INAP
INSTITUTO NACIONAL DE ADMINISTRACIÓN PÚBLICA

Curso online: **PROGRAMACIÓN ORIENTADA A OBJETOS EN JAVA (TELEFORMACIÓN - ON LINE)**

Tema 3: PROGRAMACIÓN ORIENTADA A OBJETOS

Autores

Jorge Molinero Muñoz

Miguel Sallent Sánchez

ÍNDICE

1. Introducción

Java es un *lenguaje orientado a objetos*, por lo tanto son directamente aplicables todos los conceptos descritos en el tema 1 relativos a la programación orientada a objetos.

A continuación se realizará un pequeño repaso antes de abordar como se materializan dichos conceptos en JAVA.

Un programa orientado a objetos estará compuesto de varios objetos con sus propiedades y las operaciones que se pueden realizar con ellos. Esta idea parte de lo que ocurre en el mundo real, en el cual múltiples entidades desempeñan su labor en función de los estímulos que reciben de otras entidades y de los estados que llegan a alcanzar en cada momento.

Objeto = Estado + Comportamiento

La idea principal es que el objeto conoce bien su cometido, de modo que al utilizar un objeto que cumpla unas especificaciones podemos obviar su funcionamiento interno e integrarlo donde sea necesario. Esto introduce dos conceptos básicos relativos a la POO:

Abstracción y Encapsulación

La **abstracción** es la capacidad del ser humano para entender una situación excluyendo detalles y sólo viéndola a alto nivel. Esta propiedad permite distinguir a un objeto de los demás, observando sus características y comportamientos, pensando en qué es y no en cómo se codificaría en un lenguaje. Con la abstracción se destaca lo importante y se ignora lo irrelevante. Gracias a esta capacidad somos capaces de identificar las propiedades y las operaciones de los objetos que formarán parte de los programas.

Para abordar la resolución de un problema, será necesario abstraerse e identificar aquellas características y comportamientos importantes, para posteriormente modelizarlos en forma de objetos.

Por ejemplo, si vamos a modelizar vehículos, lo primero que debemos hacer es abstraernos e identificar los comportamientos básicos de los vehículos: acelerar, frenar, transportar, etc… Por otro lado, habrá que tener en cuenta sus características básicas, capacidad, autonomía, velocidad, etc.. Una vez que hayamos identificado su comportamiento y sus propiedades, procederemos a implementarlo, y aquí es donde entra la segunda propiedad de la programación orientada a objetos, la encapsulación.

La ***encapsulación*** consiste en combinar los datos y el comportamiento de los objetos ocultando los detalles de su implementación. Será el programador quien determine cómo se comportará el objeto según su estado, siendo algo transparente para quien utilice dicho objeto.

Lo importante es que el objeto ofrezca una imagen robusta al exterior, de modo que realice su función correctamente, independientemente de cómo haga dicha tarea.

Extrapolemos esta idea a un ejemplo de la vida real: cuando aprendemos a conducir un coche somos capaces de conducir prácticamente cualquiera. ¿Por qué? Porque lo que aprendemos es cómo hacer que ese coche realice su función: para que acelere, pisamos el pedal del acelerador; para que frene, el pedal del freno, y no nos importa si acelera con más o menos válvulas o que el freno sea de disco o de tambor. Así, si pasamos de tener un Seiscientos a un BMW no tendremos que aprender a conducir de nuevo.

Por otra parte, a nosotros no nos importa cómo acelere el coche mientras que cuando aceleremos obtengamos la respuesta esperada.

Otra de las razones para defender la encapsulación está en el hecho de que a través de ella es más fácil controlar los errores, será el objeto que guarda los datos el que determinará si existen valores inconsistentes.

Como ejemplo supongamos que queremos tener un objeto que represente las fechas del calendario. Si permitimos que cualquier otro objeto pueda fijar una fecha, sería posible tener como válido el 30 de febrero de 1998, mientras que, si no permitimos acceso directo, nuestro objeto de las fechas podría evitar que tal cosa ocurriera.

La clave para lograr la encapsulación está en que un objeto de una clase nunca pueda manipular directamente los datos internos de otro objeto perteneciente a otra clase, sino que, si desea variar alguno de ellos, tenga que "pedírselo" a quien los guarda. Así, toda la comunicación debe realizarse mediante mensajes.

Bien, ahora que hemos explicado las ventajas de la *POO*, pasaremos a definir los términos fundamentales para trabajar con ella.

2. Clases

2.1 Introducción

Es el término más importante en programación orientada a objetos, una clase es similar a la plantilla o el molde a partir del cual se construyen los objetos.

La clase define una serie de propiedades comunes a todos los objetos que pertenecen a ella (atributos) **y las operaciones que se pueden realizar con dichos objetos** (métodos). Normalmente las clases se identifican en el lenguaje natural por venir precedidas de un artículo indeterminado, como "un coche", "una casa", etc. o por un artículo determinado plural, como "los coches", "las casas", etc.

Como ya hemos comentado, los atributos de una clase son cada una de las propiedades que presentan los objetos que pertenecen a esa clase, de modo que dichos objetos (o instancias) se distinguen por los valores de sus atributos.

2.2 Declaración de una Clase

Para declarar una clase se deberá escribir la palabra reservada *class* seguido del nombre de la clase. Después, entre llaves, aparecerá el cuerpo la clase.

A continuación se muestra un ejemplo de la estructura de una clase mínima, en la cual sólo aparecen los elementos obligatorios.

```
class MiClase {

   // declaraciones de campos

  // constructores y métodos

}
```

Norma de convención:
El nombre de la clase debe comenzar por letra mayúscula, aunque el compilador no lo impone.

El cuerpo de la clase se encuentra delimitado por llaves *"{ }"*. Y es ahí donde se declararían los atributos, los métodos y los constructores de la clase.

La sintaxis para declarar una clase es la siguiente:

```
modificadoresDeClase class NombreClase extends ClasePadre implements InterfazClase
{
        // declaraciones de campos
          …
        // constructores y métodos
          …
      }
```

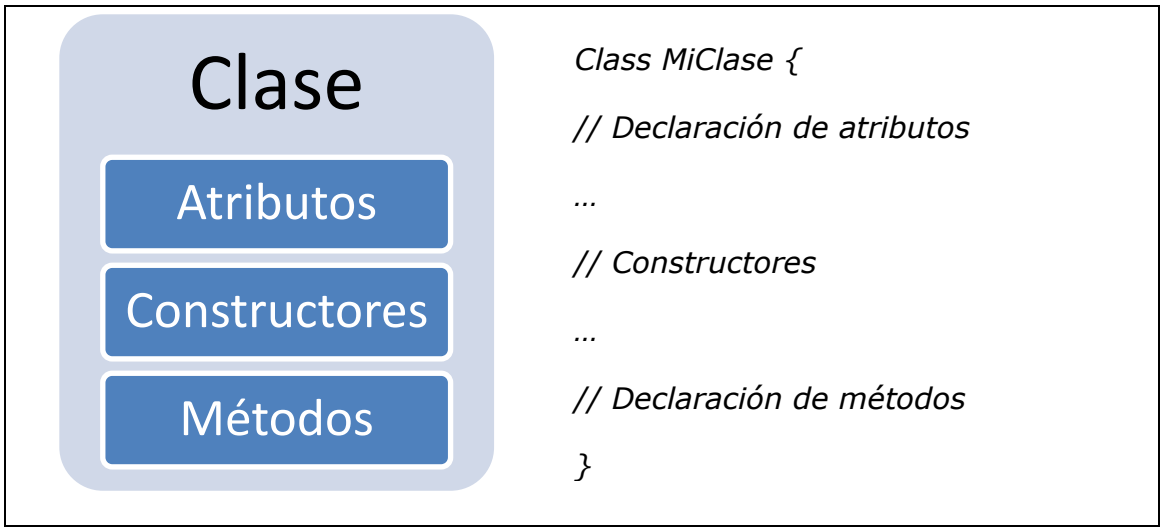
Los elementos que componen la declaración de una clase son:

- Los modificadores de clase, son opcionales, los más comunes son *public* y *private,* que definen la visibilidad de la clase desde otras clases. Después se describirá más detalladamente.
- La palabra reservada *class*
- Nombre de la clase.
- *extends ClasePadre,* que significa que nuestra clase es una clase hija o subclase de la *ClasePadre*. Se verá más detalladamente al hablar de la herencia.
- *implements InterfazClase*, que indica que la clase implementa el interfaz *InferfazClase.* Se hablará más detalladamente al tratar los interfaces.
- El cuerpo de la clase

A su vez el cuerpo de la clase estará compuesto por:

- Atributos.
- Constructores
- Métodos.

En la siguiente figura aparece una representación gráfica de las partes que conforman una clase:



2.3 Declaración de Variables

Existen distintos tipos de variables según el ámbito en el cual son declaradas:

- **Atributos**, se declaran dentro de la clase y fuera de cualquier método o constructor. Permiten guardar la información que almacena la clase.

- **Variables locales**, aquellas que se declaran en un bloque de código delimitado por llaves "{ }" o en el interior de un método, no se puede acceder a ellas desde fuera del bloque de código o del método.

