MODIFICADORES

Los modificadores son elementos del lenguaje que se colocan delante de la definición de variables locales, datos miembro, métodos o clases y que alteran o condicionan el significado del elemento.

Modificadores de acceso: public, private, protected y default

Los modificadores de acceso, determinan desde qué clases se puede acceder a un determinado elemento. En Java tenemos 4 tipos: **`public`**, **`private`**, **`protected`** y el tipo por defecto, que no tiene ninguna palabra clave asociada, pero se suele conocer como *default* o *package-private*.

Si no especificamos ningún modificador de acceso se utiliza el nivel de acceso por defecto, que consiste en que el elemento puede ser accedido sólo desde las clases que pertenezcan al mismo paquete.

El nivel de acceso **`public`** permite a acceder al elemento desde cualquier clase, independientemente de que esta pertenezca o no al paquete en que se encuentra el elemento.

`private`, por otro lado, es el modificador más restrictivo y especifica que los elementos que lo utilizan sólo pueden ser accedidos desde la clase en la que se encuentran. Este modificador sólo puede utilizarse sobre los miembros de una clase y sobre interfaces y clases internas, no sobre clases o interfaces de primer nivel, dado que esto no tendría sentido.

Es importante destacar también que **`private`** convierte los elementos en privados para otras clases, no para otras instancias de la clase. Es decir, un objeto de una determinada clase puede acceder a los miembros privados de otro objeto de la misma clase, por lo class MiObjeto {

```
1.    private short valor = 0;
2.
3.    MiObjeto(MiObjeto otro) {
4.      valor = otro.valor;
5.    }
6.  }
```

El modificador **`protected`**, por último, indica que los elementos sólo pueden ser accedidos desde su mismo paquete (como el acceso por defecto) y desde cualquier clase que extienda la clase en que se encuentra, independientemente de si esta se encuentra en el mismo paquete o no.

Los distintos modificadores de acceso quedan resumidos en la siguiente tabla:

Modificadores de acceso

	La misma clase	Otra clase del mismo paquete	Subclase de otro paquete	Otra clase de otro paquete
`public`	X	X	X	X
`protected`	X	X	X	
`default`	X	X		
`private`	X			

Modificador static

A pesar de lo que podría parecer por su nombre, heredado de la terminología de C++, el modificador **`static`** no sirve para crear constantes, sino para crear miembros que pertenecen a la clase, y no a una instancia de la clase. Esto implica, entre otras cosas, que no es necesario crear un objeto de la clase para poder acceder a estos atributos y métodos. Este es el motivo por el cual es obligatorio que **`main`** se declare como **`static`**; de esta forma no tenemos que ofrecer un constructor vacío para la clase que contiene el método, o indicar de alguna forma a la máquina virtual cómo instanciar la clase.

Un uso del modificador **`static`** sería, por ejemplo, crear un contador de los objetos de la clase que se han creado, incrementando la variable estática en el constructor:

```
1. class Usuario {
2.   static int usuarios = 0;
3.
4.   Usuario() {
5.     usuarios++;
6.   }
7. }
```

Como es de esperar, dado que tenemos acceso a los atributos sin necesidad de crear un objeto, los atributos estáticos como `usuarios` no se inicializan al crear el objeto, sino al cargar la clase.

Podemos acceder a estos métodos y atributos bien desde la propia clase

```
1.  public class Ejemplo {
2.    public static void main(String[] args) {
3.      Usuario raul = new Usuario();
4.      Usuario juan = new Usuario();
5.      System.out.println("Hay " + Usuario.usuarios + " usuarios");
6.    }
7.  }
```

o bien desde una instancia cualquiera de la clase:

```
1.  public class Ejemplo {
2.    public static void main(String[] args) {
3.      Usuario raul = new Usuario();
4.      Usuario juan = new Usuario();
5.      System.out.println("Hay " + raul.usuarios + " usuarios");
6.    }
7.  }
```

Otro uso sería el de crear una recopilación de métodos y atributos relacionados a los que poder acceder sin necesidad de crear un objeto asociado, que podría no tener sentido o no ser conveniente, como es el caso de la clase `Math`.

```
1.  public class Ejemplo {
2.    public static void main(String[] args) {
3.      System.out.println("PI es " + Math.PI);
4.      System.out.println("El coseno de 120 es " + Math.cos(120));
5.    }
6.  }
```

Una característica no muy conocida que se introdujo en Java 1.5 son los ***static imports***, una sentencia similar al `import` habitual, con la salvedad de que esta importa miembros estáticos de las clases, en lugar de clases de los paquetes, permitiendo utilizar estos miembros sin indicar el espacio de nombres en el que se encuentran. El ejemplo anterior podría haberse escrito también de la siguiente forma utilizando esta característica:

```
1.  import static java.lang.Math.*;
2.
3.  public class Ejemplo {
4.    public static void main(String[] args) {
5.      System.out.println("PI es " + PI);
6.      System.out.println("El coseno de 120 es " + cos(120));
7.    }
8.  }
```

Si por algún motivo requerimos cualquier tipo de computación para inicializar nuestras variables estáticas, utilizaremos lo que se conoce como bloque estático o inicializador estático, el cuál se ejecuta una sola vez, cuando se carga la clase.

```
1. public class Reunion {
2.   static {
3.     int zona_horaria = Calendar.getInstance().get(Calendar.ZONE_OFFSET)
4.       / (60 * 60 * 1000);
5.   }
6. }
```

Bloques de inicialización `static`. Son bloques de código -encerrados entre {}-, y precedidos del modificador `static`. Por ejemplo:

```
1. private static final int arrayFib[];
2. static{
3.     arrayFib = new int[100];
4.     arrayFib[0] = 0;
5.     arrayFib[1] = 1;
6.     for ( int i = 2; i < arrayFib.length; i++ ){
7.         arrayFib[i] = arrayFib[i-1] + arrayFib[i-2];
8.     }
9. }
```

En una clase puede haber cualquier número de bloques de inicialización `static`, y pueden aparecer en cualquier lugar del cuerpo de la clase. El servicio de tiempo de ejecución garantiza que estos bloques serán llamados en el orden en que aparecen en el código fuente.

