Gobierno de Canarias
Servicio Canario de Empleo

Centro GASTRONÓMICO del Atlántico

GOBIERNO DE ESPAÑA
MINISTERIO DE EDUCACIÓN Y FORMACIÓN PROFESIONAL
MINISTERIO DE TRABAJO Y ECONOMÍA SOCIAL
SERVICIO PÚBLICO DE EMPLEO ESTATAL
SEPE

Programación en Java 2023

Tema 7 Programación Orientada a Objetos

Tema 7

1. Programación Orientada a Objetos

1.1. Clases y Objetos

1.1.1. Objetivos

- Presentar el concepto de objeto, clase, atributo, método e instancia.
- Interpretar el código fuente de una aplicación Java donde aparecen implementados los conceptos anteriores.
- Construir una aplicación Java sencilla, convenientemente especificada, que emplee los conceptos anteriores.
- Introducir el concepto de constructor de una clase en Java.
- Interpretar el código fuente de una aplicación Java donde aparecen declaraciones y llamadas a constructores.
- Construir una aplicación Java sencilla, convenientemente especificada, que emplee clases en los que se declaren explícitamente constructores.

1.1.2. Introducción

Aunque parezca una obviedad, la base de la Programación Orientada a Objetos es el objeto. En la vida real todos los objetos tienen una serie de características y un comportamiento. Por ejemplo, una puerta tiene color, forma, dimensiones, material... (goza de una serie de características) y puede abrirse, cerrarse... (posee un comportamiento). En Programación Orientada a Objetos, un objeto es una combinación de unos datos específicos y de las rutinas que pueden operar con esos datos. De forma que los dos tipos de componentes de un objeto son:

- **Campos o atributos**: componentes de un objeto que almacenan datos. También se les denomina variables miembro. Estos datos pueden ser de tipo primitivo (*boolean*, *int*, *double, char,* ...) o, a su vez, de otro tipo de objeto (lo que se denomina agregación o composición de objetos). La idea es que un atributo representa una propiedad determinada de un objeto.
- **Rutinas o métodos**: es una componente de un objeto que lleva a cabo una determinada acción o tarea con los atributos.

1.1.3. Clases

Una clase representa al conjunto de objetos que comparten una estructura y un comportamiento comunes. Una clase es una combinación específica de atributos y métodos y puede considerarse un tipo de dato de cualquier tipo no primitivo. Así, una clase es una especie de plantilla o prototipo de objetos: define los atributos que componen ese tipo de objetos y los métodos que pueden emplearse para trabajar con esos objetos. Aunque, por otro lado, una clase también puede estar compuesta por métodos estáticos que no necesitan de objetos (como las clases construidas en los capítulos anteriores que contienen un método estático main). La declaración de una clase sigue la siguiente sintaxis:

```
[modificadores] class IdentificadorClase {
	// Declaraciones de atributos
	...
	// Declaraciones de métodos
	...
}
```

Nota: Los identificadores de las clases deberían ser simples, descriptivos y sustantivos y, en el caso de nombres compuestos, con la primera letra de cada uno en mayúsculas.

1.1.4. Instancias

Una instancia es un elemento tangible (ocupa memoria durante la ejecución del programa) generado a partir de una definición de clase. Todos los objetos empleados en un programa han de pertenecer a una clase determinada.

Aunque el término a veces se emplea de una forma imprecisa, un objeto es una instancia de una clase predefinida en Java o declarada por el usuario y referenciada por una variable que almacena su dirección de memoria. Cuando se dice que Java no tiene punteros simplemente se indica que Java no tiene punteros que el programador pueda ver, ya que todas las referencias a objeto son de hecho punteros en la representación interna.

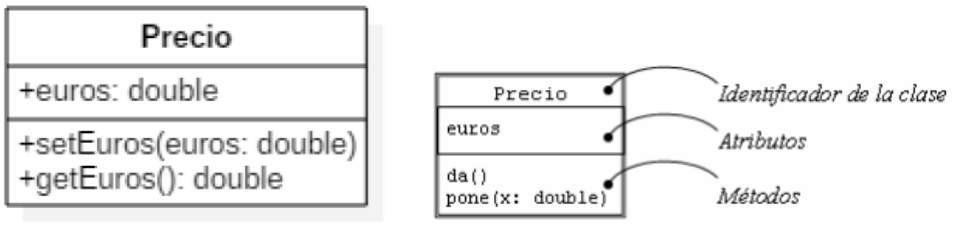
En general, el acceso a los atributos se realiza a través del operador punto, que separa al identificador de la referencia del identificador del atributo (idReferencia.idAtributo). Las llamadas a los métodos para realizar las distintas acciones se llevan a cabo separando los identificadores de la referencia y del método correspondiente con el operador punto (idReferencia.idMetodo(parametros)).

Ejemplo sencillo de clase e instancia:

El siguiente código muestra la declaración de la clase Precio. La clase Precio consta de un único atributo (euro) y dos métodos: uno que asigna un valor al atributo llamado setEuros sin devolver ningún valor y otro que devuelve el valor del atributo llamado getEuros:

```
/**
* Ejemplo de declaracion de la clase Precio
* double getEuros() --> devuelve el valor almacenado en euros
* void setEuros(double euros) --> almacena valor en euros
*/
public class Precio {
	// Atributo o variable miembro
	public double euros;
	// Metodos
	public double getEuros() {
		return euros;
	}
	public void setEuros(double euros) {
		this.euros = euros;
	}
}
```

Gráficamente una clase puede representarse como un rectángulo según se muestra en la siguiente figura:



- +/-: indica el ámbito, '-' representa privated y '+' representa 'public'.

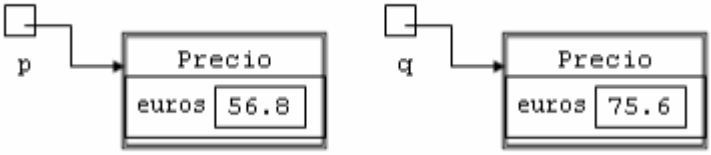
El anterior código puede compilarse de la siguiente forma:

```
$ javac Precio.java
```

generando el archivo de bytecodes Precio.class. Este archivo no es directamente ejecutable por el intérprete, ya que el código fuente no incluye ningún método principal (main). Para poder probar el código anterior, puede construirse otro archivo con el código fuente que se muestra a continuación:

```
/**
* Ejemplo de uso de la clase Precio
*/
public class PruebaPrecio {
        public static void main (String [ ] args ) {
                Precio p; // Crea una referencia de la clase Precio
                p = new Precio(); // Crea el objeto de la clase Precio
                // Llamada al metodo setEuros que asigna 56.8 al atributo euros
                p.setEuros(56.8);
                System.out.println("Valor = " + p.da()); // Llamada al metodo get Euros
                // que devuelve el valor de euros
                Precio q = new Precio(); // Crea una referencia y el objeto de la clase Precio
                q.euros = 75.6; // Asigna 75.6 al atributo euros
                System.out.println("Valor = " + q.euros);
        }
}
```

Representación gráfica del espacio de la memoria utilizado por las referencias e instancias de la clase Precio durante la ejecución del método main() de la clase PruebaPrecio:



El código anterior puede compilarse y ejecutarse mostrando la siguiente salida por pantalla:

```
$ javac PruebaPrecio.java
$ java PruebaPrecio
Valor = 56.8
Valor = 75.6
```

Explicación del código anterior

Para poder trabajar con objetos se tendrá que seguir un proceso de dos pasos. Lo primero que debe hacer el programa es crear una referencia o puntero de la clase Precio con el identificador p. De forma similar a cómo se declara una variable de un tipo primitivo, la declaración del identificador de la referencia se realiza con la sintaxis:

identificadorClase identificadorReferencia;