- **Parámetros**, son las variables que se definen al declarar los métodos, permiten transmitir información a los métodos.

A continuación se presenta un ejemplo de la clase Bicicleta donde se pueden observar distintos tipos de variables:

```
class Bicicleta {

    int cadencia = 0;
    int velocidad = 0;
    int marcha = 1;

    void cambiaCadencia(int nuevoValor) {
        cadencia = nuevoValor;
    }

    void cambiaMarcha(int nuevoValor) {
```

```
        marcha = nuevoValor;
    }

    void aumentaVelocidad(int incremento) {
        velocidad = velocidad + incremento;
    }

    void frena(int decremento) {
        velocidad = velocidad - decremento;
    }

    void imprimeEstado() {
        String cadenaTexto =  "cadencia:" + cadencia +
                              " velocidad:" + velocidad +
                             " marcha:" + marcha;
        System.out.println(cadenaTexto);
    }
}
```

Los tipos de variables que aparecen son:

- Atributos: *cadencia, velocidad y marcha.* Todos estos son de tipo básico.
- Parámetros: *nuevoValor, incremento y decremento*.
- Variables locales: cadenaTexto. Que se encuentra dentro del método imprimeEstado.

Para declarar un atributo, tenemos que darle un nombre y anteponerle el tipo del atributo (que puede ser un tipo primitivo o una clase).

modificadores NombreClase nombreAtributo;

Por convención, los nombres de los atributos comienzan por minúscula.

Por lo tanto la declaración de atributos, está compuesta por los siguientes elementos, en este orden:

- Modificadores: El atributo, al igual que la clase, puede llevar distintos tipos de *modificadores*, para establecer la visibilidad de los atributos (qué objetos tienen acceso directo al atributo y cuáles no) entre otras características.
- Nombre de la clase o tipo básico del atributo.

- Nombre del atributo.

2.4 Declaración de Métodos

Para declarar un método, tenemos que darle un nombre y anteponerle el nombre de la clase (o tipo básico) del resultado que devuelve dicho método. Si el método no devuelve nada antepondremos la palabra reservada *void.* Detrás del nombre, encontraremos entre paréntesis los parámetros del método.

Un ejemplo básico de declaración de un método es el siguiente:

```
void aumentaVelocidad(int incremento) {

    velocidad = velocidad + incremento;
}
```

Donde se observa que el método no devuelve nada (*void), el nombre del método es aumentaVelocidad,* y el único parámetro que tiene es *incremento*. En el cuerpo del método se observa que aumenta la velocidad sumándole el incremento recibido por parámetro.

A continuación se presenta la sintaxis para declarar un método:

```
modificadores  tipoResultado nombreMetodo( tipo1 arg1,...,tipoN  argN){
            // implementación o cuerpo del método
           …
           }
```

Los elementos que lo componen son:

- Modificadores. Al igual que ocurría con los atributos, los métodos pueden incluir uno o varios modificadores (entre ellos los de acceso).
- Clase o tipo del resultado que devuelve el método.
- Nombre del método.
- Los parámetros que recibe el método entre paréntesis.
- El cuerpo del método entre llaves.

Por convención, los nombres de método empiezan con minúsculas

2.5 Declaración de Constructores

El **constructor** es un método existente en toda clase y tiene como finalidad crear instancias de dicha clase y dar valor a los atributos. Es decir, con la simple declaración de una variable objeto no se está creando el objeto propiamente, ni se está reservando memoria para él. Esto solamente ocurre al invocar al operador *new* seguido del constructor.

El constructor debe obligatoriamente tener el mismo nombre que la clase, respetando las mayúsculas y minúsculas. Puesto que por convención hemos establecido que las clases empiezan por mayúscula, el constructor empezará por mayúscula, a diferencia del resto de métodos de la clase.

El constructor es un método que nunca devuelve nada, a pesar de lo cual no se pone la palabra reservada void, lo cual supone otra diferencia respecto a los otros métodos.

Java define un constructor por defecto para cada clase. Este constructor no tiene parámetros y se encarga de inicializar los atributos con los valores por defecto que corresponden a su tipo (a cero los números, a null los objetos, a false los booleanos y al carácter nulo los caracteres).

Sin embargo, si deseamos crear un objeto e inicializar sus atributos con unos valores distintos, se podrá escribir un constructor que inicialice los atributos a nuestro gusto. Además podemos escribir varios constructores para una misma clase, siempre y cuando se diferencien en sus parámetros, ofreciendo varias formas distintas de crear el objeto en función de los atributos que deseemos inicializar al crearlo.

Ejemplo: si vamos a dar de alta un empleado, lo normal es que conozcamos su nombre, por lo que no querremos que al atributo "*nombre*" (al ser de tipo String) se le asigne el valor *null*. En este caso crearemos un constructor que tome como argumento el nombre y lo asigne al atributo correspondiente.

Asimismo, puede suceder que por la lógica de las clases, ciertos atributos no puedan tener los valores que asigna el lenguaje por defecto.

Ejemplo: cuando damos de alta un cliente es porque nos ha comprado algún artículo, por lo que el atributo "comprasEfectuadas" nunca tendrá el valor 0.

En estos casos, conviene declarar e implementar constructores "a medida", es decir, que inicialicen los atributos según nuestras necesidades. Para ello, el constructor puede recibir por parámetro los valores de los atributos que queramos inicializar. En cualquier caso, siempre con la restricción de que el nombre del constructor definido sea el mismo que el de la clase a la que pertenece.

Ejemplo: tenemos una clase "*Cliente*" que posee entre otros atributos el "*nombreCliente*" y el "*comprasEfectuadas*". Es frecuente dar de alta a un cliente introduciendo su nombre, por lo que decidimos crear un constructor que lo permita:

```
class Cliente {
   String nombreCliente;
   int comprasEfectuadas;

   Cliente (String nombre) {
        nombreCliente = nombre;
        comprasEfectuadas = 1;
    }
}
```

De igual forma se puede sobrescribir el constructor por defecto de forma que inicialice los valores de los atributos que deseemos.

Ejemplo: queremos que el "*comprasEfectuadas*" valga al menos uno.

```
class Cliente {
     String nombreCliente;
     int comprasEfectuadas;

    Cliente ( ) {
        nombreCliente = null;
        comprasEfectuadas = 1;
     }
```

```
}
```

Ejemplo: además de los que ya teníamos, añadimos a la clase Cliente un atributo "*numeroCuenta*" de tipo entero. Tenemos también la clase Cuenta, la cual presenta un método "*obtenerNuevoNumero*( )" que devuelve un nuevo número de cuenta, que no esté asignado a otro cliente. Puede darse el caso de necesitar crear un cliente sólo por su nombre, o por su nombre y un número de cuenta que le hayamos previamente asignado. En estas situaciones tendremos al menos dos constructores:

```
class Cliente {
   String nombreCliente;
   int comprasEfectuadas;
   int numeroCuenta

   Cliente (String nombre) {
        this(nombre, Cuenta.obtenerNuevoNumero() );
    }

   Cliente (String nombre, int numCuenta) {
        nombreCliente = nombre;
        numeroCuenta = numCuenta;
        comprasEfectuadas = 1;
    }
}
```

Observemos la aparición de la palabra reservada "*this*"; aunque insistiremos en ella más adelante, por ahora conviene saber que (entre otras funciones) actúa como sustituto del nombre del constructor de una clase, y que por tanto la línea

this(nombre, Cuenta.obtenerNuevoNumero( ));

equivale a

Cliente(nombre, Cuenta.obtenerNuevoNumero( ));

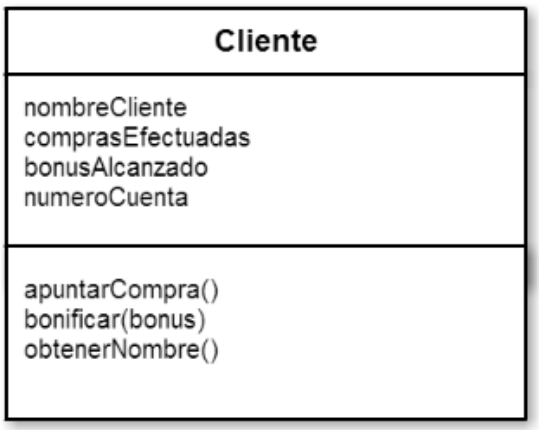
Es decir, mediante el uso de la palabra reservada *this* podemos invocar a un constructor desde otro constructor, evitando duplicar código.

En caso de que alguno o varios de los atributos tengan un valor fijo para todas las instancias de la clase, se considera buena práctica inicializarlo fuera de todos los constructores, es decir, cuando lo declaremos.

