ejemplo se proporciona una clase en la que se usa este método de la clase **`Saldo`**.

```
package fp2.poo.practica6;

public class Practica6Ejercicio06 {
    public static void main (String args[]){
        Saldo saldo1 = new Saldo( 6000.33d );
        Saldo saldo2 = saldo1.crearCopia();
        System.out.println("saldo1 .getSaldo() = " + saldo1.getSaldo());
        System.out.println("saldo2 .getSaldo() = " + saldo2.getSaldo());
        System.out.println("saldo1 == saldo2   : " + saldo1.igual(saldo2));
    }
}
```

Practica6Ejercicio06.java

Nótese que la clase `Saldo.java`, incorpora además de lo mostrado anteriormente el siguiente método:

```
    /*
     * Ejemplo de devolución de objetos.
     */
    public Saldo crearCopia() {
        return new Saldo(this);
    }
```

Método `crearCopia()` de la clase `Saldo.java`

Compile y ejecute el programa con el siguiente comando:

`make -f makeP6 prueba06`

El programa anterior presenta otro aspecto importante. Como todos los objetos se crean dinámicamente utilizando el operador **`new`**, no es necesario preocuparse de su existencia una vez que el método en el que fue creado termine, ya que el objeto seguirá existiendo mientras haya una referencia a él en alguna parte del programa. Cuando no existan más referencias a ese objeto, entonces será eliminado por el sistema de recogida de basura.

Argumentos en la línea de órdenes (comandos)

Algunas veces es necesario pasar información a un programa cuando éste se ejecuta. Esto se consigue mediante el paso de *argumentos en la línea de órdenes* a **`main()`**. Los argumentos de la línea de órdenes es la información que directamente sigue al nombre del programa en la línea de órdenes cuando se ejecuta el programa. Acceder a los argumentos de la línea de órdenes dentro de un programa Java es bastante fácil, ya que son almacenados como cadenas en la matriz de **`String`** que se pasa a **`main()`**. Se puede obtener el número de argumentos en la línea de comandos con el atributo **`length`** del array. En Java, se conoce siempre el nombre de la aplicación ya que es el nombre de la clase en la cual el método principal está definido. Por esto el sistema de ejecución de Java no pasa el nombre de la clase que se invoca al método principal.

El siguiente programa muestra todos los argumentos que recibe de la línea de órdenes.

```
package fp2.poo.practica6;

public class Practica6Ejercicio07 {
    public static void main (String args[]){
        for ( int i = 0; i < args.length ; i++) {
            System.out.println("args [" + i + "] : " + args[i]);
        }
    }
}
```

Practica6Ejercicio07.java

Compile y ejecute el programa con el siguiente comando:

`make -f makeP6 prueba07`

Ejercicios.

1. Descomprima el fichero **`CodigoDeLaParteFinalDeLaPractica6.zip`**. Proporcionado en esta práctica. Dada la interfaz `TitularInterfaz.java` del paquete `fp2.poo.utilidades`, cuyo código aparece a continuación:

```
package fp2.poo.utilidades;


import  fp2.poo.pfpooXXX.Dni;

/**
 * Descripcion: Esta es una clase que representa un usuario de
 *              una cuenta bancaria.
 *
 * @version version 1.0 Mayo 2011
 * @author  Fundamentos de Programacion II
 */
public interface TitularInterfaz {

    /*
     *  Descripcion: Metodo de configuracion del atributo nombre.
     */
    public void setNombre( String nombre );

    /*
     *  Descripcion: Metodo getter de nombre.
     */
    public String getNombre( );


    /*
     *  Descripcion: Metodo de configuracion del atributo
     *               relacionado con el primer apellido.
     */
    public void setPrimerApellido( String primerApellido);

    /*
     *  Descripcion: Metodo getter del primer apellido.
     */
    public String getPrimerApellido( );

    /*
     *  Descripcion: Metodo de configuracion del atributo
     *               relacionado con el segundo apellido.
     */
    public void setSegundoApellido( String segundoApellido);

    /*
     *  Descripcion: Metodo getter del segundo apellido.
     */
    public String getSegundoApellido( );


    /*
     *  Descripcion: Metodo getter del dni.
     */
   public void setDni(Dni obj );

    /*
     *  Descripcion: Metodo getter del dni.
     */
    public Dni getDni( );

    /*
     *  Descripcion: Metodo de configuracion del atributo domicilio.
     */
    public void  setDomicilio( String domicilio);

    /*
     *  Descripcion: Metodo getter del domicilio.
     */
    public String getDomicilio( );
}
```

TitularInterfaz.java

Implemente la clase **`Titular`** en el paquete **`fp2.poo.pfpooXXX`**, siendo **`XXX`** el **login** del alumno proporcionado por el Centro de Cálculo (CDC).
La clase **`Titular`** mantiene la información relacionada con el titular de una cuenta bancaria. Deberá tener los atributos privados necesarios para almacenar dicha información.