// En el ejemplo anterior: Precio p;

Precio p;

Creación de la referencia p:

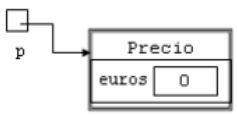
La referencia o puntero, p, tiene como misión almacenar la dirección de memoria de (apuntar a) los componentes de la instancia que todavía no ha sido creada ni referenciada. En este momento se dice que la referencia p, recién creada, almacena una dirección de memoria nula (que no corresponde a objeto alguno) o null. El segundo paso del proceso para trabajar con objetos lleva a la creación de una nueva instancia referenciada por p, que se realiza con la sentencia:

identificadorReferencia = new identificadorClase();

// Ejemplo: p = new Precio();

p = new Precio();

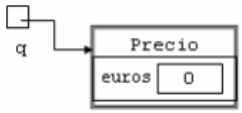
Creación de la nueva instancia de la clase Precio referenciado por p:



A esta operación se le denomina también instanciación. Aunque las dos operaciones anteriores (creación de la referencia y creación de la instancia referenciada) pueden realizarse conjuntamente en la misma línea de código:

Precio q = new Precio();

Creación de la referencia q y de la nueva instancia de la clase Precio referenciado por q:



El resultado de la ejecución del código anterior son dos nuevas instancias de la clase Precio referenciados respectivamente por p y q. El atributo euros de cada una de las nuevas instancias de la clase Precio es accesible a través del identificador de la referencia y del operador punto (p.euros y q.euros). Los métodos da y pone pertenecientes a la clase Precio son accesibles a través del identificador de la referencia y del operador punto:

p.getEuros()

p.setEuros(56.8)

q.getEuros()

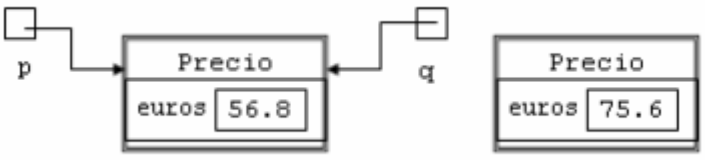
q.setEuros(75.6)

En el caso de los métodos, la instancia mediante la cual se realiza la llamada correspondiente actúa como un parámetro o argumento implícito del método.

Si se asigna una referencia a otra mediante una sentencia de asignación, no se copian los valores de los atributos, sino que se tiene como resultado una única instancia apuntada por dos referencias distintas. Por ejemplo:

q = p;

Resultado de la asignación de valores entre referencias:



En este caso ¿qué ocurre con la instancia referenciada previamente por q? Dicha instancia se queda sin referencia (inaccesible). Esto puede ser un problema en algunos lenguajes de programación, como es el caso de Pascal o de C, que utilizan variables dinámicas y que necesitan liberar explícitamente el espacio en memoria reservado para las variables que van a dejar de ser referenciadas. La gestión dinámica de la memoria suele ser una tarea engorrosa para el programador y muy dada a la proliferación de errores de ejecución. Para evitar tales inconvenientes, Java permite crear tantas instancias como se desee (con la única limitación de la memoria que sea capaz de gestionar el sistema), sin que el programador tenga que preocuparse de destruirlas o liberarlas cuando ya no se necesiten. El entorno de ejecución de Java elimina automáticamente las instancias cuando detecta

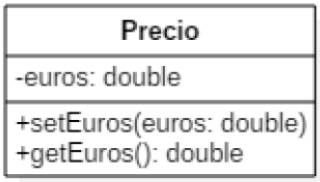
que no se van a usar más (cuando dejan de estar referenciadas). A este proceso se le denomina recogida o recolección de basura (garbage collection).

1.1.5. Modificadores de visibilidad

El modificador **public** indica que la componente del método es accesible fuera del código de la clase a la que pertenece el componente a través del operador punto. El modificador **private** indica que el componente solamente es accesible a través de los métodos de la propia clase. El modificador **protected** se verá posteriormente. En el siguiente código se declara el atributo euros con el modificador *private*:

```
/**
* Ejemplo de declaracion de la clase Precio
* double getEuros() --> devuelve el valor almacenado en euros
* void setEuros(double euros) --> almacena valor en euros
*/
public class Precio {
	// Atributo o variable miembro
	private double euros;
	// Metodos
	public double getEuros(){
		return euros;
	}
	public void setEuros(double euros){
		this.euros = euros;
	}
}
```

Gráficamente una clase puede representarse como un rectángulo según se muestra en la siguiente figura:



Si se construye otro código que intente utilizar directamente el atributo euros:

```
/**
* Ejemplo de uso de la clase PrecioPrivado
* double getEuros() --> devuelve el valor almacenado en euros
* void setEuros(double euros) --> almacena valor en euros
* euros --> Atributo de acceso privado
*/
public class PruebaPrecioPrivado {
	public static void main (String [] args ) {
	precioPrivado p = new precioPrivado(); // Crea instancia
	p.pone(56.8); // Asigna 56.8 a euros
```

```
        System.out.println("Valor = " + p.getEuros());
        p.euros = 75.6; // Asigna 75.6 a euros - ERROR
        System.out.println("Valor = " + p.euros); // Tambien ERROR
        }
}
```

se producirá un error de compilación:

```
$ javac PruebaPrecioPrivado.java
pruebaPrecioPrivado.java:15: euros has private access in precioPrivado
p.euros=75.6;
^
pruebaPrecioPrivado.java:16: euros has private access in precioPrivado
System.out.println("Valor = " + p.euros);
                                 ^
```

ya que el atributo euros sólo es accesible a través de los métodos de la clase getEuros() y setEuros().

La utilización del modificador private sirve para implementar una de las características de la programación orientada a objetos: el ocultamiento de la información o encapsulación.

La declaración como público de un atributo de una clase no respeta este principio de ocultación de información. Declarándolos como privados, no se tiene acceso directo a los atributos del objeto fuera del código de la clase correspondiente y sólo puede accederse a ellos de forma indirecta a través de los métodos proporcionados por la propia clase. Una de las ventajas prácticas de obligar al empleo de un método para modificar el valor de un atributo es asegurar la consistencia de la operación.

Nota: Por ejemplo, un método que asigne valor al tributo euros de un objeto de la clase Precio puede garantizar que no se le asignará un valor negativo.

1.1.6. ¿Qué significa la palabra reservada this?

El **this** sirve para hacer referencia a un método o propiedad del objeto actual.

El this, en el caso anterior, se utiliza dentro del método setEuros() para diferenciar la variable euros pasada por parámetro del atributo euros de la clase Precio, de esta manera tenemos 2 variables que se llaman igual pero con dos alcances diferentes, la pasada por parámetro solo tiene vida útil dentro del método mientras que this.euros hasta que el objeto se destruya.

¿Dónde se puede usar el this?

Puede referirse a cualquier miembro del objeto actual desde dentro de un método de instancia o un constructor. Si se intenta utilizar dentro de un método estático dará el siguiente error:

"Cannot use This in a static context"

Nota: No se puede usar en un método que es estático ya que un método estático se puede acceder sin la instancia del objeto, por lo que no podemos hacer referencia a propiedades o métodos que todavía no existen.

1.1.7. Método toString()

El método **toString()** nos permite mostrar la información completa de un objeto, es decir, el valor de sus atributos.
Este método también se hereda de **java.lang.Object**, por lo que deberemos sobrescribir este método.

A continuación se muestra un ejemplo de uso del método *toString()*:

```
Public class Empleado(){
        // Atributos
        private String nombre;
        private String apellido;
        private int edad;
        // Métodos
        public Empleado(String nombre, String apellido, int edad){
                this.nombre = nombre;
                this.apellido = apellido;
                this.edad = edad;
        }
        public String toString (){
                String mensaje = "El empleado se llama " + nombre + " " + apellido + " con " +
                edad + " años " + "y un salario de " + salario;
                return mensaje;
}
```

El mensaje lo configuramos nosotros como queramos.

