Tema 4: Desarrollo de clases

1. Clases

1.1 Concepto de clase

Una **clase** es un molde que especifica las características comunes de un conjunto de **objetos**. Los programas están formados por un conjunto de **clases** a partir de las cuales se van creando **objetos** que interactuarán unos con otros. Las **clases** están compuestas por **atributos** y **métodos**.

1.2 Declaración de una clase

Esquema general de la declaración de una clase en Java:

```
// Cabecera de la clase
[modificadores] class <NombreClase> [herencia] [interfaces] {
// Cuerpo de la clase
Declaración de los atributos
Declaración de los métodos
}
```

Esquema más preciso:

```
// Cabecera de la clase
[public] [final | abstract] class <NombreClase> [extends <NombreSuperClase>] [implements <NombreInterface1>] [[implements <NombreInterface2>] ...] {
// Cuerpo de la clase
Declaración de los atributos
Declaración de los métodos
}
```

1.2.1 Modificadores de acceso

Modificador **public**. Indica que la clase es visible (se pueden crear objetos de esa clase) desde cualquier otra clase, es decir, desde cualquier otra parte del programa. Sólo puede haber una clase public (clase principal) en un archivo .java. El resto de clases que se definan en ese archivo no serán públicas.

Modificador **abstract**. Indica que la clase es abstracta. Una clase abstracta no es instanciable, es decir, no es posible crear objetos de esa clase, habrá que utilizar clases que hereden de ella y esto puede resultar útil a la hora de crear una jerarquía de clases.

Modificador **final**. Indica que no podrás crear clases que hereden de ella. Los modificadores final y abstract son excluyentes (solo se puede utilizar uno de ellos).

Si no se especifica modificador, la clase solo podrá ser utilizada desde clases que estén en el mismo **paquete**.

2. Atributos

2.1 Concepto de atributo

Atributos son aquellas características propias de la clase y de los objetos que se crearán con ese "molde".

2.2 Declaración de atributos

Esquema general de la declaración de un atributo en una clase de Java:

[modificadores] <tipo> <nombreAtributo>;

Esquema más preciso:

[private | protected | public] [static] [final] [transient] [volatile] <tipo> <nombreAtributo>;

Tipo. Indica el tipo del atributo. Puede tratarse de un tipo primitivo (int, char, bool, double, etc) o bien de uno referenciado (objeto, array, etc.).

Nombre. Identificador único para el nombre del atributo. Por convenio se suelen utilizar las minúsculas. En caso de que se trate de un identificador que contenga varias palabras, a partir de la segunda palabra se suele poner la letra de cada palabra en mayúsculas. Por ejemplo: primerValor, valor, puertaIzquierda, cuartoTrasero,equipoVecendor, sumaTotal, nombreCandidatoFinal, etc.

2.2.1 Modificadores de un atributo

Son palabras reservadas que permiten modificar la utilización del atributo y permiten indicar cierto comportamiento de un atributo a la hora de utilizarlo. Entre los modificadores de un atributo podemos distinguir:

Modificadores de acceso. Indican la forma de acceso al atributo desde otra clase. Son modificadores excluyentes entre sí, es decir sólo se puede poner uno.

Cuadro de niveles accesibilidad a los atributos de una clase (Tema 3):

zona	private (privado)	sin modificador (*friendly*)	protected (protegido)	public (público)
Misma clase	X	X	X	X
Subclase en el mismo paquete		X	X	X
Clase (no subclase) en el mismo paquete		X		X
Subclase en otro paquete			X	X
No subclase en otro paquete				X

Modificadores de contenido.

No son excluyentes. Pueden aparecer varios a la vez.

Modificador **static**. Hace que el atributo sea común para todos los objetos de una misma clase. Es decir, toda la clase compartirá ese mismo atributo con el mismo valor, así que puede decirse que la existencia del atributo no depende de la existencia del objeto, sino de la propia clase y por tanto sólo habrá uno, independientemente del número de objetos que se creen.
Uno de los ejemplos más habituales (y sencillos) de atributos estáticos o de clase es el de un **contador** que indica el número de objetos de esa clase que se han ido creando.

Modificador **final**. Indica que el atributo es una constante y su valor no podrá ser modificado a lo largo de la vida del objeto. Por convenio, el nombre de los atributos constantes (final) se escribe con todas las letras en mayúsculas.

Otros modificadores: transient y volatile. El primero se utiliza para indicar que un atributo es transitorio (no persistente) y el segundo es para indicar al compilador que no debe realizar optimizaciones sobre esa variable.

3. Métodos

3.1 Concepto de método

Los métodos en una clase recogen cuáles son las acciones que van a poder realizar los objetos o realizarse sobre ellos.