2. Dada la interfaz **`CuentaBancariaInterfaz.java`** del paquete `fp2.poo.utilidades`, cuyo código aparece a continuación:

```
package fp2.poo.utilidades;

import  fp2.poo.pfpooXXX.Titular;
import  fp2.poo.pfpooXXX.NumeroDeCuenta;
import  fp2.poo.pfpooXXX.Saldo;


/**
 * Descripcion: Esta es una clase que representa una cuenta bancaria.
 *              Mantiene una asociacion entre un usuario (de tipo Usuario),
 *              su saldo (de tipo Saldo) y numero de cuenta
 *              (de tipo NumeroDeCuenta).
 *
 * @version version 1.0 Mayo 2011
 * @author  Fundamentos de Programacion II
 */
public interface CuentaBancariaInterfaz {

    /**
     * Descripcion: Configura el saldo de una cuenta.
     */
    public void            setSaldo (Saldo saldo);

    /**
     * Descripcion: Devuelve el saldo de una cuenta.
     */
    public Saldo           getSaldo();

    /**
     * Descripcion: Configura el numero de cuenta de una cuenta.
     */
    public void            setNumeroDeCuenta (NumeroDeCuenta numeroDeCuenta);

    /**
     * Descripcion: Devuelve el numero de cuenta de una cuenta.
     */
    public NumeroDeCuenta  getNumeroDeCuenta();

    /**
     * Descripcion: Configura el titular de cuenta de una cuenta.
     */
    public void            setTitular (Titular titular);

    /**
     * Descripcion: Devuelve el titular de cuenta de una cuenta.
     */
    public Titular         getTitular();
}
```

`NumeroDeCuentaInterfaz.java`

Implemente la clase **`CuentaBancaria`** en el paquete **`fp2.poo.pfpooXXX`**, siendo **`XXX`** el **login** del alumno proporcionado por el Centro de Cálculo (CDC).
La clase **`CuentaBancaria`** mantiene la información relacionada con una cuenta bancaria.

GESTION DE DOCUMENTACIÓN (`javadoc`)

Los comentarios de documentación, conocidos normalmente como *comentarios doc o comentarios javadoc*, permiten asociar directamente a nuestro código documentación de referencia para programadores utilizando el formato HTML.

Los comentarios de documentación se diseñan generalmente para introducir documentación de referencia en las clases con el nivel de detalle que la mayoría de los programadores necesita para la utilización de estas clases.

Un procedimiento para generar documentación a partir de los comentarios de documentación es usar el programa `javadoc`.

El programa `javadoc` se encarga de generar de manera automática la documentación correspondiente al código fuente que se ha creado. Para ello el programador debe añadir ciertos comentarios en su código fuente (siguiendo unas reglas) de forma que el programa `javadoc` pueda reconocerlas y así crear la documentación correspondiente. La sintaxis de este programa es la siguiente:

`javadoc [opciones] [paquetes] [ficheros_java]`

donde:

- `[opciones]` modifican el comportamiento de la herramienta `javadoc`. Algunas de las opciones más frecuentes son las siguientes:

`-public`	Genera la documentación correspondiente a las clases y miembros con modificadores `public` presentes en el código fuente a documentar.
`-protected`	Genera la documentación correspondiente a las clases y miembros con modificadores `public` y `protected` presentes en el código fuente a documentar. Esta es la opción por defecto.
`-package`	Genera la documentación correspondiente a las clases y miembros con modificadores `public`, `protected` y package presentes en el código fuente a documentar.
`-private`	Genera la documentación correspondiente a todas clases y miembros presentes en el código fuente a documentar.
`-d directorio`	Opción encargada de indicar a `javadoc` el directorio destino de la documentación a generar.
`-sourcepath` ruta	Opción encargada de indicar a `javadoc` la ruta donde se encuentra los paquetes que se indican en el comando. Cada una de las rutas se indicarán separadas por :.
`-charset`	Controla la codificación de los ficheros generados. En esta práctica se utilizará la codificación “UTF-8”.