Ejemplo práctico:

```
public class EmpleadoApp {
        public static void main(String[ ] args){
                //Creamos dos objetos distintos
                Empleado empleado1=new Empleado("Fernando", "Ureña", 23, 600);
                Empleado empleado2=new Empleado("Antonio", "Lopez", 28, 900);
                Empleado empleado3=new Empleado("Alvaro", "Perez", 19, 800);
```

```
                //Mostramos la informacion del objeto
                System.out.println(empleado1.toString());
                System.out.println(empleado2.toString());
                System.out.println(empleado3.toString());
        }
}
```

La salida del programa anterior es la siguiente:

```
El empleado se llama Fernando Ureña con 23 años y un salario de 600.0
El empleado se llama Antonio López con 28 años y un salario de 900.0
El empleado se llama Álvaro Pérez con 19 años y un salario de 800.0
```

1.1.8. Constructores

Aunque en un principio pueda parecer lo contrario, un constructor no es en realidad un método estrictamente hablando. Un **constructor** es un elemento de una clase cuyo identificador coincide con el de la clase correspondiente y que tiene por objetivo obligar a y controlar cómo se inicializa una instancia de una determinada clase, ya que el lenguaje Java no permite que las variables miembro de una nueva instancia queden sin inicializar. Además, a diferencia de los métodos, los constructores sólo se emplean cuando se quiere crear una nueva instancia.

Por defecto toda clase tiene un constructor sin parámetros cuyo identificador coincide con el de la clase y que, al ejecutarse, inicializa el valor de cada atributo de la nueva instancia: los atributos de tipo primitivo se inicializan a 0 o false, mientras que los atributos de tipo objeto (referencia) se inicializan a null.

En el ejemplo de la clase PruebaPrecio, que utiliza una instancia de la clase Precio, la llamada al constructor se produce en la sentencia p = new Precio();. Mientras que la ejecución de new genera una nueva instancia y devuelve su dirección de memoria, la ejecución del constructor Precio() inicializa los valores de los atributos.

```
public class PruebaPrecio {
        public static void main (String [ ] args ) {
                Precio p; // Crea una referencia de la clase Precio
                p = new Precio(); // Crea el objeto de la clase Precio y realiza
                // una llamada al metodo constructor
                // Resto del codigo ...
        }
}
```

1.1.8.1 Declaración de un constructor

La declaración de un constructor diferente del constructor por defecto, obliga a que se le asigne el mismo identificador que la clase y que no se indique de forma explícita un tipo de valor de retorno. La existencia o no de parámetros es opcional. Por otro lado, la sobrecarga permite que puedan declararse varios constructores (con el mismo identificador que el de la clase), siempre y cuando tengan un tipo y/o número de parámetros distinto. Por ejemplo, para la clase Fecha se declaran dos constructores, el primero sin parámetros (por lo tanto, se redefine el constructor por defecto) y el segundo con tres parámetros:

```
/**
* Declaracion de la clase Fecha
*/
public class Fecha {
        // Atributos o variables miembro
        private int dia;
        private int mes;
        private int anho;
        /**
        * Constructor 1
        * Asigna los valores 1, 1 y 2000 a los atributos
        * dia, mes y anho respectivamente
        */
        public Fecha(){
                dia = 1;
                mes = 1;
                anho = 2000;
        }
        /**
        * Constructor 2
        * @param ndia el dia del mes a almacenar
        * @param nmes el mes del anho a almacenar
        * @param nanho el anho a almacenar
        */
        public Fecha(int ndia, int nmes, int nanho){
                dia = ndia;
                mes = nmes;
                anho = nanho;
                }
```

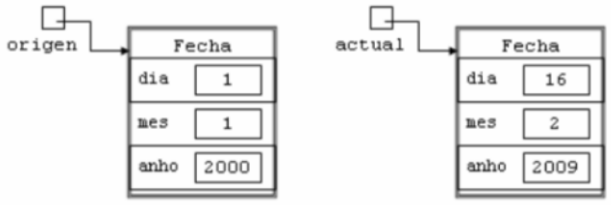
```
        public String toString(){
                return dia + "/" + mes + "/" + anho;
        }
}
```

La sobrecarga permite que puedan declararse varios constructores (con el mismo identificador que el de la clase), siempre y cuando tengan un tipo y/o número de parámetros distinto.

Para probar el código anterior, se construye el siguiente programa:

```
/**
* Ejemplo de uso de la clase Fecha
*/
public class PruebaFecha {
    public static void main (String [ ] args) {
        Fecha origen = new Fecha();
        Fecha actual = new Fecha(16,2,2009);
        System.out.println("Primera fecha: " + origen.toString());
        System.out.println("Segunda fecha: " + actual.toString());
    }
}
```

Resultado de la ejecución de los respectivos constructores para las nuevas instancias referenciadas por origen y actual:



El código anterior puede compilarse y ejecutarse, mostrando la siguiente salida por pantalla:

```
$ javac PruebaFecha.java
Primera fecha: 1/1/2000
Segunda fecha: 16/2/2009
```

Nota: una vez construido un constructor ya no se puede emplear el constructor por defecto sin parámetros. Si se desea trabajar con él, es necesario declararlo explícitamente.

1.1.8.2 *Más sobre la declaración y uso de varios constructores*

Un **constructor** sólo puede ser llamado por otros constructores o por métodos de clase (static). En el siguiente código se muestra un ejemplo de cómo se declaran dos constructores *CuentaBancaria*: el primero no tiene parámetros y hace una llamada al segundo constructor, que tiene un parámetro numérico real.

```
/**
* Declaracion de la clase CuentaBancaria
* Ejemplo de declaracion de variables
* metodos estaticos y uso de this
*/
public class CuentaBancaria {
        // Atributos o variables miembro
        private double saldo;
        public static int totalCuentas=0;
        // Métodos
        public CuentaBancaria( ){
                this(0.0); // Llamada al constructor que tiene un parametro
        }
        public CuentaBancaria( double ingreso ){
                saldo = ingreso;
        incCuentas();
        }
        public double saldo(){
                return saldo;
        }
        public static void incCuentas (){
                totalCuentas++;
        }
        public void transferencia( CuentaBancaria origen ){
                saldo += origen.saldo;
                origen.saldo=0;
        }
}
```

La sentencia *this(0.0);* en el primer constructor realiza la llamada el segundo constructor. Ejemplo de programa que emplea la clase *CuentaBancaria*:

```
/**
* Ejemplo de uso de la clase CuentaBancaria
*/
public class PruebaCuentaBancaria {
        public static void main(String[] args){
        System.out.println("Total cuentas: " + CuentaBancaria.totalCuentas);
        CuentaBancaria c1;
        c1 = new CuentaBancaria();
        System.out.println("Nueva cuenta con: " + c1.saldo() + " euros");
        System.out.println("Total cuentas: " + CuentaBancaria.totalCuentas);
        CuentaBancaria c2;
        c2 = new CuentaBancaria(20.0);
        System.out.println("Nueva cuenta con: " + c2.saldo() + " euros");
        System.out.println("Total cuentas: " + CuentaBancaria.totalCuentas);
        }
}
```

La ejecución del código anterior origina la siguiente salida por pantalla:

```
$ java PruebaCuentaBancaria
Total cuentas: 0
Nueva cuenta con: 0.0 euros
Total cuentas: 1
Nueva cuenta con: 20.0 euros
Total cuentas: 2
```

1.2. Herencia

1.2.1. Objetivos

- Definir el concepto de herencia entre clases.
- Interpretar el código fuente de una aplicación Java donde aparecen clases relacionadas mediante la herencia.
- Construir una aplicación Java sencilla, convenientemente especificada, que haga uso de la herencia entre clases.

