MODIFICADORES DE ACCESO: PUBLIC, PRIVATE, PROTECTED Y DEFAULT

Los modificadores de acceso, como su nombre indica, 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 que algo como lo siguiente sería perfectamente válido:


```
01. class MiObjeto {
02.   private short valor = 0;
03.
04.   MiObjeto(MiObjeto otro) {
05.     valor = otro.valor;
06.   }
07. }
```

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. Este modificador, como `private`, no tiene sentido a nivel de clases o interfaces no internas.

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			

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:


```
01. class Usuario {
02.   static int usuarios = 0;
03.
04.   Usuario() {
05.     usuarios++;
06.   }
07. }
```

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


```
01. public class Ejemplo {
02.   public static void main(String[] args) {
03.     Usuario raul = new Usuario();
04.     Usuario juan = new Usuario();
```

```
05.       System.out.println("Hay " + Usuario.usuarios + " usuarios");
06.     }
07.   }
```

o bien desde una instancia cualquiera de la clase:

```
view plain  copy to clipboard  print  ?
01.   public class Ejemplo {
02.     public static void main(String[] args) {
03.       Usuario raul = new Usuario();
04.       Usuario juan = new Usuario();
05.       System.out.println("Hay " + raul.usuarios + " usuarios");
06.     }
07.   }
```

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`.

```
view plain  copy to clipboard  print  ?
01.   public class Ejemplo {
02.     public static void main(String[] args) {
03.       System.out.println("PI es " + Math.PI);
04.       System.out.println("El coseno de 120 es " + Math.cos(120));
05.     }
06.   }
```

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:

```
view plain  copy to clipboard  print  ?
01.   import static java.lang.Math.*;
02.
03.   public class Ejemplo {
04.     public static void main(String[] args) {
05.       System.out.println("PI es " + Math.PI);
06.       System.out.println("El coseno de 120 es " + Math.cos(120));
07.     }
08.   }
```

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.

```
view plain  copy to clipboard  print  ?
01.  public class Reunion {
02.    static {
03.      int zona_horaria = Calendar.getInstance().get(Calendar.ZONE_OFFS
04.        / (60 * 60 * 1000);
05.    }
06.  }
```

Por último, una curiosidad relacionada que podéis utilizar para romper el hielo con una programadora Java es que podemos utilizar un bloque static para escribir un programa sencillo sin necesidad de un `main`, añadiendo una llamada a `System.exit`para que el programa termine tras cargar la clase sin intentar llamar al método `main` 

```
view plain  copy to clipboard  print  ?
01.  public class Ejemplo {
02.    static {
03.      System.out.println("Hola mundo");
04.      System.exit(0);
05.    }
06.  }
```

STRICTFP

`strictfp` es un modificador de lo más esotérico, muy poco utilizado y conocido cuyo nombre procede de *strict floating point*, o punto flotante estricto.

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.

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).

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 serialización estándar.

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`:

```
view plain  copy to clipboard  print  ?
01.  volatile int contador;
02.
03.  public void aumentar() {
04.    contador++;
05.  }
```

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.

```
view plain  copy to clipboard  print  ?
01.  int contador;
02.
03.  public void aumentar() {
04.    synchronized(this) {
05.      contador++;
06.    }
07.  }
```

```
view plain  copy to clipboard  print  ?
01.  int contador;
02.
03.  public void synchronized aumentar() {
04.    contador++;
05.  }
```

ABSTRACT

Un viejo conocido para la mayoría de los programadores Java. 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.

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 una clase se marca como `final`, no se podrá extender la clase. Si es un método el que se declara como `final`, no se podrá sobreescribir.

Algo muy a tener en cuenta a la hora de utilizar este modificador es que si es un objeto lo que hemos marcado como `final`, esto no nos impedirá modificar el objeto en sí, sino tan sólo usar el operador de asignación para cambiar la referencia. Por lo tanto el siguiente código no funcionaría:

```
view plain  copy to clipboard  print  ?
01.  public class Ejemplo {
02.    public static void main(String[] args) {
03.      final String cadena = "Hola";
04.      cadena = new String("Adios");
05.    }
06.  }
```

pero sin embargo, este si:

```
view plain  copy to clipboard  print  ?
01.  public class Ejemplo {
02.    public static void main(String[] args) {
03.      final String cadena = "Hola";
04.      cadena.concat(" mundo");
05.    }
06.  }
```

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