- La opción `paquetes` indica los paquetes de los que se desea generar la documentación. La ubicación de los paquetes vendrá dada por el valor de la opción `-sourcepath` que se indique previamente.
- La opción `ficheros_java` indica los ficheros con el código fuente (ficheros `.java`) de los que se desea generar la documentación. Se tendrá que especificar la ruta correspondiente, no utilizándose en este caso el valor de la opción `-sourcepath`.

Ejercicio 1:
Se proporciona la siguiente interfaz:

```
package fp2.poo.practica6.javaDoc;


public interface SaldoInterfaz{
    public double getSaldo();
    public Double getSaldoDouble();
    public void setSaldo(Double d);
    public void setSaldo(double d);
}
```

SaldoInterfaz.java

Ejecute la siguiente orden:

`make -f makeJavadoc  prueba01`

Esta orden genera mediante el programa **`javadoc`** documentación asociada a la interfaz `SaldoInterfaz.java`, en el directorio **`doc`**. Utilice el navegador para visualizar el fichero **`index.html`** que se ha generado en el directorio **`doc`**, y navegue por la información generada por la herramienta.

La anatomía de un comentario de documentación

Los comentarios de documentación comienzan con los tres caracteres `/**` y se extienden hasta el siguiente `*/`. Todos los comentarios de documentación describen el identificador cuya declaración sigue inmediatamente después. Los caracteres `*` iniciales se ignoran en las líneas de estos comentarios, y también los espacios en blanco precediendo al `*` inicial. La primera frase del comentario es el resumen del identificador, donde "frase" significa todo el texto hasta el primer punto seguido por un espacio en blanco. Consideremos el siguiente comentario de documentación:

```
/**
  * Devuelve el saldo. El valor devuelto es de
  * tipo double.
  */
 public double getSaldo();
```

El resumen del método **`getSaldo`** será "**`Devuelve el saldo`**". La primera frase de un comentario de documentación debe ser un buen resumen.

Reconvierta los comentarios dados en `SaldoInterfaz.java` en comentarios al estilo **`javadoc`** (cambiar `/*` por `/**`) y genere de nuevo la documentación mediante el comando dado anteriormente.

A menudo se insertan etiquetas de HTML en los comentarios de documentación, que actúan como enlaces de referencias cruzadas con otra documentación. Podemos utilizar casi cualquier etiqueta de HTML, excepto las etiquetas de cabecera `<h1>`, `<h2>`..., que están reservadas para su uso en la documentación generada. Para insertar los caracteres <, >, o `&` debe utilizarse `&lt;`, `&gt;` o `&amp;` respectivamente. Si hay que poner un carácter `@` al principio de una línea, debe utilizarse `&#064`, si no, se supone que es el comienzo de una etiqueta de un comentario de documentación.

Sólo se procesan los comentarios de documentación inmediatamente anteriores a una clase, interfaz, método o atributo. Si hay algo diferente de espacios en blanco o comentarios entre un comentario de este tipo y lo que describe, el comentario se ignora. Por ejemplo, si ponemos un comentario de documentación al principio de un archivo y existe una sentencia `package` o `import` entre el comentario y una sentencia `class`, el comentario no se utilizará. Los comentarios de documentación se aplican a todos los identificadores que se declaran en una sola sentencia, por tanto se evitan este tipo de declaraciones cuando se van a utilizar estos comentarios.

Si no se proporciona un comentario de documentación para un método heredado, el método "hereda" el comentario. Esto en general es suficiente, especialmente cuando una clase hereda una interfaz. Los métodos de la implementación generalmente no hacen más que lo que la interfaz especifica, o al menos nada que no esté descrito en un comentario de documentación. Es conveniente poner un comentario explícito cuando heredemos comentarios de documentación, para que nadie pueda pensar que olvidamos documentar el método. Por ejemplo, en la clase **`Saldo`** que implementa la interfaz se puede indicar de la siguiente forma:

```
// Hereda comentarios de documentacion
public double getSaldo(){
    //...
}
```

si un método hereda comentarios de documentación de una superclase y una interfaz, se utiliza el comentario de la interfaz.