1.2.2. Definición de herencia

La **herencia** es una propiedad que permite la declaración de nuevas clases a partir de otras ya existentes. Esto proporciona una de las ventajas principales de la Programación Orientada a Objetos: la reutilización de código previamente desarrollado ya que permite a una clase más específica incorporar la estructura y comportamiento de una clase más general.

Cuando una clase B se construye a partir de otra A mediante la herencia, la clase B hereda todos los atributos, métodos y clases internas de la clase A. Además, la clase B puede redefinir los componentes heredados y añadir atributos, métodos y clases internas específicas.

Para indicar que la clase B (clase descendiente, derivada, hija o subclase) hereda de la clase A (clase ascendiente, heredada, padre, base o superclase) se emplea la palabra reservada **extends** en la cabecera de la declaración de la clase descendiente. La sintaxis es la siguiente:

```
public class ClaseB extends ClaseA {
        // Declaracion de atributos de la ClaseB.
        ...
        // Declaracion de metodos especificos de la ClaseB
        ...
        // Redeclaracion de componentes heredados de la ClaseA
        ...
}
```

Por ejemplo, a partir de la clase Precio:

```
/**
* Ejemplo de declaracion de la clase Precio
*/
public class Precio {
        // Variable de instancia
        public double euros;
        // Metodos publicos
```

```
        public double getEuros(){
                return euros;
        }
        public void setEuros(double x){
                euros = x;
        }
}
```

Se construye la clase Producto como descendiente de la clase Precio de la siguiente forma:

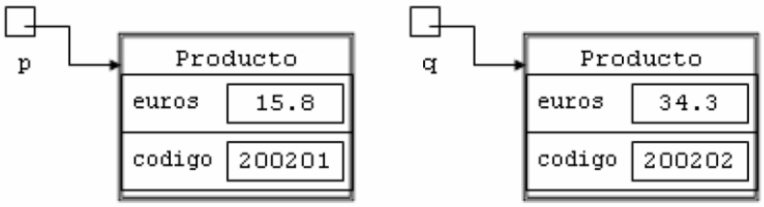
```
/**
* Ejemplo de declaracion de la clase Producto
* clase producto desciende de Precio
*/
public class Producto extends Precio {
        // Variable de instancia
        public int codigo;
        // Metodos publicos
        public int getCodigo(){
                return codigo;
        }
        public void setCodigo(int x){
                codigo = x;
        }
        public void setProducto(int codigo, double precio){
                setCodigo(codigo);
                setEuros(precio);
        }
        public String toString(){
                return "Codigo: " + codigo + " ; precio: " + euros + " euros";
        }
}
```

La clase PruebaClaseProducto trabaja con dos instancias de la clase Producto:

```
/**
* Demostracion de la clase Producto
*/
public class PruebaClaseProducto {
        public static void main (String [ ] args){
                Producto p = new Producto();
                p.setProducto(200201, 15.8);
                System.out.println(p.toString());
                Producto q = new Producto();
                q.setCodigo(200202);
                q.setEuros(34.3);
                System.out.println(q.toString());
        }
}
```

Durante la ejecución del código anterior, se generan las instancias referenciadas por p y q, cada una de las cuales está compuesta por dos atributos: euros, variable de instancia heredada de la clase Precio y codigo, variable de instancia específica de la clase Producto:

A continuación, se muestra representación gráfica de las instancias de la clase Producto:

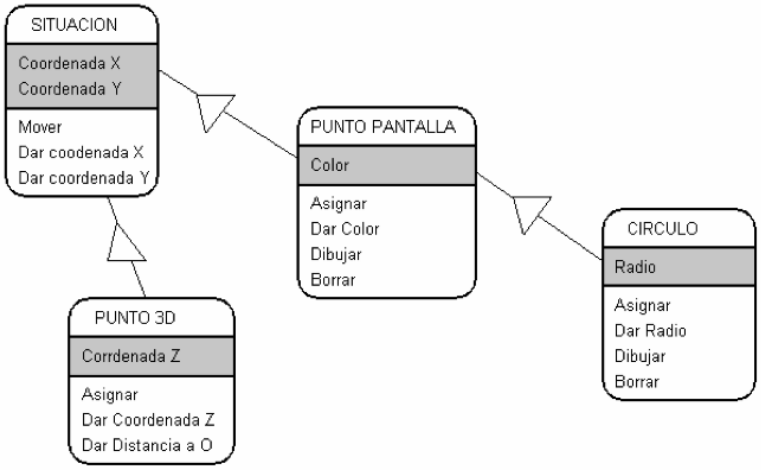


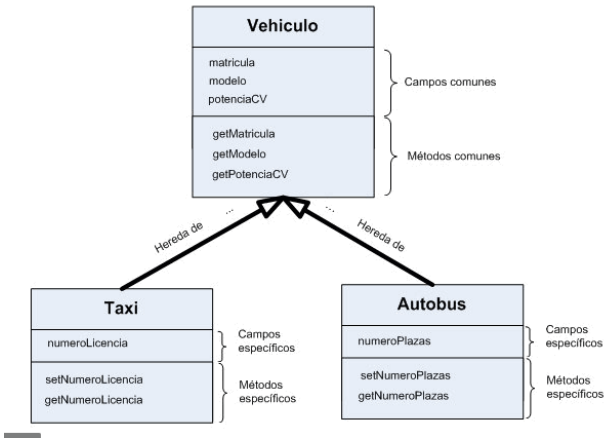
Por otro lado, la ejecución de PruebaClaseProducto produce la siguiente salida por pantalla:

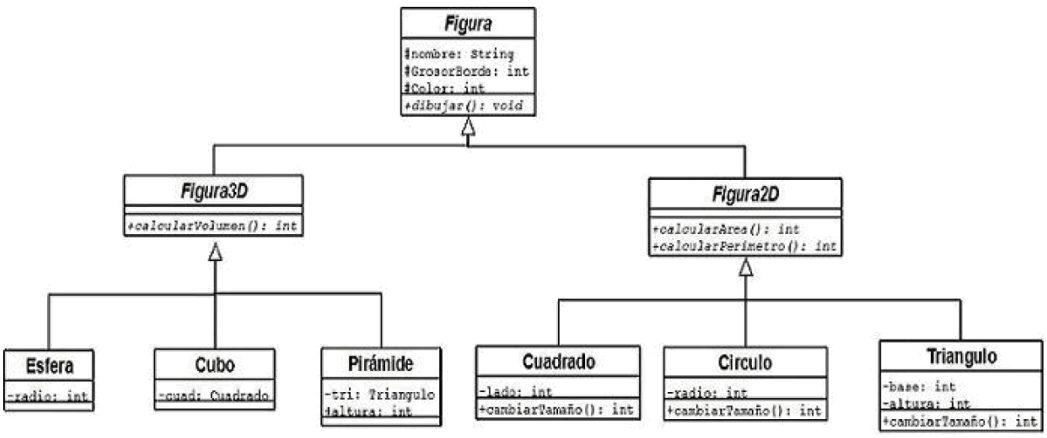
```
$ javac PruebaClaseProducto.java
$ java PruebaClaseProducto
Codigo: 200201 ; precio: 15.8 euros
Codigo: 200202 ; precio: 34.3 euros
```

1.2.3. Jerarquía de clases

Java permite múltiples niveles de herencia, pero no la herencia múltiple, es decir, **una clase sólo puede heredar directamente de una clase ascendiente**. Por otro lado, una clase puede ser ascendiente de tantas clases descendiente como se desee (un único padre, multitud de hijos). En la siguiente figura se muestra gráficamente un ejemplo de jerarquía entre diferentes clases relacionadas mediante la herencia:







1.2.4. Redefinición de elementos heredados

Como se ha comentado anteriormente la clase descendiente puede añadir sus propios atributos y métodos, pero también puede sustituir u ocultar los heredados. En concreto:

- Se puede declarar un nuevo **atributo** con el mismo identificador que uno heredado, quedando este atributo oculto. Esta técnica no es recomendable.
- Se puede declarar un nuevo **método de instancia** con la misma cabecera que el de la clase ascendiente, lo que supone su *sobreescritura*. Por lo tanto, la **sobreescritura** o redefinición consiste en que métodos adicionales declarados en la clase descendiente con el mismo nombre, tipo de dato devuelto y número y tipo de parámetros sustituyen a los heredados.
- Se puede declarar un nuevo **método de clase** con la misma cabecera que el de la clase ascendiente, lo que hace que éste quede oculto. Por lo tanto, los métodos de clase o estáticos (declarados como *static*) no pueden ser redefinidos.
- Un método declarado con el modificador final tampoco puede ser redefinido por una clase derivada.
- Se puede declarar un constructor de la subclase que llame al de la superclase de forma implícita o mediante la palabra reservada *super*.
- En general puede accederse a los métodos de la clase ascendiente que han sido redefinidos empleando la palabra reservada super delante del identificador del método. Este mecanismo sólo permite acceder al método perteneciente a la clase en el nivel inmediatamente superior de la jerarquía de clases.

1.2.5. ¿ Qué significa la palabra reservada super?

La palabra reservada **super** es similar a la palabra clave this. En POO, la palabra clave super se puede usar para acceder a cualquier atributo o método de la clase de la cual hereda nuestra clase.

La palabra clave super puede usarse a nivel de variable, método y constructor.

En el siguiente ejemplo se usa la palabra reservada super para hacer referencia al atributo color de la superclase Animal:

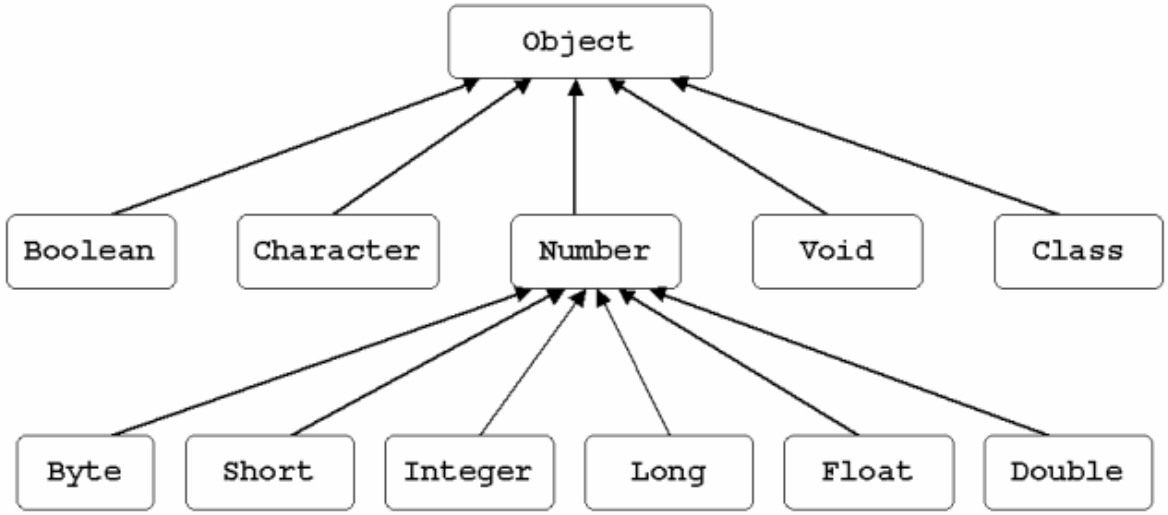
```
class Animal {
        String color="white";
}
class Dog extends Animal {
        String color="black";
        void printColor(){
        System.out.println(color);//prints color of Dog class
        System.out.println(super.color);//prints color of Animal class
        }
}
class Main {
        public static void main(String args[ ]){
```

```
        Dog d=new Dog();
        d.printColor();
        }
}
```

1.2.6. La clase Object

Independientemente de utilizar la palabra reservada **extends** en su declaración, todas las clases derivan de una superclase llamada **Object**: ésta es la clase raíz de toda la jerarquía de clases de Java.

A continuación, se muestra la jerarquía de clases predefinidas en Java:



Nota: El hecho de que todas las clases deriven implícitamente de la clase *Object* no se considera *herencia múltiple*.

Como consecuencia de ello, todas las clases tienen algunos métodos heredados de la clase Object. A continuación, se muestran algunos métodos de la clase predefinida Object:

Método	Función
clone()	Genera una instancia a partir de otra de la misma clase.
equals()	Devuelve un valor lógico que indica si dos instancias de la misma clase son iguales.
toString()	Devuelve un String que contiene una representación como cadena de caracteres de una instancia.
finalize()	Finaliza una instancia durante el proceso de recogida de basura.
hashCode()	Devuelve una clave hash (su dirección de memoria) para la instancia.
getClass()	Devuelve la clase a la que pertenece una instancia.

Es bastante frecuente tener que sobreescribir algunos de estos métodos. Por ejemplo, para verificar si dos instancias son iguales en el sentido de contener la misma información en sus atributos se debería sobreescribir el método **equals()**. El siguiente código muestra un ejemplo de módificación de la clase *Producto* para incluir una sobreescritura del método *equals()*:

```
public class Producto extends Precio {
        ...
        public boolean equals(Object a){
                if (a instanceof Producto)
                        return (codigo == a.getCodigo());
                else
                        return false;
        }
}
```

También es bastante habitual sobreescribir el método **toString()**.

1.2.7. Herencia y constructores

La subclase necesita normalmente que se ejecute el constructor de la superclase antes que su propio constructor para inicializar las variables de instancia heredadas. La solución consiste en utilizar la palabra reservada super seguida entre paréntesis de los parámetros correspondiente en el cuerpo del constructor de la subclase. Es decir, incluir la siguiente sentencia como primera línea de código:

super(argumentos opcionales);

De esta forma la implementación de un constructor de la clase descendiente sólo necesita inicializar directamente las variables de instancia no heredadas. Si no aparece como primera sentencia, el compilador inserta una llamada implícita super(); que inicializa las variables de instancia a cero, false, carácter nulo o null dependiendo de su tipo. Esta llamada en cadena a los constructores de las clases ascendientes llega hasta el origen de la jerarquía de clases, es decir, hasta el constructor de la clase Object. En cualquier caso, la creación de una nueva instancia mediante un constructor debe tener tres fases:

1. Llamada al constructor de la clase ascendiente
2. Se asignan valores a los atributos
3. Se ejecuta el resto del constructor

1.2.8. Casting o moldes entre objetos con relación de herencia

El **casting** o **moldeo** permite el uso de un objeto de una clase en lugar de otro de otra clase con el que haya una relación de herencia. Por ejemplo:

Object a = new Producto();

Entonces el objeto *a* es momentáneamente tanto una instancia de la *clase Object* como *Producto* (hasta que más adelante se le asigne un objeto que no sea un *Producto*). A esto se le llama **moldeo implícito**.

Por otro lado, si se escribe:

Producto b = a;

se obtendrá un error de compilación porque el objeto referenciado por a no es considerado por el compilador como un Producto. Sin embargo, se le puede indicar al compilador que a la referencia a se le va a asignar obligatoriamente un Producto.