Ejemplo: queremos que el atributo "*comprasEfectuadas*" valga siempre uno al crear cualquier instancia de "*Cliente*".

```
class Cliente  {

    ...

    int comprasEfectuadas = 1;

}
```

Para resumir, se muestra una figura que muestra cómo quedaría la clase "Cliente", con todas las apreciaciones efectuadas hasta ahora y tres métodos más, que permiten apuntar una nueva compra al cliente, darle una bonificación para la próxima compra y obtener su nombre respectivamente:




```
class Cliente {

   // atributos
```

```
    String nombreCliente;
    int numeroCuenta;
    int comprasEfectuadas = 1;
    int bonusAcumulado;
    // constructores
    Cliente (String nombre) {
        this(nombre, Cuenta.obtenerNuevoNumero() );
    }
    Cliente (String nombre, int numCuenta) {
        nombreCliente = nombre;
        numeroCuenta = numCuenta;
     }
    // métodos
    public void apuntarCompra(){
         comprasEfectuadas = comprasEfectuadas + 1;
    }
    public void bonificar(int bonus){
         bonusAcumulado = bonusAcumulado + bonus;
    }
    public String obtenerNombre(){
         return nombreCliente;
    }
} // class
```

3. Objetos o Instancias

3.1 Introducción

Un objeto o instancia es un caso concreto de una clase. Se caracteriza por los valores de las propiedades que define dicha clase. De modo que puede haber múltiples objetos de la misma clase que se diferencian por la información que contienen en sus atributos. Así, tendremos que para el atributo "*color*" diremos "mi coche es rojo" y "tu coche es gris".

Estado de un objeto: es el valor de los atributos de un objeto en un momento dado, podemos pensar en él, como una fotografía del objeto en un instante de determinado, en el cual los atributos del objeto tendrán un valor concreto. El estado de un objeto puede afectar a su comportamiento. De modo que un objeto se comporte de una forma u otra según su estado. Por ejemplo, si un pedido está pagado no podrá aceptar nuevos artículos, y si un pedido está vacío no debe permitir que le borren artículos.

Por norma general, un programa en Java opera con un conjunto de objetos de distintas clases que interactúan entre ellos.

Por ejemplo, en una aplicación de contabilidad se operaría con un conjunto de objetos de diversas clases: Activo (que representa los bienes cuantificables) y Pasivo (que representa las deudas u obligaciones). Donde cada uno de estos objetos tendrá distintos atributos: valor, nombre, tipo de activo o tipo de pasivo, etc…

La aplicación de contabilidad debería permitir al usuario crear nuevos objetos activo y pasivo, y modificar los valores asignados a sus atributos. Por ejemplo un objeto activo para la casa, otro para el coche… y a la par permitiría crear nuevos objetos pasivos, la hipoteca, el préstamo personal, etc…

3.2 Creación de una instancia u objeto

Para crear un objeto de una determinada clase son necesarias tres operaciones:

1. **Declarar la variable que almacenará el objeto**: para ello se escribe el nombre de la clase del objeto, seguido del nombre de la variable:

Ejemplo:

Activo miPiso;

Pasivo miHipoteca;

2. Crear o instanciar el objeto: mediante una llamada al constructor de la clase a la que pertenece precedida del operador *new*.

Ejemplo:

```
miPiso = new Activo();

miHipoteca = new Pasivo();
```

Ambos pasos se pueden realizar en uno solo:

Ejemplo:

```
Activo miCoche = new Activo( ) ;
```

3. Inicialización del objeto: Una vez se ha creado o instanciado el objeto, se procede a la ejecución del constructor, que realiza diversas operaciones como puede ser asignar valores a sus atributos.

Por ejemplo, dada la siguiente clase *Activo*:

```
public class Activo{

        // Declaracion de atributos

        int valor;

        String nombre;

        String tipo;


        // Constructor de la clase

        public Activo(String nombreActivo, String tipoActivo){

                valor = 0;

                nombre = nombreActivo;

                tipo = tipoActivo;

        }

}
```

Si ejecutamos la siguiente sentencia:

```
Activo miHipoteca = new Activo("miHipoteca","hipoteca");
```

Se creará una variable denominada *miHipoteca* de tipo *Activo*, cuyos atributos tendrán los siguientes valores:

- valor = 0
- nombre = "miHipoteca"
- tipo = "hipoteca"

3.3 Uso de Objetos

Una vez se ha creado el objeto, describiremos cómo podemos operar con dicho objeto, accediendo al valor de sus atributos, modificando dicho valor y ejecutando sus métodos.

Para referenciar a los atributos o métodos de una instancia se utiliza la notación:

nombre_instancia.nombre_atributo

nombre_instancia.nombre_método(parámetro1,...,parámetroN)

Sin embargo, sólo se podrá acceder a los atributos y métodos, cuando estos sean visibles, lo que estará estrechamente ligado a los modificadores de acceso que los acompañen.

Por ejemplo, una vez que se ha creado la hipoteca, podemos modificar el valor asignado a ella, cambiar el tipo de interés, amortizar una cantidad, etc…

```
Activo miHipoteca = new Activo();

// Establecemos el valor de la hipoteca a 120.000 €

miHipoteca.valor = 120000;

// Mostramos por pantalla el valor de la hipoteca

System.out.println("El valor de la hipoteca es "+miHipoteca.valor);

// Amortizamos 6000 euros

miHipoteca.amortizar(6000);

// Mostramos el nuevo valor de la hipoteca

System.out.println("El valor de la hipoteca es "+miHipoteca.valor);
```

3.4 This. Ámbito de una variable.

This es una palabra reservada que se utiliza para referirse al objeto actual, se utiliza en métodos y en constructores, y permite acceder a cualquier miembro del objeto actual.

Se emplea en tres situaciones:

- Para llamar a un método pasándole como parámetro el objeto actual.

  Ejemplo:

```
class Cliente {
    int codigoCliente;

    public void apuntarCompra (Pedido p){
        …
   }
} // class

class Pedido {
   Cliente cliente;
   int codigoPedido;

   public void asignarAlCliente(){
       cliente.apuntarCompra(this);
  }
} // class
```

- Para la resolución de una variable dentro de un ámbito: cuando se opera con métodos que tienen parámetros con el mismo nombre que los atributos de la clase, permiten diferenciar las variables.

  Ejemplo:

```
class Cliente {
   // atributos
   String direccion;
   …

   // métodos
   public void cambiarDireccion(String direccion){
      this.direccion = direccion;
   }
} // class
```

- Para invocar, dentro de un constructor, a otro constructor de la clase.

  Ejemplo:

```
class Cliente {
    // atributos
    String nombreCliente;
    int numeroCuenta;
     // constructores
    Cliente (String nombre) {
```

```
        this(nombre, Cuenta.obtenerNuevoNumero() );
    }
    Cliente (String nombre, int numCuenta) {
        nombreCliente = nombre;
        numeroCuenta = numCuenta;
     }
    …
  }
```

3.5 Modificadores de atributos y métodos.

3.5.1 Introducción

Los **modificadores** son palabras reservadas que se anteponen al nombre de una clase, método o atributo con el fin de imponerles un comportamiento o unas cualidades de las que carecen por defecto normalmente.

3.5.2 Modificadores de Acceso

Determinan la visibilidad del atributo o método de un objeto, es decir indican si es posible referenciar los atributos o métodos desde otro objeto. Esto se especificará a través del modificador de acceso que antecede a la declaración del método o del atributo.

Existen cuatro modificadores de acceso para un atributo:

- **public**: Un atributo o método declarado público permite que desde cualquier otro objeto de cualquier clase puedan referenciarlo directamente. Esto se hace mediante las sentencias:

```
// Para acceder al atributo NombreAtributo del objeto instancia
instancia.nombreAtributo;
// Para acceder al método nombreMetodo del objeto instancia
instancia.nombreMetodo(parametros);
```

Por ejemplo, si tenemos la clase "*Cliente*" con el campo *comprasEfectuadas* público, y desde otra clase, creamos una instancia de tipo Cliente a la que llamamos "*miCliente*", podemos incrementar el número de compras que ha efectuado sin más que escribir:

miCliente.comprasEfectuadas ++ ;

- **private**: Un atributo declarador privado no permite que se acceda a él directamente desde otra clase distinta. Se deberá recurrir a algún método que nos proporcione el valor del atributo. Si el método es privado, desde ninguna otra clase podrá invocársele (estos métodos suelen definirse para realizar labores auxiliares para otros métodos de la clase que sí sean accesibles desde fuera de ella).

- **protected**: Los atributos o métodos declarados protected son accesibles directamente desde la clase en que se definieron, las subclases de ésta (ver el apartado de Herencia) y las clases que se encuentran dentro de su mismo paquete (ver apartado de Paquetes).

- **default:** Es el modificador que se aplica por defecto, es muy semejante a *protected*, los atributos o métodos son accesibles directamente desde la clase en que se definieron y las clases que se encuentran dentro de su mismo paquete (ver apartado de Paquetes), pero no desde las subclases de otros paquetes.

¿Qué tipo de modificador es más conveniente?

Aunque en algunos casos puede resultar si no conveniente, sí admisible, el declarar un atributo como público, un programa que pretenda ser orientado a objetos debería declarar todos los atributos de las clases como privados.

La razón es que permite que los objetos se comporten realmente como cajas negras (encapsulación y reusabilidad), escondiendo su implementación al exterior, con las ventajas que se derivan y que se mencionaron en la introducción a la POO. De este modo, cualquier objeto que pretenda acceder al valor de los atributos de otro, bien para consultarlo bien para modificarlo, se verá obligado a invocar a los métodos que el objeto defina para ello.