La clase `Saldo00` proporcionada a continuación implementa la interfaz `SaldoInterfaz`. Esta clase hereda todos los comentarios de la interfaz que implementa, por tanto se ha indicado en el método de la clase.

```
package fp2.poo.practica6.javaDoc;

import  fp2.poo.practica6.javaDoc.SaldoInterfaz;


public class Saldo00 implements SaldoInterfaz {

    Double saldo;

    //Hereda comentarios de documentacion
    public Saldo00(Double d) {
        saldo = d;
    }

    //Hereda comentarios de documentacion
    public Saldo00(double d) {
        this.saldo = new Double(d);
    }

    //Hereda comentarios de documentacion
    public double getSaldo() {
        return this.saldo;
    }

    //Hereda comentarios de documentacion
    public Double getSaldoDouble() {
        return this.saldo;
    }

    //Hereda comentarios de documentacion
    public void setSaldo(Double d) {
        this.saldo = d;
    }

    //Hereda comentarios de documentacion
    public void setSaldo(double d) {
        this.saldo = new Double(d);
    }
}
```

Saldo00.java

Ejecute la siguiente orden

make -f makeJavadoc prueba02

Observe la documentación generada.

Etiquetas

Las etiquetas que se pueden utilizar en los comentarios `javadoc` son las siguientes:

- **`@author`** `nombre`

  Esta etiqueta sirve para indicar el autor de la clase. Se pueden especificar tantos párrafos de **`@author`** como deseemos.

- **`@version`** `datos de la versión (número y fecha)`

  Esta etiqueta sirve para indicar la versión del software que contiene esta clase.

- **`@since`** `version inicial en la que aparece`

  La etiqueta **`@since`** permite indicar una especificación de versión con información de cuándo la entidad etiquetada se añadió al sistema.
  Etiquetar la "versión de nacimiento" puede ayudarnos a seguir qué entidades son nuevas, y por tanto necesitan documentación o pruebas más intensas. Una etiqueta **`@since`**, en una clase o interfaz aplica a todos sus miembros que no tengan su propia etiqueta **`@since.`**

- **`@param`** `nombre`

  La etiqueta **`@param`** documenta un solo parámetro de un método. Si utilizamos etiquetas **`@param`** hay que emplear una por cada parámetro del método. A la etiqueta **`@param`** debe seguirle el nombre del parámetro del método que documenta

- **`@return`** `valor devuelto`
  La etiqueta **`@return`** documenta el valor de retorno de un método.

- **`@throws`** y **`@exception`**

  La etiqueta **`@throws`** documenta una excepción lanzada por un método. Si utilizamos etiquetas **`@throws`**, debe haber una por cada tipo de excepción que lance el método. Esta lista a menudo es más amplia que sólo las excepciones comprobadas de la cláusula **`throws`**. Es una buena idea declarar todas las excepciones en la cláusula **`throws`**, se requieran o no, y esto también es cierto al utilizar etiquetas **`@throws`**. Por ejemplo, supongamos que nuestro método comprueba sus parámetros para asegurarse de que ninguno de ellos es **`null`**, y lanza una excepción **`NullPointerException`** si encuentra un parámetro **`null`**. Debemos declarar la excepción **`NullPointerException`** en nuestra cláusula **`throws`** y en nuestras etiquetas **`@throws`**. La etiqueta **`@exception`** es equivalente a **`@throws`**.

- **`@deprecated`**

  La etiqueta **`@deprecated`** marca un identificador como desaprobado, es decir, inadecuado para continuar con su uso. El código que utiliza un tipo, constructor o campo desaprobado puede generar un aviso cuando compila. Es conveniente asegurarse de que la entidad desaprobada continúa funcionando de forma que no dañemos código existente que no haya sido actualizado. La desaprobación nos ayuda a animar a los usuarios de nuestro código a actualizarse a la última versión, preservando la integridad del código existente. Los usuarios pueden moverse a las nuevas versiones cuando lo prefieran, en vez de ser forzados a actualizarse tan pronto como lancemos una nueva versión de nuestros tipos. Debemos recomendar a los usuarios que sustituyan los tipos desaprobados. Observe que en la clase **`String`** aparecen algunos métodos y constructores marcados como *deprecated*, en el siguiente enlace:
  `http://download.oracle.com/javase/1.4.2/docs/api/java/lang/String.html`

- **`@see`**

  La etiqueta **`@see`** crea un enlace de referencia cruzada a otra documentación **`javadoc`**. Se puede nombrar cualquier identificador, aunque debe calificarse de la siguiente forma **`paquete.clase#miembro`**. Por ejemplo, generalmente podremos nombrar a un miembro de una clase utilizando su nombre simple. Sin embargo, si el miembro es un método sobrecargado, deberemos especificar a qué sobrecarga del método nos referimos indicando los tipos de los parámetros. Podemos especificar una clase o interfaz que pertenezca al paquete actual utilizando su nombre sin calificar, pero los tipos de otros paquetes se deben especificar con sus nombres completamente calificados. Los miembros de tipos se especifican utilizando # antes de su nombre.