3.2 Declaración de métodos

Esquema general de la cabecera de un método:

```
// Cabecera del método
[modificadores] <tipo> <nombreMétodo> ( [<lista_parametros>] ) [throws <lista_excepciones>]
```

Esquema más preciso:

```
//Cabecera de método.
[private | protected | public] [static] [abstract] [final] [native] [synchronized]
<tipo> <nombreMétodo> ( [<lista_parametros>] ) [throws <lista_excepciones>]
```

Por tanto, la declaración de un método puede incluir los siguientes elementos:
Modificadores (que se explican en el próximo subapartado)

El **tipo devuelto** (o tipo de retorno), que consiste en el tipo de dato (primitivo o referencia) que el método devuelve tras ser ejecutado. Si se elige void como tipo devuelto, el método no devolverá ningún valor.

El **nombre del método**, por convenio, un verbo en minúscula o bien un nombre formado por varias palabras que comience por un verbo en minúscula, seguido por adjetivos, nombres, etc. los cuales sí aparecerán en mayúsculas.

Una **lista de parámetros** separados por comas y entre paréntesis donde cada parámetro debe ir precedido por su tipo. Si el método no tiene parámetros la lista estará vacía y únicamente aparecerán los paréntesis.
Es posible utilizar una construcción especial llamada **varargs** (argumentos variables) que permite que un método pueda tener un número variable de parámetros. Para utilizar este mecanismo se colocan unos puntos suspensivos (tres puntos: "...") después del tipo del cual se puede tener una lista variable

de argumentos, un espacio en blanco y a continuación el nombre del parámetro que aglutinará la lista de argumentos variables.

<nombreMetodo> **(<tipo>... <nombre>)**

Es posible además mezclar el uso de varargs con parámetros fijos, apareciendo esta al final (y sólo puede aparecer una).
En realidad, se trata una manera transparente de pasar un array con un número variable de elementos para no tener que hacerlo manualmente. Dentro del método habrá que ir recorriendo el array para ir obteniendo cada uno de los elementos de la lista de argumentos variables.

Una **lista de excepciones** que el método puede lanzar. Se utiliza la palabra reservada throws seguida de una lista de nombres de excepciones separadas por comas. No es obligatorio que un método incluya una lista de excepciones, aunque muchas veces será conveniente.

El **cuerpo del método**, encerrado entre llaves. El cuerpo contendrá el código del método (una lista sentencias y estructuras de control en lenguaje Java) así como la posible declaración de variables locales.

3.2.1 Modificadores de un método

Modificadores de acceso. Son los mismos que en el caso de los atributos (por omisión o de paquete, **public, private y protected**) y tienen el mismo cometido (acceso al método sólo por parte de clases del mismo paquete, o por cualquier parte del programa, o sólo para la propia clase, o también para las subclases).

Modificadores de contenido. Son también los mismos que en el caso de los atributos (**static y final**).
Un método final es un método que no permite ser sobrescrito por las clases descendientes de la clase a la que pertenece el método.

Otros modificadores: abstract, native, synchronized.
El modificador **native** es utilizado para señalar que un método ha sido implementado en código **C** o **C++**. Se indica la cabecera del método, pues no tiene cuerpo escrito en Java.
Un método **abstract** es un método que no tiene implementación (el cuerpo está vacío). La implementación será realizada en las clases descendientes. Un método sólo puede ser declarado como abstract si se encuentra dentro de una clase abstract.

Si un método es declarado como **synchronized**, el entorno de ejecución obligará a que cuando un proceso esté ejecutando ese método, el resto de procesos que tengan que llamar a ese mismo método deberán esperar a que el otro proceso termine. Puede resultar útil si sabes que un determinado método va a poder ser llamado frecuentemente por varios procesos a la vez.

3.3 Constructores

Como se ha estudiado en la unidad anterior, el constructor es un método particular de la clase. La estructura de los constructores es similar a la de cualquier método, con las excepciones de que no tiene tipo de dato devuelto (no devuelve ningún valor) y que el nombre del método constructor debe ser obligatoriamente el nombre de la clase.
Si se ha creado un constructor con parámetros y no se ha implementado el constructor por defecto, el intento de utilización del constructor por defecto producirá un error de compilación (el compilador no lo hará por nosotros).