Producto b = (Producto)a;

Este **moldeo explícito** introduce la verificación durante la ejecución de que a la referencia a se le ha asignado un *Producto* así que el compilador no genera un error. En el caso que durante la ejecución la referencia a no fuera a un *Producto*, se generaría una excepción. Para asegurar esta situación y evitar el error de ejecución se podría emplear el operador *instanceof*:

```
if (a instanceof Producto) {
        Producto b = (Producto)a;
}
```

1.3. Clases y métodos finales

Una clase declarada con la palabra reservada final no puede tener clases descendientes. Por ejemplo, la clase predefinida de Java Math está declarada como final.

A modo de ejemplo, se desarrolla una clase final MathBis (de operatividad similar a la clase Math estándar de Java) que incluye la declaración de dos métodos que calculan y devuelven respectivamente las siguientes funciones trigonométricas:

$$\operatorname{senh}^{-1}(x) = \ln\left(x + \sqrt{x^2 + 1}\right)$$

$$\cosh^{-1}(x) = \ln\left(x + \sqrt{x^2 - 1}\right)$$

El código fuente de la clase es:

```
/**
* Ejemplo de declaracion de una clase final
* Declaracion de la clase MathBis
*/
public final class MathBis {
        public static double asinh(double x) {
                return Math.log(x+Math.sqrt(x*x+1));
        }
        public static double acosh(double x) {
                return Math.log(x+Math.sqrt(x*x-1));
        }
}
```

Ejemplo de uso de la clase MathBis:

```
/**
* Ejemplo de uso de una clase final
* Declaracion de la clase pruebaMathBis
*/
public class PruebaMathBis {
        public static void main (String [ ] args) {
                for (int i=-5; i<10; i++) {
                        double x = i/5.0;
                        System.out.print("Para x = " + x);
                        System.out.print(": asinh(x) = " + MathBis.asinh(x));
                        System.out.println(", acosh(x) = " + MathBis.acosh(x));
                }
        }
}
```

Salida por pantalla de la ejecución del código anterior:

```
$ java PruebaMathBis
Para x = -1.0: asinh(x) = -0.8813735870195428, acosh(x) = NaN
Para x = -0.8: asinh(x) = -0.7326682560454109, acosh(x) = NaN
Para x = -0.6: asinh(x) = -0.5688248987322477, acosh(x) = NaN
Para x = -0.4: asinh(x) = -0.39003531977071504, acosh(x) = NaN
Para x = -0.2: asinh(x) = -0.19869011034924128, acosh(x) = NaN
Para x = 0.0: asinh(x) = 0.0, acosh(x) = NaN
Para x = 0.2: asinh(x) = 0.19869011034924142, acosh(x) = NaN
Para x = 0.4: asinh(x) = 0.3900353197707155, acosh(x) = NaN
Para x = 0.6: asinh(x) = 0.5688248987322475, acosh(x) = NaN
Para x = 0.8: asinh(x) = 0.732668256045411, acosh(x) = NaN
Para x = 1.0: asinh(x) = 0.8813735870195429, acosh(x) = 0.0
Para x = 1.2: asinh(x) = 1.015973134179692, acosh(x) = 0.6223625037147785
Para x = 1.4: asinh(x) = 1.1379820462933672, acosh(x) = 0.867014726490565
Para x = 1.6: asinh(x) = 1.2489833279048763, acosh(x) = 1.0469679150031885
Para x = 1.8: asinh(x) = 1.3504407402749723, acosh(x) = 1.1929107309930491
```

Por otro lado, un método declarado como final no puede ser redefinido por una clase descendiente. Los métodos que son llamados desde los constructores deberían declararse como final, ya que, si un constructor llama a un método que no lo sea, la subclase podría haberla redefinido con resultados indeseables.

1.4. Clases y métodos abstractos

Una **clase abstracta** es una clase de la que no se pueden crear instancias. Su utilidad consiste en permitir que otras clases deriven de ella. De esta forma, proporciona un modelo de referencia a seguir a la vez que una serie de métodos de utilidad general. Las clases abstractas se declaran empleando la palabra reservada abstract como se muestra a continuación):

public abstract class IdClase . . .

Una clase abstracta puede componerse de varios atributos y métodos, pero debe tener, al menos, un método abstracto (declarado también con la palabra reservada abstract en la cabecera). Los métodos abstractos no se implementan en el código de la clase abstracta pero las clases descendientes de ésta han de implementarlos o volver a declararlos como abstractos (en cuyo caso la subclase también debe declararse como abstracta). En cualquier caso, ha de indicarse el tipo de dato que devuelve y el número y tipo de parámetros. La sintaxis de declaración de un método abstracto es:

abstract modificador tipo_retorno idClase(lista_parametros);

Si una clase tiene métodos abstractos, entonces también la clase debe declararse como abstracta. Como los métodos de clase (static) no pueden ser redefinidos, un método abstracto no puede ser estático. Tampoco tiene sentido que declarar constructores abstractos ya que un constructor se emplea siempre al crear una instancia (y con las clases abstractas no se crean instancias).

Ejemplo de código con la declaración de clase abstracta:

```
/**
* Declaracion de la clase abstracta FiguraGeometrica
*/
public abstract class FiguraGeometrica {
        // Declaracion de atributos
        private String nombre;
        // Declaracion de metodos
        abstract public double area();
        public figuraGeometrica (String nombreFigura ){
                nombre = nombreFigura;
        }
        final public boolean mayorQue (FiguraGeometrica otra){
                return area()>otra.area();
        }
        final public String toString(){
                return nombre + " con area " + area();
        }
}
```

Como ejemplo de utilización de una clase abstracta en el siguiente código la clase Rectangulo se construye a partir de la clase abstracta FiguraGeometrica:

```
/**
* Ejemplo de uso de la declaracion de una clase abstracta
* Declaracion de la clase Rectangulo
*/
public class Rectangulo extends FiguraGeometrica {
        // Definición de atributos
        private double base;
        private double altura;
```

```
        // Definición de métodos
        public Rectangulo (double largo, double ancho){
                super("Rectangulo");
                base=largo;
                altura=ancho;
        }
        public double area (){
                return base * altura;
        }
}
```

Ejemplo de uso de la clase Rectangulo:

```
/**
* Ejemplo de uso de la clase Rectangulo
*/
public class pruebaRectangulo {
        public static void main (String [ ] args ){
                Rectangulo r1;
                r1 = new Rectangulo(12.5, 23.7);
                System.out.println("Area de r1 = " + r1.area());
                Rectangulo r2 = new Rectangulo(8.6, 33.1);
                System.out.println("Area de r2 = " + r2.toString());
                if (r1.mayorQue(r2))
                        System.out.println("El rectangulo de mayor area es r1");
                else
                        System.out.println("El rectangulo de mayor area es r2");
        }
}
```

Salida por pantalla de la ejecución del código anterior:

```
$ java PruebaRectangulo
Area de r1 = 296.25
Area de r2 = Rectangulo con area 284.66
El rectangulo de mayor area es r1
```

1.4.1. Polimorfismo

El **polimorfismo** es la habilidad de una función, método, variable u objeto de poseer varias formas distintas. Podríamos decir que un mismo identificador comparte varios significados diferentes.

El propósito del polimorfismo es implementar un estilo de programación llamado envío de mensajes en el que los objetos interactúan entre ellos mediante estos mensajes, que no son más que llamadas a distintas funciones.

Java tiene 4 grandes formas de polimorfismo (aunque conceptualmente, muchas más):

- Polimorfismo de asignación
- Polimorfismo puro
- Sobrecarga
- Polimorfismo de inclusión

1.4.1.1 Polimorfismo de asignación

El **polimorfismo de asignación** es el que está más relacionado con el enlace dinámico.