Una alternativa al bloque de inicialización `static` es utilizar un método static. NOTA: aquí definiremos el método como `private`, ya que limitará a un uso interno de la clase.

```
1. private static final int arrayFib[] = inicializaClassVariable();
2.
3. private static int initializeClassVariable() {
4.       arrayFib = new int[100];
5.       arrayFib[0] = 0;
6.       arrayFib[1] = 1;
7.       for ( int i = 2; i < arrayFib.length; i++ ){
8.         arrayFib[i] = arrayFib[i-1] + arrayFib[i-2];    }
9. }
```

Modificador abstract

La palabra clave **`abstract`** indica que no se provee una implementación para un cierto método, sino que la implementación vendrá dada por las clases que extiendan la clase actual.

Una clase que tenga uno o más métodos `abstract` debe declararse como `abstract` a su vez.

Una clase **`abstract`** no podrá ser instanciada. Es decir, no podremos crear ningún objeto de esa clase.

En cierto modo, una clase **`abstract`** es similar a un interface, ya que podemos definir métodos que tendrán que ser implementados por otras clases que hereden o extiendan a estas. Pero a diferencia de una clase **`abstract`**, una interface no admite métodos concretos ni atributos no constantes, y todos miembros son **`public`**.

Modificador final

Indica que una variable, método o clase no se va a modificar, lo cuál puede ser útil para añadir más semántica, por cuestiones de rendimiento, y para detectar errores.

Si una variable se marca como **`final`**, no se podrá asignar un nuevo valor a la variable. Si se permite no asignarle valor en el momento de declararla, y hacerlo a posteriori; es lo que se suele denominar `constantes blancas`, que nos permitirías tener diferentes valores para cada objeto.

Si una clase se marca como `final`, no se podrá extender la clase. Marca el fin de la jerarquía de herencia. Además sus métodos son implícitamente `final`.

Si es un método el que se declara como `final`, no se podrá sobrescribir. Los `private` lo son por defecto. Si se sobrescribe, es realmente un nuevo método, no una sobrecarga (upcast).

Si un parámetro de un método es un objeto y se declara como `final`, no se pueden cambiar dentro del método. Sí se les puede asignar el valor null. NOTA: los tipos primitivos en argumentos son read only (no necesitaría el modificador final). Los demás argumentos se pasan como referencias al objeto, lo que implica que su modificación sería permanente.

Una variable con modificadores **`static`** y **`final`** sería lo más cercano en Java a las **constantes** de otros lenguajes de programación.

Modificadores strictfp

Su uso sobre una clase, interfaz o método sirve para mejorar su portabilidad haciendo que los cálculos con números flotantes se restrinjan a los tamaños definidos por el estándar de punto flotante de la IEEE (float y double), en lugar de aprovechar toda la precisión que la plataforma en la que estemos corriendo el programa pudiera ofrecernos.

No es aconsejable su uso a menos que sea estrictamente necesario.

Modificador native

`native` es un modificador utilizado cuando un determinado método está escrito en un lenguaje distinto a Java, normalmente C, C++ o ensamblador para mejorar el rendimiento. La forma más común de implementar estos métodos es utilizar JNI (Java Native Interface).

Modificador transient

Utilizado para indicar que los atributos de un objeto no son parte persistente del objeto o bien que estos no deben guardarse y restaurarse utilizando el mecanismo de socialización estándar.

Modificadores volatile y synchronized

`volatile` es, junto con **`synchronized`**, uno de los mecanismos de sincronización básicos de Java.

Se utiliza este modificador sobre los atributos de los objetos para indicar al compilador que es posible que dicho atributo vaya a ser modificado por varios **threads** de forma simultanea y asíncrona, y que no queremos guardar una copia local del valor para cada thread a modo de caché, sino que queremos que los valores de todos los threads estén sincronizados en todo momento, asegurando así la visibilidad del valor actualizado a costa de un pequeño impacto en el rendimiento.

`volatile` es más simple y más sencillo que `synchronized`, lo que implica también un mejor rendimiento. Sin embargo `volatile`, a diferencia de `synchronized`, no proporciona atomicidad, lo que puede hacer que sea más complicado de utilizar.

Una operación como el incremento, por ejemplo, no es atómica. El operador de incremento se divide en realidad en 3 instrucciones distintas (primero se lee la variable, después se incrementa, y por último se actualiza el valor) por lo que algo como lo siguiente podría causarnos problemas a pesar de que la variable sea `volatile`:

```
1. volatile int contador;
2.
3. public void aumentar() {
4.   contador++;
5. }
```

En caso de que necesitemos atomicidad podemos recurrir a `synchronized` o a cosas más avanzadas, como las clases del API `java.util.concurrent` de Java 5.

`synchronized` se diferencia de `volatile` entre otras cosas en que este modificador se utiliza sobre bloques de código y métodos, y no sobre variables. Al utilizar `synchronized` sobre un bloque se añade entre paréntesis una referencia a un objeto que utilizaremos a modo de lock.

```
1. int contador;
2.
3. public void aumentar() {
4.   synchronized(this) {
5.     contador++;
6.   }
7. }
```

```
1. int contador;
2.
3. public void synchronized aumentar() {
4.   contador++;
5. }
```