3.3.1 Constructores de Copia

Una forma de iniciar un objeto puede ser mediante la copia de los valores de los atributos de otro objeto ya existente. Estaría bien disponer de un constructor que hiciera copias idénticas de un objeto que ya tienes configurado, algo así como poder clonar el objeto tantas veces como haga falta. A este tipo de mecanismo se le llama **constructor copia** o **constructor de copia**.
Un constructor copia es un método constructor que recibe como parámetro una referencia al objeto cuyo contenido se desea copiar. Este método revisa cada uno de los atributos del objeto recibido como parámetro y se copian todos sus valores en los atributos del objeto que se está creando en ese momento en el método constructor.
Un ejemplo de constructor copia para la clase Punto podría ser:

```
public Punto (int x, int y){
this.x= x;
this.y= y;
}
```

```
public Punto (Punto p){
this.x= p.obtenerX();
this.y= p.obtenerY();
}
```

Un ejemplo de utilización de ese constructor podría ser:

```
Punto p1, p2;
p1= new Punto (10, 7);
p2= new Punto (p1);
```

En este caso el objeto p2 se crea a partir de los valores del objeto p1.

3.4 Destrucción de objetos

Cuando un objeto deja de ser utilizado, los recursos usados por él (memoria, acceso a archivos, conexiones con bases de datos, etc.) deberían de ser liberados para que puedan volver a ser utilizados por otros procesos (mecanismo de destrucción del objeto).
De la destrucción se encarga un proceso del entorno de ejecución conocido como **recolector de basura** (**garbage collector**). Este proceso va buscando periódicamente objetos que ya no son referenciados (no hay ninguna variable que haga referencia a ellos) y los marca para ser eliminados. Posteriormente los irá eliminando de la memoria cuando lo considere oportuno (en función de la carga del sistema, los recursos disponibles, etc.).

En realidad, en Java también es posible implementar el método destructor de una clase, se trata del método finalize(). Si necesitas implementar un destructor (normalmente no será necesario), debes tener en cuenta que:

- El nombre del método destructor debe ser finalize ().
- No puede recibir parámetros.
- Solo puede haber un destructor en una clase. No es posible la sobrecarga dado que no tiene parámetros.
- No puede devolver ningún valor. Debe ser de tipo void.

Este método finalize es llamado por el recolector de basura cuando va a destruir el objeto. Si ese método no existe, se ejecutará un destructor por defecto (el método finalize que contiene la clase Object, de la cual heredan todas las clases en Java) que liberará la memoria ocupada por el objeto. Se recomienda, por tanto, que si un objeto utiliza determinados recursos de los cuales no tienes garantía que el entorno de ejecución los vaya a liberar (cerrar archivos, cerrar conexiones de red, cerrar conexiones con bases de datos, etc.), implementes explícitamente un método finalize en tus clases. Si el único recurso que utiliza tu clase es la memoria necesaria para albergar sus atributos, eso sí será liberado sin problemas.

Si en un momento dado fuera necesario garantizar que el proceso de finalización (método finalize) sea invocado, puedes recurrir al método runFinalization () de la clase System para forzarlo: **System.runFinalization ();**
Este método se encarga de llamar a todos los métodos finalize de todos los objetos marcados por el recolector de basura para ser destruidos.

4 Introducción a la herencia

La **herencia** es uno de los conceptos fundamentales que introduce la programación orientada a objetos. La idea fundamental es permitir crear nuevas clases aprovechando las características (atributos y métodos) de otras clases ya creadas evitando así tener que volver a definir esas características.
A una clase que hereda de otra se le llama **subclase o clase hija** y aquella de la que se hereda es conocida como **superclase o clase padre**. También se puede hablar en general de **clases descendientes** o **clases ascendientes**.
Al heredar, la subclase adquiere todas las características (atributos y métodos) de su superclase, aunque algunas de ellas pueden ser sobrescritas o modificadas dentro de la subclase (a eso se le suele llamar **especialización**).
Una clase puede heredar de otra que a su vez ha podido heredar de una tercera y así sucesivamente. Esto significa que las clases van tomando todas las características de sus clases ascendientes (no sólo de su superclase o clase padre inmediata).

Una clase **SerVivo** que tendrá un conjunto de atributos (por ejemplo: respiración, alimentación, reproducción, hábitat,…) y de métodos (por ejemplo: nacer, crecer, desplazar,...).
Se pueden crear otras clases que heredan de SerVivo, por ejemplo, **Animal** que hereda todas las características de SerVivo, y además tendrá otras como (extremidades, colorOjos, emitirSonido,…). A su vez la clase **Mamífero** hereda de la clase Animal añadiendo características como tiempoLactancia.

4.1 Creación de clases heredadas.

class <NombreClase> **extends** <nombreSuperClase> {
…
}

En el ejemplo anterior:

```
class Animal extends SerVivo  {
…
}
```

```
class Mamífero extends Animal {
…
}
```

Cualquier clase con la que trabajes tendrá un ascendiente. Si en la declaración de clase no indicas la clase de la que se hereda (no se incluye un extends), el compilador considerará automáticamente que se hereda de la **clase Object,** que es la clase que se encuentra en el nivel superior de toda la jerarquía de clases en Java (y que es la única que no hereda de nadie).