En la bibliografía se habla de métodos "*accesor*" y métodos "*mutator*" (también conocidos como métodos "*getter*" y "*setter*"). Estos métodos se utilizan para consultar (accessor) o cambiar (mutator) los atributos. Si se desea que algún atributo no pueda ser visto desde fuera, lo declararemos privado y no implementaremos métodos accessor, ni mutator (por ejemplo, para aquellas variables internas que necesitamos controlar para deducir el valor de otros atributos).

En cuanto a los métodos, el modificador más conveniente dependerá de la situación. Según la visibilidad que se desee dar al método, se declarará como *public* (para poder invocarlo desde cualquier clase), *protected* (si queremos que sólo las clases hijas o las que estén en el mismo paquete puedan invocarlo) o *private* (si el método realiza

operaciones intermedias pero que no interesa que puedan ser invocadas desde el exterior).

Un ejemplo de esto sería un objeto que supervisa un sistema y que dispara una alarma si se cumplen ciertas condiciones. No nos interesará que otras clases creadas fuera de ésta puedan hacer saltar la alarma.

A continuación se presenta una tabla que resume la visibilidad de los modificadores de acceso:

Modificadores de Acceso				
	Acceso desde la misma clase	Acceso desde otra clase del mismo paquete	Acceso desde una subclase de otro paquete	Acceso desde una clase de otro paquete
public	X	X	X	X
protected	X	X	X	
default	X	X		
private	X			

3.5.3 Otros modificadores

Los modificadores de acceso son los más comunes, pero existen otro tipo de modificadores que se pueden aplicar a los atributos y a los métodos.

Para atributos

- **static**: un atributo declarado como *static* se asocia con la clase, de modo que toma un mismo valor para todos los objetos de dicha clase; es decir, si se cambia el valor del atributo, automáticamente todas las instancias de dicha clase adquirirán ese valor. Una variable de este tipo se denomina variable de clase, y funcionan de manera similar a las variables globales o compartidas en el ámbito de la clase. Hay que tener mucho cuidado con ellas.

- **final:** son constantes, es decir, variables que no permiten modificar su valor, por lo que deben ser inicializadas antes de ser referenciadas.

Para métodos

- **static**: identifica a un método de clase. Estos métodos se pueden ejecutar directamente, sin crear una instancia de la clase. Por tanto, se invocan anteponiendo el nombre de la clase, no el de la instancia de la clase.

  Ejemplo: el método *pow*(base, exponente) de la clase *Math*, permite ejecutar potencias, es estático. Por lo tanto, para invocarlo sólo hay que poner:

  ```
  y=Math.pow(x,a);
  ```

  Como se observa, es posible llamar al método directamente, sin necesidad de instanciar la clase poniendo new Math().

- **final**: indica que el método no puede ser sobrescrito por las subclases, como se verá en el apartado de Herencia.

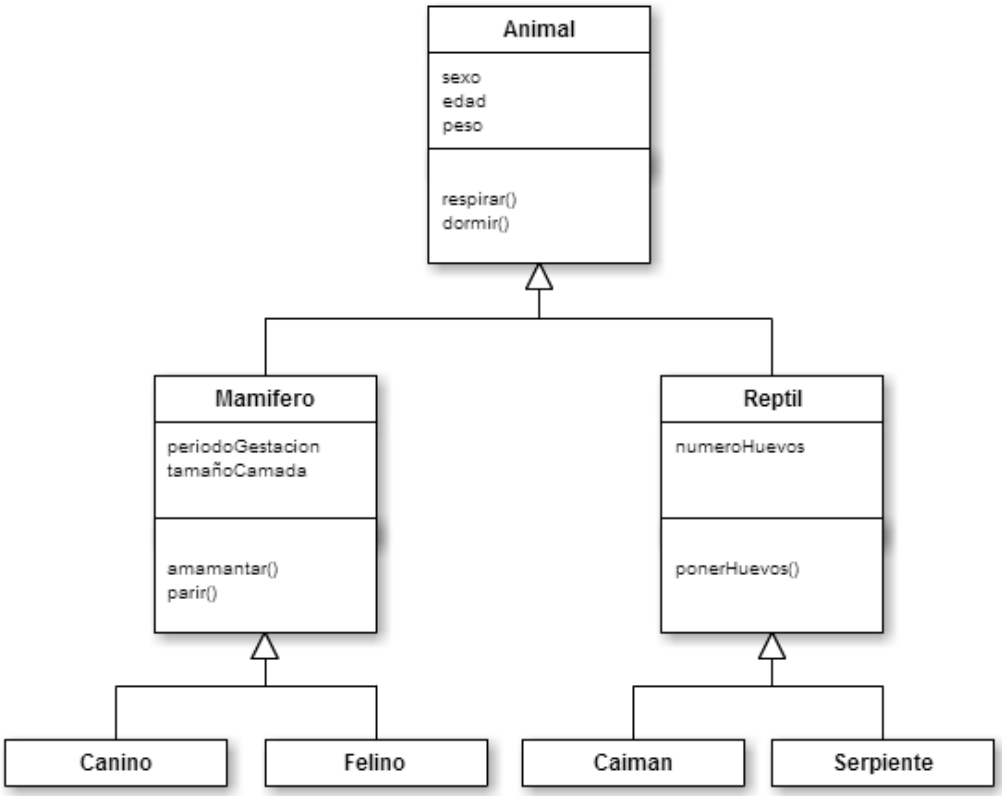
- **abstract**: indica que el método no está implementado, tan sólo definido, y que se deberá implementar en alguna clase hija, se verá más detalladamente en el apartado de Herencia.

- **native**: identifica a métodos escritos en un lenguaje que no es Java.

4. Herencia e Interfaces

4.1 Introducción

Cuando observamos el mundo real vemos cómo los objetos se relacionan unos con otros de manera jerárquica, compartiendo ciertas características de estructura y comportamiento pero añadiendo especificidad respecto a sus semejantes. Así, tenemos la clase *"animales"*, que a su vez engloba *"mamíferos"*, *"reptiles"*, *"aves"*, etc., las cuales se especializan en nuevos grupos. Cada clase define unas propiedades y un comportamiento, y aquellas otras clases que manifiestan pertenecer a la primera los compartirán, pudiendo añadir sus cualidades particulares. Estas clases son conocidas como **subclases**, **clases hijas** o **clases derivadas**, frente a la superclase o **clase padre**.



4.2 Herencia

Consiste en establecer una jerarquía de clases de modo que cada clase hija hereda los atributos y los métodos de la clase padre, pudiendo añadir métodos y atributos propios.

Al heredar la subclase se comporta por defecto como la clase padre, es posible invocar a los métodos de la clase padre aunque no estén definidos en la clase hija.

En el ejemplo anterior de los animales, si se declara una instancia de la clase *"mamífero"* y se llama al método *"respirar"*, en realidad se está invocando a la implementación del método definido en la clase *"animal"*.

Inmediatamente nos surge una pregunta: ¿qué ocurre cuando una clase hija especializa un método, es decir, decide que no quiere hacer las cosas exactamente cómo define su padre?

Java proporciona la respuesta a través de la sobrescritura de un método, que consiste en permitir que una clase hija sobrescriba cualquier método de la clase padre para adaptarlo a sus necesidades. Así, cuando se invoque a un método de un objeto, el compilador lo buscará en la subclase a la que pertenezca, si lo encuentra lo ejecuta y en caso contrario, asciende en la jerarquía de clases hasta encontrarlo.

En el ejemplo anterior, todos los animales respiran, para lo cual toman oxígeno del exterior, por lo que podemos declarar el método respirar en la clase animal. No obstante, los mamíferos respiran a través de pulmones mientras que los peces utilizan branquias. Para particularizar estos comportamientos podemos sobrescribir el método en cada una de las subclases mamífero y pez, sabiendo que el compilador ejecutará el que corresponda dependiendo de la clase a la que pertenezca el objeto.

Asimismo, y a diferencia de lo que sucede en otros lenguajes, como C++, Java no permite la herencia múltiple, es decir, una subclase sólo puede heredar implementación de una clase padre.

En Java la herencia de implementación se realiza incluyendo la palabra clave ***extends*** al final de la declaración de la clase hija.

```
class MiClaseHija extends MiClasePadre {

    //atributos de la clase hija

    //métodos propios de la clase hija

    //métodos redefinidos

}
```

Ejemplo: Dada una clase *Empleado*, se quiere crear una clase hija denominada Directivo

```
class Directivo extends Empleado {
    ...
}
```

En Java todas las clases heredan por defecto de una superclase denominada Object, de la que obtienen una serie de métodos y atributos comunes a todos los objetos. Ésta clase se encuentra definida en el paquete java.lang.