En java, una misma variable referenciada (Clases, interfaces…) puede hacer referencia a más de un tipo de Clase. El conjunto de las que pueden ser referenciadas está restringido por la herencia o la implementación.

Esto significa, que una variable A declarada como un tipo, puede hacer referencia a otros tipos de variables siempre y cuando haya una relación de herencia o implementación entre A y el nuevo tipo. Podemos decir que un tipo A y un tipo B son compatibles si el tipo B es una subclase o implementación del tipo A.

Supongamos este ejemplo:

```
abstract class vehiculo
{
        abstract public void iniciar();
}
class Coche extends Vehiculo
{
        @Override
        public void iniciar()
        {
        }
}
```

En él tenemos una clase que hereda de otra. La forma normal de instanciar una clase de tipo Coche sería esta…

Coche j = new Coche();

Sin embargo, el polimorfismo de asignación permite a una variable declarada como otro tipo usar otra forma, siempre y cuando haya una relación de herencia o implementación. Sabiendo esto, este fragmento demuestra el polimorfismo de asignación:

Vehiculo j = new Coche();

En el vemos como una variable inicializada como tipo *Vehiculo* puede usar el *polimorfismo de asignación* para hacer referencia a una clase de tipo *Coche*. Podemos decir que el **tipo estático** de la variable *j* es *Vehiculo*, mientras que su **tipo dinámico** es *Coche*.

Esto también puede hacerse con el nombre de interfaces implementadas.

```
interface Comprable
{
	public void comprar();
}
class Casa implements Comprable
{
	@Override
	public void comprar()
	{
	}
}
class Coche extends Vehiculo implements comprable
{
	@Override
	public void iniciar()
	{
	}
	@Override
	public void comprar()
	{

	}
}
```

Teniendo las anteriores clases iniciales, el siguiente código es una clara muestra del polimorfismo de asignación en Java.

Comprable a = new Casa();

a = new Coche();

1.4.1.2 Polimorfismo Puro

El **polimorfismo puro** se usa para nombrar a una función o método que puede recibir varios tipos de argumentos en tiempo de ejecución. Esto no lo debemos confundir con la sobrecarga, que es otro tipo de polimorfismo en tiempo de compilación. Conociendo el polimorfismo de asignación, podemos hacer una función que acepte varios tipos de objetos distintos en tiempo de ejecución. Veamos un ejemplo, usando las clases anteriormente mencionadas:

```
class PolimorfismoPuroTest
```

```
{
        public function funcionPolimorfica(Comprable ob)
        {
                // La función acepta cualquier "comprable", es decir,
                // cualquier objeto que implemente esa interfaz.
                // El tipo de objeto se determina en tiempo de ejecución. En nuestros ejemplos,
                // puede ser una casa o coche.
        }
}
```

En el ejemplo se ve como el método funcionPolimorfica es capaz de trabajar con varios objetos gracias al polimorfismo de asignación. Esto es lo que se conoce como polimorfismo puro.

1.4.1.3 Polimorfismo de sobrecarga

Muy similar al anterior, pero este se realiza en tiempo de compilación.

En el **polimorfismo de sobrecarga**, dos o más funciones comparten el mismo identificador, pero distinta lista de argumentos. Al contrario que el polimorfismo puro, el tipado de los argumentos se especifica en tiempo de compilación.

Es muy habitual ver esto en las clases envolventes (Integer, Float, etc...) y por eso mismo se va a mostrar un ejemplo de sobrecarga de la clase String de Java:

```
public final class String
implements java.io.Serializable, Comparable, CharSequence
{
        ...
        public static String valueOf(Object obj)
        {
                return (obj == null) ? "null" : obj.toString();
        }
        public static String valueOf(char data[ ])
        {
                return new String(data);
        }
        public static String valueOf(char data[ ], int offset, int count)
        {
                return new String(data, offset, count);
```

```
        }
        ...
}
```

En el ejemplo anterior vemos una misma función (*valueOf*) con diferentes listas de argumentos. En función de los argumentos especificados en el mensaje, la clase *String* utilizará uno u otro para adecuarse al contexto. Fijaros que la primera función admite un Objeto, la segunda un *array* de *Chars* y la tercera Un *array* de *Chars* y dos *integers*. Esto es el *polimorfismo de sobrecarga*, de definición similar al polimorfismo puro, pero de implementación muy distinta.

1.4.1.4 Polimorfismo de inclusión

La habilidad para redefinir por completo el método de una superclase en una subclase es lo que se conoce como **polimorfismo de inclusión** (o redefinición).

En él, una subclase define un método que existe en una superclase con una lista de argumentos (si se define otra lista de argumentos, estaríamos haciendo sobrecarga y no redefinición).

Un ejemplo muy básico:

```
abstract class Pieza
{
        public abstract void movimiento(byte X, byte Y);
}
class Alfil extends Pieza
{
        @Override
        public void movimiento(byte X, byte Y)
        {
        }
}
```

En el ejemplo vemos como la clase *Alfil* sobreescribe el método *movimiento()*. Esto es el *polimorfismo de inclusión*. Un error común de un desarrollador es pensar que el siguiente ejemplo es otra muestra de este tipo de polimorfismo:

```
class Caballo extends Pieza
{
        public void movimiento(int X, int Y)
        {
        }
```

```
}
```

El ejemplo de la clase *Caballo* está sobrecargando un método definido en su superclase. No está sobreescribiéndolo, porque para usar el polimorfismo de inclusión debemos usar el mismo identificador y la misma lista de parámetros que en la superclase. Debemos fijarnos en que el método de la superclase usa dos *bytes*, mientras que el de la subclase usa dos *ints*.

1.5. Interfaces

1.5.1. Objetivos

- Definir el concepto de interfaz.
- Interpretar el código fuente de una aplicación Java donde aparecen interfaces.
- Construir una aplicación Java sencilla, convenientemente especificada, que declare y utilice una interfaz.

Una **interfaz** es una especie de plantilla para la construcción de clases. Normalmente una interfaz se compone de un conjunto de declaraciones de *cabeceras de métodos* (sin implementar, de forma similar a un método abstracto) que especifican un *protocolo de comportamiento* para una o varias clases. Además, una clase puede implementar una o varias interfaces: en ese caso, la clase debe proporcionar la declaración y definición de todos los métodos de cada una de las interfaces o bien declararse como clase *abstract*. Por otro lado, una interfaz puede emplearse también para declarar constantes que luego puedan ser utilizadas por otras clases.

Una *interfaz* puede parecer similar a una clase abstracta, pero existen una serie de diferencias entre una interfaz y una clase abstracta:

- Todos los métodos de una interfaz se declaran implícitamente como abstractos y públicos.
- Una clase abstracta no puede implementar los métodos declarados como abstractos, una interfaz no puede implementar ningún método (ya que todos son abstractos).
- Una interfaz no declara variables de instancia.
- Una clase puede implementar varias interfaces, pero sólo puede tener una clase ascendiente directa.
- Una clase abstracta pertenece a una jerarquía de clases mientras que una interfaz no pertenece a una jerarquía de clases. En consecuencia, clases sin relación de herencia pueden implementar la misma interfaz.

1.5.2. Declaración de una interfaz

La declaración de una interfaz es similar a una clase, aunque emplea la palabra reservada **interface** en lugar de *class* y no incluye ni la declaración de variables de instancia ni la implementación del cuerpo de los métodos (sólo las cabeceras). La sintaxis de declaración de una interfaz es la siguiente:

```
public interface IdentificadorInterfaz {
```

```
        // Cuerpo de la interfaz . . .
}
```

Una interfaz declarada como *public* debe ser definida en un archivo con el mismo nombre de la interfaz y con extensión .java. Las cabeceras de los métodos declarados en el cuerpo de la interfaz se separan entre sí por caracteres de punto y coma y todos son declarados implícitamente como *public* y *abstract* (se pueden omitir). Por su parte, todas las constantes incluidas en una interfaz se declaran implícitamente como *public*, *static* y *final* (también se pueden omitir) y es necesario inicializarlas en la misma sentencia de declaración.