  En el siguiente ejemplo se muestra la clase `Saldo01` que implementa la interfaz `SaldoInterfaz`, en ella se muestran varios ejemplos de las etiquetas comentadas anteriormente.

```java
package fp2.poo.practica6.javaDoc;

import  fp2.poo.practica6.javaDoc.SaldoInterfaz;

 /**
  * Clase de ejemplo para mostrar el funcionamiento de javadoc.
  *
  * @author Fundamentos de Programacion II
  * @since  18-Mayo-2011
  * @version 1.0
  */
public class Saldo01 implements SaldoInterfaz {

    Double saldo;

    /**
      * Constructor de Saldo01.
      *
      * @param d de tipo Double
      * @see Saldo01#Saldo01(double)
      */
    public Saldo01(Double d) {
        saldo = d;
    }

    /**
      * Constructor de Saldo01.
      *
      * @param d de tipo double
      * @see Saldo01#Saldo01(Double )
      */
    public Saldo01(double d) {
        this.saldo = new Double(d);
    }

     /**
      * Devuelve el saldo.
      *
      * @return devuelve el saldo como double
      * @see Saldo01#getSaldoDouble()
      */
   public double getSaldo() {
        return this.saldo;
    }

     /**
      * Devuelve el saldo en un objeto Double.
      *
      * @return devuelve el saldo como Double
      * @see Saldo01#getSaldo()
      */
    public Double getSaldoDouble() {
        return this.saldo;
    }

     /**
      * Devuelve el saldo en un objeto Double.
      *
      * @param d de tipo Double.
      * @see Saldo01#getSaldo(double)
      * @throws NullPointerException si el objeto es null.
      */
    public void setSaldo(Double d) throws NullPointerException {
        this.saldo = d;
    }

     /**
      * Devuelve el saldo en un objeto Double.
      *
      * @param d de tipo double.
      * @see Saldo01#getSaldo(Double)
      */
    public void setSaldo(double d) {
        this.saldo = new Double(d);
    }
}
```

Saldo01.java

Ejecute la siguiente orden

make -f makeJavadoc prueba03

La siguiente tabla muestra el ámbito en el que se pueden utilizar las etiquetas `javadoc`:

	Comentarios en Clases	Comentarios en métodos
`@author`	Si	NO
`@param`	No	Si
`@return`	No	Si
`@throws`	No	Si
`@version`	Si	No
`@see`	Si	Si
`@since`	Si	Si
`@deprecated`	Si	Si

Ejercicio

Ponga los comentarios **`javadoc`** a las siguientes clases ya implementadas en las prácticas anteriores, siguiendo el ejemplo proporcionado en esta práctica:

- `Saldo.java`
- `SaldoInterfaz.java`
- `Dni.java`
- `DniInterfaz.java`
- `NumeroDeCuenta.java`
- `NumeroDeCuentaInterfaz.java`
- `TitularInterfaz.java`
- `Titular.java`
- `CuentaBancariaInterfaz.java`
- `CuentaBancaria.java`

Trabajo a entregar

1. Implemente la clase **`Main`** en el fichero **`Main.java`** del paquete **`fp2.poo.pfpooXXX`**, siendo **`XXX`** el **login** del alumno, para crear en el método **`main`** objetos de la clase **`Titular`** y **`CuentaBancaria`**, implementados en la anterior sección Ejercicios. Proporcione un makefile para realizar la compilación y ejecución. Nota: la clase **`Titular`** usa **`Dni`**, utilizado en la práctica anterior, y la clase **`CuentaBancaria`** utiliza las clases **`Titular`**, **`NumeroDeCuenta`**, y **`Saldo`**, y por tanto deben estar definidas y compiladas.
2. Comprima el directorio de nombre su **`uvus`** en el fichero **`uvus.zip`** y entréguelo en la actividad correspondiente a la práctica.