La palabra reservada *super*

Se utiliza desde una clase hija para referenciar a la clase padre, ya sea a sus atributos, métodos o constructores. Permite:

- Referenciar desde una clase hija a un atributo que se ha declarado en la clase padre. La sintaxis que se emplea es la siguiente:

  ```
  super.atributo;
  ```

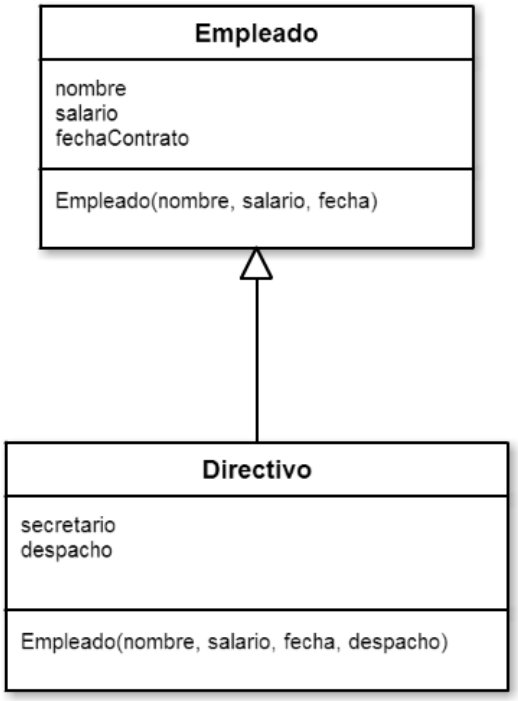
- Al sobreescribir un método del padre, invocar al método original desde el método que lo sobreescribe

  ```
  public void metodo(Param parametros){
    super.metodo(parámetros);
    …
  }
  ```

- Invocar desde el constructor de una subclase al constructor de la clase padre, el cual inicializará los atributos declarados en la clase padre. En caso de usarse, debe ser obligatoriamente la primera sentencia en el constructor de la subclase.

  Si el desarrollador no realiza la llamada de forma explícita al constructor de la clase padre, Java insertará automáticamente una llamada al constructor de la clase padre sin argumentos, y en el caso de no existir dicho constructor en la clase padre, se producirá un error de compilación.

  Ejemplo: implementación de las clases *Empleado* y *Directivo*.




```
public class Empleado {
    // Declaración de atributos de la clase
    private String nombre;
    private int salario;
    private Fecha fechaContrato;
    // Constructor
    public Empleado(String nombre,  int salario, Fecha fecha) {
        this.nombre = nombre;
        this.salario = salario;
        this.fechaContrato = fecha;
    } //contructor
    …
```

```
} // class Empleado

public class Directivo extends Empleado {

    // atributos

    private String secretario;

    private String despacho;

    …

    // constructor

    public Directivo (String nombre, int salario, Fecha fecha, String despacho){

        super(nombre,salario,fecha);

        secretario = "";

        this.despacho = despacho;

    }

    …

}
```

4.2.1 Clases finales y métodos finales

En ocasiones puede interesarnos que una clase no pueda tener descendencia, es decir, que nadie pueda extender una clase creada por nosotros para crear otra (normalmente por motivos de seguridad). En estos casos es necesario declarar nuestra clase como *"clase final"*, para lo cual se antepone el modificador ***final*** en la declaración.

```
final class MiClaseSinHijos {

    // métodos de la clase

    // atributos de la clase

} // class
```

Análogamente, si queremos evitar que una clase hija pueda redefinir un método implementado por la clase padre, tenemos que añadir el modificador final en la declaración del método. Por defecto, todos los métodos de una clase final son finales.

```
modificadorAcceso final tipoResultado miMetodoFinal(parametros) {
```

```
    ...
}
```

Ejemplo: tenemos una clase que controla el estado de un sistema y dispara una alarma bajo ciertas condiciones. No nos interesará que nadie pueda extender de nuestra clase, ya que podría sobreescribir métodos sensibles alterando las condiciones que hacen saltar la alarma.

Como última consideración, hablaremos de si conviene o no que los atributos presentes en una clase padre se declaren como *protected* o como *private*. Al declarar como *protected* estamos permitiendo que desde cualquier clase hija se acceda directamente al atributo. A priori esto parece buena idea, puesto que las clases hijas lo son por compartir métodos y atributos de una superclase. Ahora bien; si otro programador crea una clase hija de una creada por nosotros y accede directamente a nuestros atributos, cada cambio que se decida realizar sobre la representación de los mismos deberá serle comunicado, pues de otra forma su programa no funcionará.

Esto nos condiciona enormemente, y por ello se recomienda que, aun en una jerarquía de herencia, el acceso a todos los atributos del padre se lleve a cabo a través de métodos accesor y mutator. Hay otra situación aún peor. Supongamos que el valor que toma un atributo está en un determinado rango y no podemos permitir que tome valores fuera de él (por ejemplo, un atributo Fecha no debería tomar nunca el valor 30-2-98). Si dejamos que las posibles subclases que se crean a partir de la nuestra accedan directamente a los atributos de ella no podremos controlar la consistencia.

No obstante, si se está completamente seguro de que la representación de los atributos no va a cambiar y de que no hay problemas respecto a los valores que puedan tomar, podemos declararlos como *protected*.

4.2.2 Clases Abstractas

Una clase abstracta en Java es aquélla que se declara como tal, anteponiendo en la declaración la palabra reservada ***abstract***, y que posee al menos un método abstracto. Un método abstracto es un método que no se implementa en la clase que lo declara, obligando a que sean las clases hijas quienes lo hagan. Para ello la declaración del método presenta el modificador ***abstract***. La sintaxis es como sigue:

```
modificador_de_acceso abstract class MiClaseAbstracta {

    // atributos

    // métodos implementados

    //método/-s abstractos

    modificador_acceso abstract tipo_resultado método1(...);

    …

} //class
```

Ejemplo: la clase *Mensaje*

```
public abstract class Mensaje{
  //  atributos
  private String remitente;
  // constructor
  Mensaje (String origen){
     Remitente = origen;
   }
  // métodos
  public String obtenerRemitente(){
     return remitente;
  }
  public abstract void reproducir();
} // class
```

Para indicar que una clase hereda de una clase abstracta, se utilizará la misma sintaxis, es decir, la emplea palabra clave ***extends*** en la declaración.

Ejemplo:

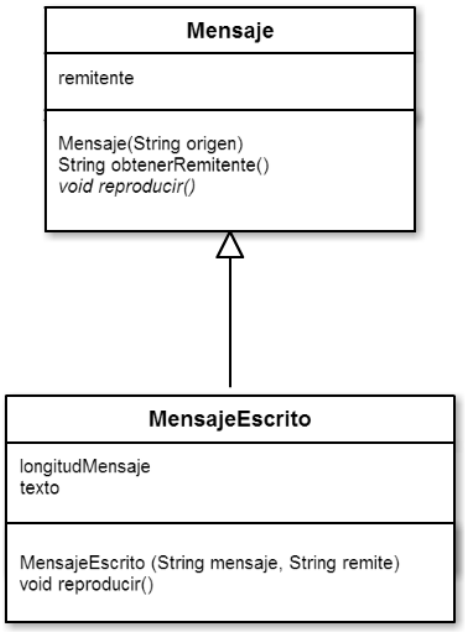
```
public class MensajeEscrito extends Mensaje{
     …
}
```

Una clase que hereda de una clase abstracta está obligada a implementar los métodos abstractos que se hayan definido en la clase padre. Si no se implementan, la clase hija también debe declararse como abstract.

No se pueden crear instancias de las clases abstractas directamente con el operador *new*, dado que su implementación no está completamente definida.

Las clases abstractas sirven para cuando tenemos una jerarquía en la que varios hijos comparten cierto comportamiento, que será implementado en la clase padre, pero también tienen cierto comportamiento que los diferencia y para el cual no queremos dar una implementación por defecto, pero sí queremos forzar a que todos los hijos lo implementen.

Ejemplo: implementamos la clase *MensajeEscrito*, que hereda de *Mensaje.*




```
public class MensajeEscrito extends Mensaje{
  // atributos
  int longitudMensaje;
  String texto;
  // constructor
  MensajeEscrito (String mensaje, String remite){
     super(remite); // Llamamos al constructor del padre
     texto = mensaje;
  }
  public void reproducir(){
     System.out.prinltn(texto);
```

```
  }

} // class
```

4.3 Interfaces

Un interfaz es un elemento, semejante a una clase, que establece una serie de funcionalidades que después son realizadas por las clases que lo implementen. Tradicionalmente los interfaces contenían:

- Constantes
- La declaración de los métodos, sin cuerpo.

De este modo el interfaz define un comportamiento, indica qué funciones deben desempeñar las clases que pretendan implementarlo sin entrar en detalles de cómo se realizará dicha función, pues será cada clase particular quien decida cómo quiere hacer las cosas.

Una clase puede implementar múltiples interfaces (ésta es una importante diferencia respecto a la herencia), puesto que nada impide que un objeto pueda comportarse de varias maneras. Así por ejemplo, que una persona sea informático no le impide ser jugador de baloncesto.

Para la declaración de un interfaz se utiliza la palabra reservada *interface*.

```
interface NombreInterfaz {

    // constantes

    // declaracion de métodos

    public tipoResultado1 metodo1(parametros);

    ...

    public tipoResultadoN metodoN(parametros);

} // interface
```

Como se observa, los métodos carecen de cuerpo: se reducen a una declaración terminada en ";".

Una clase que implementa un interfaz deberá indicarlo a través de la palabra reservada *implements*. Cuando hereda de varios interfaces a la vez, los nombres de dichos interfaces se separan por ",".