Por ejemplo, la interfaz *Modificacion* declara la cabecera de un único método:

```
/**
* Declaracion de la interfaz Modificacion
*/
public interface Modificacion {
        void incremento(int a);
}
```

que se almacena en el archivo fuente Modificacion.java y que, al compilarse:

$ javac Modificacion.java

genera un archivo Modificacion.class. Al no corresponder a una clase que implementa un método main(), este archivo no puede ejecutarse con el intérprete de Java.

Este es otro ejemplo donde la interfaz Constantes declara dos constantes reales con el siguiente código fuente:

```
/**
* Declaracion de la interfaz Constantes
*/
public interface Constantes {
        double valorMaximo = 10000000.0;
        double valorMinimo = -0.01;
}
```

que se almacena en el archivo fuente Constantes.java y que, al compilarse, genera un archivo Constantes.class.

Otro ejemplo en el que la interfaz Numerico declara una constante real y dos cabeceras de métodos con el siguiente código fuente:

```
/**
* Declaracion de la interfaz Numerico
*/
```

```
public interface Numerico {
        double EPSILON = 0.000001;
        void establecePrecision(float p);
        void estableceMaximo(float m);
}
```

que se almacena en el archivo fuente Numerico.java y que, al compilarse, genera un archivo Numerico.class.

1.5.3. Implementación de una interfaz en una clase

Para declarar una clase que implemente una interfaz es necesario utilizar la palabra reservada **implements** en la cabecera de declaración de la clase. Las cabeceras de los métodos (identificador y número y tipo de parámetros) deben aparecer en la clase tal y como aparecen en la interfaz implementada. Por ejemplo, la clase *Acumulador* implementa la interfaz *Modificacion* y por lo tanto debe declarar un método incremento:

```
/**
* Declaracion de la clase Acumulador
*/
public class Acumulador implements Modificacion {
        private int valor;
        public Acumulador (int i){
                valor = i;
        }
        public int daValor (){
                return valor;
        }
        public void incremento (int a){
                valor += a;
        }
}
```

Esta cabecera con la palabra *implements* implica la obligación de la clase *Acumulador* de definir el método incremento declarado en la interfaz *Modificacion*. El siguiente código muestra un ejemplo de uso de la clase Acumulador.

```
/**
* Demostracion de la clase Acumulador
*/
public class PruebaAcumulador {
```

```
        public static void main (String [ ] args) {
                Acumulador p = new Acumulador(25);
                p.incremento(12);
                System.out.println(p.daValor());
        }
}
```

La compilación y posterior ejecución del código anterior origina la siguiente salida por pantalla:

```
$ javac PruebaAcumulador.java
$ java PruebaAcumulador
```

La clase *Acumulador* tendría también la posibilidad de utilizar directamente las constantes declaradas en la interfaz si las hubiera.

Para poder emplear una constante declarada en una interfaz, las clases que no implementen esa interfaz deben anteponer el identificador de la interfaz al de la constante.

1.5.4. Jerarquía entre interfaces

La jerarquía entre interfaces permite la herencia simple y múltiple. Es decir, tanto la declaración de una clase, como la de una interfaz pueden incluir la implementación de otras interfaces. Los identificadores de las interfaces se separan por comas. Por ejemplo, la interfaz *Una* implementa otras dos interfaces: *Dos* y *Tres*.

```
public interface Una implements Dos, Tres {
        // Cuerpo de la interfaz . . .
}
```

Las clases que implementan la interfaz Una también lo hacen con *Dos* y *Tres*.

Ejemplo de cómo pueden construirse dos interfaces, llamadas *Constantes* y *Variaciones*, y una clase llamada *Factura* que las implementa:

```
// Declaracion de la interfaz Constantes
public interface Constantes {
        double valorMaximo = 10000000.0;
        double valorMinimo = -0.01;
}
// Declaracion de la interfaz Variaciones
public interface Variaciones {
        void asignaValor(double x);
        void rebaja(double t);
```

```
}
// Declaracion de la clase Factura
public class Factura implements Constantes, Variaciones {
        private double totalSinIVA;
        public final static double IVA = 0.16;
        public double sinIVA() {
                return totalSinIVA;
        }
        public double conIVA() {
                return totalSinIVA *(1+IVA);
        }
        public void asignaValor(double x) {
                if (valorMinimo<x) totalSinIVA=x;
        else totalSinIVA=0;
        }
public void rebaja(double t) {
        totalSinIVA *= (1-t/100);
}
public static void main (String [ ] args) {
        factura a = new Factura();
        a.asignaValor(250.0);
        System.out.println("El precio sin IVA es: " + a.sinIVA());
        System.out.println("El precio con IVA es: " + a.conIVA());
        System.out.println("Rebajado durante el mes de mayo un 20%");
        a.rebaja(20);
        System.out.println("Rebajado sin IVA es: " + a.sinIVA());
        System.out.println("Rebajado con IVA es: " + a.conIVA());
        }
}
```

Si una interfaz implementa otra, incluye todas sus constantes y declaraciones de métodos, aunque puede redefinir tanto constantes como métodos.

Nota: Es peligroso modificar una interfaz ya que las clases dependientes dejan de funcionar hasta que éstas implementen los nuevos métodos.

Una clase puede, de manera simultánea, descender de otra clase e implementar una o varias interfaces. En este caso la sección *implements* se coloca a continuación de *extends* en la cabecera de declaración de la clase. Por ejemplo:

```
public class ClaseDescendiente extends ClaseAscendiente implements Interfaz {
        ...
}
```

1.5.5. Utilización de una interfaz como un tipo de dato

Al declarar una interfaz, se declara un nuevo tipo de referencia. Pueden emplearse identificadores de interfaz en cualquier lugar donde se pueda utilizar el identificador de un tipo de dato (o de una clase). El objetivo es garantizar la sustituibilidad por cualquier instancia de una clase que la implemente. Por ejemplo, puede emplearse como tipo de un parámetro de un método:

```
public class Calculos {
        public void asignacion(Variaciones x); {
                ...
        }
}
```

Solamente una instancia de una clase que implemente la interfaz puede asignarse al parámetro cuyo tipo corresponde al identificador de la interfaz. Esta facultad se puede aprovechar dentro la propia interfaz. Por ejemplo:

```
public interface Comparable {
        // La instancia que llama a esMayor (this) y el parametro otra
        // deben ser de la misma clase o de clases que implementen esta interfaz
        // La funcion devuelve 1, 0, -1 si this es mayor, igual o menor que otra
        public int esMayor(Comparable otra);
}
```

En algún caso puede ser útil declarar una interfaz vacía como, por ejemplo:

```
public interface Marcador {
}
```

Esta declaración es totalmente válida ya que no es obligatorio incluir dentro de una interfaz la declaración de una constante o la cabecera de un método. La utilidad de estas interfaces reside en la posibilidad de ser empleadas como tipos de dato para especificar clases sin necesidad de obligar a éstas a implementar algún método en concreto. Una interfaz no es una clase, pero se considera un tipo en Java y puede ser utilizado como tal.

2. Bibliografía

- Material suministrado por CGA.
- https://javadesdecero.es/poo/que-es-una-clase-ejemplos/
- https://www.arquitecturajava.com/java-polimorfismo-herencia-y-simplicidad/
- https://ifgeekthen.everis.com/es/polimorfismo-en-java-programaci%C3%B3n-orientada-objetos