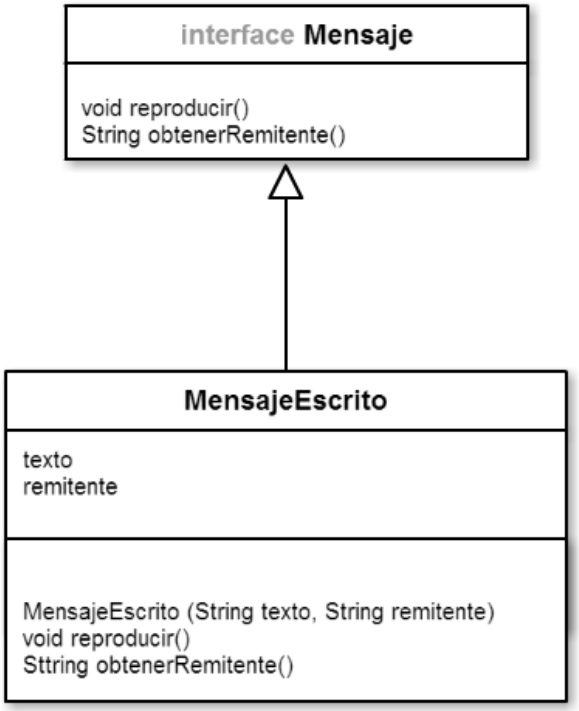
No se debe olvidar nunca que una clase que implementa un interfaz está obligada a implementar todos los métodos declarados en el mismo respetando su signatura exacta.

```
class ClaseQueHereda implements Interfaz {
    // atributos de la clase
    // métodos de la clase
    // métodos del interfaz
}
```

Si implementa más de un interfaz:

```
class ClaseQueHereda implements Interfaz1,..., InterfazN {
    // atributos de la clase
    // métodos de la clase
    // métodos de todos los interfaces
}
```

Ejemplo: vamos a declarar el interfaz *Mensaje* indicando los métodos que deben cumplir aquellas clases que implementen un mensaje.




```
public interface Mensaje {
    public void reproducir();
    public String obtenerRemitente();
}

public class MensajeEscrito implements Mensaje {
    // atributos
    String texto;
    String remitente;
    // constructor
    public MensajeEscrito (String texto, String remitente){
        this.texto = texto;
        this.remitente = remitente;
```

```
    }
    //  métodos
    public void reproducir(){
        System.out.println(texto);
    }
    public String obtenerRemitente(){
        return remitente;
    }
}
```

Tal y como se comentaba al principio, se tradicionalmente los interfaces incluían constantes y declaraciones de método. Con la última versión de Java, Java 8, se ha dotado a los interfaces de dos tipos nuevos de métodos, que sí tienen implementación:

- Métodos default.
- Métodos static.

Los métodos default, permiten añadir métodos a los interfaces con una funcionalidad determinada, de modo que las clases que implementen el interfaz heredan la implementación de dicho método, el cual si lo desean pueden sobreescribirlo.

Realmente es semejante a crear una clase abstracta con varios métodos abstractos y alguno que sí se ha implementado. Las subclases que hereden de la clase abstracta, tienen que implementar los métodos abstractos, pero no los métodos ya implementados por la clase padre no.

En este caso, ¿Qué diferencia hay? La más reseñable es que gracias a los interfaces con métodos default, podemos crear una clase que implemente varios interfaces con funcionalidad, mientras que con las clases abstractas, únicamente se puede heredar de una única clase. Es decir, gracias a este mecanismo se le ha dotado a Java de algo semejante a la herencia múltiple.

Los métodos static, se caracterizan por ser métodos del propio interfaz, no de los objetos que lo implementen (siguiendo una filosofía similar a los métodos static de las clases). Esto da lugar a que todas las instancias de objetos de la clase que implemente dicho interfaz compartirán ese método.

Este tipo de métodos surgió porque en Java existían muchas clases que daban soporte a los interfaces, implementado métodos estáticos que eran utilizados por las clases que heredaban

dicho interfaz. Gracias a este nuevo tipo, estos métodos se pueden ubicar directamente en el interfaz, organizando mejor el código.

4.4 Conversión entre tipos. Casting

Una vez se ha descrito la herencia y los interfaces, aparecen situaciones en las cuales se desea realizar conversiones de tipos. Esto es aplicable a tipos primitivos y a objetos.

En ambos casos, la conversión entre tipos se puede realizar de forma natural o forzándola a través del casting.

En el caso de tipos primitivos:

La conversión natural, se da cuando no supone una pérdida de información, porque el tipo de la variable destino tenga mayor precisión que el tipo de la variable origen.

Por ejemplo, para pasar a un tipo **double** (permite decimales) de un tipo **int** (que sólo permite números enteros), no es necesario realizar el casting ya que no hay pérdida de información.

```
int cuenta1 = 4;
double cuenta2 = cuenta1; // La variable cuenta2 toma el valor 4.0
```

En cambio, si deseamos realizar el proceso inverso, sí se podría producir una pérdida de información, ya que la precisión de int es menor que la de double, y por lo tanto será necesario realizar el casting.

```
double cuenta1 = 4.2;
int cuenta2 = (int)cuenta1; // La variable cuenta2 toma el valor 4
```

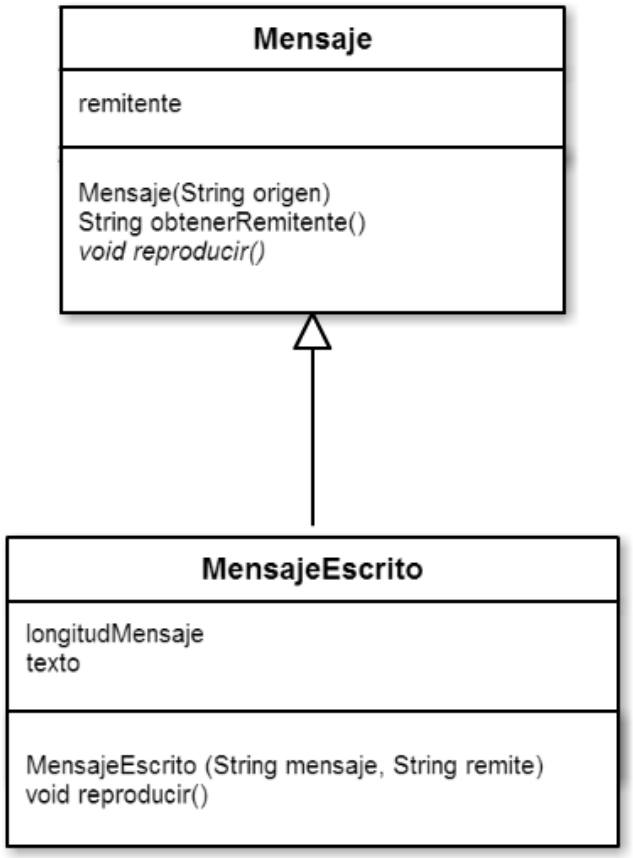
La sintáxis para realizar un casting, dada una variable "variableOrigen" de tipo "TipoOrigen", el casting a un "TipoDestino" será:

```
TipoDestino variableDestino = (TipoDestino) variableOrigen;
```

En el caso de objetos:

La conversión natural se da cuando la clase de la variable origen es una clase hija, de la clase de la variable destino.

Por ejemplo, dada la siguiente jerarquía de clases:



A continuación se observa cómo un método de otra clase recibe por parámetro objetos de la clase MensajeEscrito y posteriormente realiza una conversión de tipos para ejecutar el método reproducir de la clase Mensaje:

```
public void reproducirMensaje (MensajeEscrito mensajeRecibido){

  Mensaje mensajeGenerico = mensajeRecibido;
  mensajeGenerico.reproducir();

}
```

Este ejemplo, únicamente se ha presentado para mostrar cómo es la conversión de tipos: gracias a la herencia, no sería necesario realizar el casting, puesto que un objeto de tipo MensajeEscrito es también de tipo Mensaje y puede ser asignado a cualquier variable de ese tipo.

En cambio, si se desea realizar al revés, es necesario forzar el proceso a través del casting, siguiendo la sintaxis descrita anteriormente, ya que no todos los objetos de tipo Mensaje son necesariamente de tipo MensajeEscrito. A continuación se muestra un

ejemplo de método que recibe un objeto de tipo Mensaje e intenta obtener el remitente para mostrarlo por pantalla.

```
public void mostrarRemitente (Mensaje mensajeRecibido){

  MensajeEscrito mensaje =  (MensajeEscrito) mensajeRecibido;
  String  remitente = mensaje.obtenerRemitente();
  System.out.println(remitente);

}
```

Tal y como se observa, esta conversión está forzada (cast), y por lo tanto es posible que genere errores: en caso de que el objeto no sea de tipo MensajeEscrito sino de otro tipo se producirá un error en tiempo de ejecución. Para evitar esto, es conveniente comprobar previamente si el objeto recibido es del tipo apropiado. Para ello se utilizará palabra reservada **instanceof,** que permite comprobar si un objeto es de un tipo determinado.

```
public void mostrarRemitente (Mensaje mensajeRecibido){

  if (mensajeRecibido instanceof MensajeEscrito){
        MensajeEscrito mensaje  =  (MensajeEscrito) mensajeRecibido;
        String  remitente = mensaje.obtenerRemitente();
        System.out.println(remitente);
   }
}
```

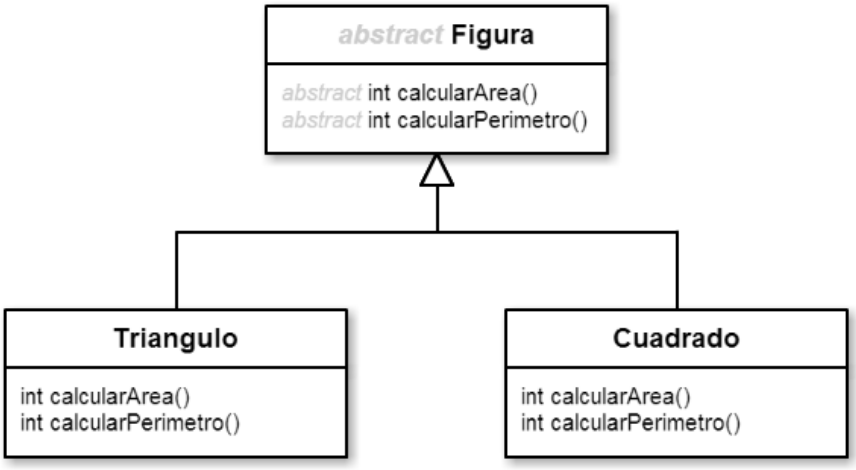
Por último, indicar que el proceso del casting entre objetos no cambia realmente el tipo del objeto, tan sólo permite operar con él como si fuese de otro tipo.

5. Polimorfismo, sobrecarga y sobrescritura

El polimorfismo es la capacidad de que un mismo objeto tenga distintas formas (distintos comportamientos) dependiendo del contexto en el que se halle. El polimorfismo se obtiene por medio de diversos mecanismos, los más comunes son la sobrecarga y sobrescritura.

La sobrescritura, como ya se ha comentado anteriormente, se da cuando una clase hereda de otra, y la clase hija sobrescribe o redefine los métodos de la clase padre, dotando a los objetos de un comportamiento específico. Gracias a este mecanismo, podemos operar con objetos de la clase padre habiendo instanciado objetos de la clase hija, de modo que se comportará en función de clase que lo implementa realmente.

Ejemplo: Tenemos una clase Figura de la cual heredan dos clases Cuadrado y Triangulo, ambas clases hijas heredarán los métodos calcularArea, calcularPerimetro, pero cada uno realizará distintas operaciones para obtener el resultado. Sin embargo un programa que opere con objetos de tipo figura, no tiene que preocuparse de los tipos de figura con los que opera, tan sólo les pedirá que devuelvan su área y ellas por sí mismas devolverán el área.




```
public abstract class Figura(){

   public abstract int calcularArea();

   public abstract int calcularPerimetro(){
}
```

```
class Cuadrado{
  // Se definen las variables del cuadrado
  …
   public int calcularArea(){
     // Se realizarán las operaciones pertinentes
     …
    return area;
 }

   public int calcularPerimetro(){
     // Se realizarán las operaciones pertinentes
     …
    return perimetro;
 }
}

class Triangulo{
  // Se definen las variables del cuadrado
  …
   public int calcularArea(){
     // Se realizarán las operaciones pertinentes
     …
    return area;
 }
```

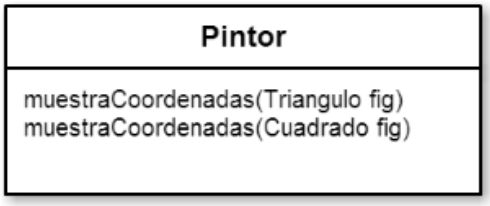
```
    public int calcularPerimetro(){

      // Se realizarán las operaciones pertinentes

      …

     return perimetro;

  }

}
```

```
public static void main (String args[]){
    //Empleamos objetos de la clase figura y almacena Triangulos o Cuadrados

   Figura figura1 = new Triangulo();
   Figura figura2 = new Cuadrado();
   …
   int areaFigura1 = figura1.calcularArea();
   int areaFigura2 = figura2.calcularArea();


}
```

En el ejemplo anterior vemos un ejemplo claro de polimorfismo: aunque la variable figura1 es de tipo Figura, al invocar su método calcularArea se ejecuta el método calcularArea de la clase Triangulo. Es decir, una variable de tipo Figura puede comportarse de diferente manera en función de la clase concreta que implemente el objeto, y no necesitamos saber si figura1 es un triángulo o un cuadrado para poder calcular su área.

Sobrecarga. Es la posibilidad de tener en un mismo objeto varios métodos con el mismo nombre pero con distintos parámetros, de modo que en función de los parámetros se ejecutará un método u otro. De este modo tenemos distintos comportamientos para el mismo objeto.

Ejemplo: dado el ejemplo anterior, si tenemos la clase Pintor que es capaz de representar por pantalla las coordenadas de objetos *Figura*, lógicamente no representa de la misma forma un *Triangulo* que un *Cuadrado*. Para resolver este problema recurrimos a la sobrecarga, declarando dos veces el método muestraCoordenadas con diferentes argumentos.




```
class Pintor {
    public void muestraCoordenadas(Triangulo fig){
        // Obtenemos las tres coordenadas del triangulo
       System.out.println("Mostramos las tres coordenadas del triángulo");
       System.out.println(fig.obtenerCoordenada1());
       System.out.println(fig.obtenerCoordenada2());
       System.out.println(fig.obtenerCoordenada3());
    }
    public void muestraCoordenadas(Cuadrado fig){
        System.out.println("Mostramos las cuatro coordenadas del cuadrado");
       System.out.println(fig.obtenerCoordenada1());
       System.out.println(fig.obtenerCoordenada2());
       System.out.println(fig.obtenerCoordenada3());
        System.out.println(fig.obtenerCoordenada4());
    }
}
```

6. La clase Object. Métodos equals() , toString().

Una vez se ha descrito la herencia, es necesario hacer una mención a la clase Object. Esta clase, es la clase padre de todos los objetos, por defecto cuando se crea una clase, hereda de Object, pese a que no se especifique nada en su declaración.

Por lo tanto, todas las clases heredan los métodos de Object. Entre ellos los más conocidos son:

public boolean equals(Object obj)

Este método compara el objeto recibido por parámetro consigo mismo, es decir dado un objeto, permite consultarle si otro objeto es igual al primero. Permitiendo establecer cuándo dos objetos son iguales. Ojo, es importante tener en cuenta que no es lo mismo decir: Dos objetos son iguales (obj1.equals(obj2)) que dos objetos son el mismo objeto (obj1 == obj2). Ya que dos objetos pueden ser iguales, cuando todos los atributos contengan los mismos datos.

Por ejemplo, podemos tener dos objetos Coche, que sean iguales: el mismo modelo, el mismo color, el mismo motor, etc… Pero siendo dos objetos, en un momento dado uno de los coches puede tener distintos valores en sus atributos, por ejemplo, puede cambiar el número de kilómetros realizados.

```
Coche coche1 = new Coche();

Coche coche2 = new Coche();

Coche coche3 = coche1; //las dos variables apuntan al mismo objeto.

if (coche1 == coche3)

{

  System.out.println("coche1 y coche3 son el mismo objeto");

}

if (coche1.equals(coche2))

{

  System.out.println("coche1 y coche2 son iguales");

}
```

Típicamente dos objetos serán iguales cuando lo sean sus atributos. La clase Object provee una implementación por defecto del método equals que devolverá true

únicamente si se trata del mismo objeto. Por tanto es conveniente sobrescribir el método equals para que devuelva true en caso de que los atributos de los objetos sean iguales.

public String toString()

Este método permite obtener una cadena de caracteres que representa al objeto. La clase Object provee una implementación por defecto que no muestra información útil, por lo que deberemos sobrescribirla para mostrar la información que nos interese. De esta forma por ejemplo, dado un objeto, sería posible consultar los atributos que contiene con una simple llamada al método toString().

Estos métodos permiten crear programas que operen con cualquier objeto de Java, sin conocer su implementación. Por ejemplo, podemos crear un método que sea capaz de mostrar por pantalla una lista de objetos, independientemente de cómo sean estos objetos.

```
public void mostarLista(Object[5] elementos){
  int i=0;
  while(i<5){
    System.out.println(elementos[i].toString());
    i++
  }
}
```

Lógicamente, esto es posible si los objetos sobrescriben el método toString(), tal y como se describió en el tema del polimorfismo.

Como curiosidad, indicar que muchos métodos y operadores de Java hacen uso de los métodos de la clase Object. Por ejemplo, el método *println de la clase System.out e*jecuta el método toString del objeto recibido por parámetro y lo muestra por pantalla.

Por lo tanto, el comportamiento de la siguiente línea de código:

```
System.out.println(elementos[i].toString());
```

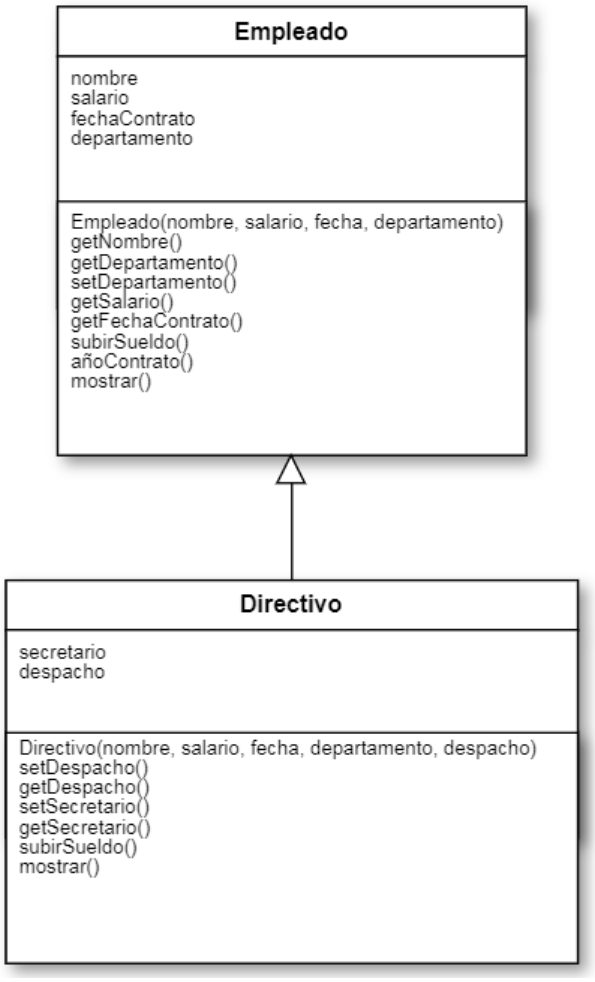
Es el mismo a:

```
System.out.println(elementos[i]);
```

7. Ejercicios Prácticos

7.1 Ejercicio resuelto

Como resumen del capítulo vamos a plantear un ejemplo que consiste en modelizar la clase de los empleados, entre los cuales queremos distinguir a los directivos por presentar características especiales. El diagrama de objetos es el que se representa a continuación:



Los atributos significan lo siguiente:

- **nombre:** es un *String* que representa el nombre del empleado.
- **salario:** es un entero que indica el sueldo mensual del empleado en euros.
- **fechaContrato:** es un atributo de tipo Fecha (Será necesaria una clase Fecha).

- **departamento:** es un atributo de tipo *String* que representa el departamento al que pertenece el empleado.

En cuanto a los métodos:

- **getNombre:** devuelve un *String* con el nombre del empleado.
- **getDepartamento:** devuelve un *String* con el departamento del empleado.
- **setDepartamento:** recibe un *String* con el nuevo departamento del empleado y cambia el valor del atributo departamento.
- **getSalario:** devuelve un entero correspondiente al valor del atributo salario.
- **getFechaContrato:** devuelve la fecha de contratación.
- **subirSueldo:** es un método que no devuelve nada y que tiene por misión subir el sueldo a un empleado. Para ello recibe como parámetros un double que representa el tanto por ciento de aumento y una cantidad entera asociada con su productividad, que viene dada en centenas de euros (por ejemplo, el valor de productividad igual a 12 equivale a 1.200 euros).
- **añoContrato:** es un método que devuelve un entero que indica el año en que se contrató al empleado.
- **mostrar:** es un método sin parámetros que presenta en pantalla toda la información disponible sobre el empleado.

Respecto a la clase *Directivo,* añade como atributos:

- **secretario:** es un atributo de tipo *String* para almacenar el nombre del secretario de ese directivo.
- **despacho:** es un atributo de tipo *String* que contiene el nombre del despacho del directivo.

Y como métodos:

- **setDespacho:** recibe un *String* y actualiza el valor del atributo despacho con el valor de dicho *String.*
- **getDespacho:** método que devuelve un *String* con el valor del despacho.
- **getSecretario:** es un método que devuelve un *String* con el nombre del secretario. Además, este nombre debe aparecer cuando se muestre la información disponible sobre un jefe.
- **setSecretario:** método que recibe un *String* que representa el nombre del nuevo secretario.

- **subirSueldo:** es como en la clase anterior, pero en este caso la subida depende del tanto por ciento y de la antigüedad: por cada año de antigüedad el sueldo se aumenta en medio punto. Para implementar este método, se cuenta con un método de la clase Fecha que devuelve un entero representando el año de una fecha dada, y con el constructor por defecto de la clase Fecha que crea un objeto de ese tipo representando la fecha actual tomándola del sistema.

- **mostrar:** es un método sin parámetros que presenta en pantalla toda la información disponible sobre el directivo. Este método es necesario sobrescribirlo, ya que de no hacerlo mostraría únicamente la información que contiene el empleado.

Además, queremos un programa principal que dé de alta un array con tres empleados, uno de los cuales es directivo. Debe subirles el sueldo un 5% más doscientos euros de productividad. Todos están en el departamento de Informática, pero al jefe le van a pasar al de Marketing, con lo que cambiará de despacho. Se debe mostrar la información disponible sobre ellos.

```
public class Empleado {


    // Declaración de atributos de la clase

    private String nombre;


    private int salario;


    private Fecha fechaContrato;


    private String departamento;


    // Constructor

    public Empleado(String nombre,  int salario, Fecha f, String dep) {

        this.nombre = nombre;

        this.salario = salario;
```

```
            this.fechaContrato = f;
            this.departamento = dep;
        } //contructor

        public void mostrar(){
            System.out.println(nombre +" "+salario+" "+fechaContrato+"
departamento "+departamento);
        } // mostrar

        public void subirSueldo(double tantoPorCiento, int productividad){
            salario = (int) (salario * (1+(tantoPorCiento/100)));
            salario += productividad;
        } // subirSueldo

        public int añoContrato(){
            return fechaContrato.año();
        } // añoContrato

        public String getNombre(){
            return nombre;
        } // getNombre

        public String getDepartamento(){
            return departamento;
        } // getDepartamento
```

```
        public void setDepartamento(String departamento){
                this.departamento = departamento;
        } // setDepartamento

        public int getSalario(){
                return salario;
        } // getSalario

        public Fecha getFechaContrato(){
                return fechaContrato;
        } // dar_fecha_contrato;
}
public class Directivo extends Empleado {
        // atributos
        private String secretario;

        private String despacho;

        // Constructor
        public Directivo (String n, int s, Fecha f, String dep, String despacho){
          super(n,s,f,dep);
          secretario = "";
          this.despacho = despacho;
    }
```

```
    public void subirSueldo(double porcentaje, int prod){
            // A los jefes se les concede un plus de medio punto por año de
            // antigüedad, así que sobreescribmos el método
            Fecha hoy = new Fecha();
            double suplemento = 0.5 * (hoy.año() - añoContrato());
            super.subirSueldo(porcentaje+suplemento, prod);
    } //subirSueldo

    public String getSecretario(){
            return secretario;
    } //getSecretario

    public void setSecretario(String nombre){
            this.secretario = nombre;
    } //setSecretario


    public void mostrar(){
            System.out.println("Jefe: " + getNombre() + " "+ getSalario() +" "+
                            getFechaContrato()+ " Secretario: "+ secretario + "Despacho:
"+despacho);
    } //mostrar

    public void setDespacho(String nuevoDespacho){
```

```
        despacho = nuevoDespacho;
    }  // setDespacho



} // class
import java.text.SimpleDateFormat;
import java.util.Date;


public class Fecha {
        Date fecha;

        public Fecha(){
                fecha = new Date();
        }


        public Fecha (int año,int mes, int dia){
                SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
                try{
                        fecha = sdf.parse(año+"-"+mes+"-"+dia);
                }catch (Exception e) {
                        // TODO: handle exception
                        System.out.println("Error "+e.getMessage());
```

```
            }
        }

        public int dia(){
            SimpleDateFormat sdf = new SimpleDateFormat("dd");
            return Integer.parseInt(sdf.format(fecha));
        }

        public int mes(){
            SimpleDateFormat sdf = new SimpleDateFormat("MM");
            return Integer.parseInt(sdf.format(fecha));
        }


        public int año(){
            SimpleDateFormat sdf = new SimpleDateFormat("yyyy");
            return Integer.parseInt(sdf.format(fecha));
        }

        public String toString(){
            SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
            return sdf.format(fecha);
        }
}
```

```
public class TestEmpleados {

	public static void main(String[] args) {

		Empleado[] plantilla = new Empleado[3];


		plantilla[0] = new Empleado("Pepe Lopez",15000,new
Fecha(1998,12,15),"Informatica");

		plantilla[1] = new Directivo("Rosa Perez",25000,new
Fecha(1997,1,11),"Informatica","2031");

		plantilla[2] = new Empleado("Maria Ruiz",10000,new
Fecha(1995,3,5),"Informatica");


		for (int i=0;i<3;i++){

			// subimos el sueldo un 5%

			plantilla[i].subirSueldo(5, 200);

			if (plantilla[i] instanceof Directivo) {

				plantilla[i].setDepartamento("Marketing");

				((Directivo)plantilla[i]).setDespacho("2560");


			}

			// mostramos el empleado

			plantilla[i].mostrar();

		}// for i

	}// main

	}// Test_Empleados
```