Centro Asociado Palma de Mallorca

# Introducción Práctica de Programación Java

Antonio Rivero Cuesta

# Sesión

# VII

Documentación del Código .......................................... 7
Comentarios .......................................................... 10
Comentario de Línea ............................................... 11
Comentario de Multilínea.......................................... 12
Comentario de Documentación ................................ 13
Líneas en Blanco y Espacios ................................... 17
Etiquetas ................................................................ 19
Orden de las Etiquetas ............................................. 20

Convenios de Programación en Java........................ 23
Estructura de una Clase ........................................ 26
Modificadores de Acceso ...................................... 29
Visibilidad de los Campos..................................... 31
Visibilidad de los Métodos.................................... 33
Métodos get y set................................................... 34
Método get ............................................................ 35
Métodos set ........................................................... 37

Ciclo de Vida en Cascada ........................................ 39
Acoplamiento y Cohesión ........................................ 40
Acoplamiento ........................................................ 41
Cohesión................................................................ 44
Tipos Enumerados.................................................. 46
toString( ) ............................................................. 47
printf...................................................................... 48
Palabras Clave static y final ................................... 61

static ........................................................................ 65

final ......................................................................... 68

# Documentación del Código

Las ***buenas prácticas de programación*** son técnicas que le ayudarán a producir programas más claros, comprensibles y fáciles de mantener.

Ciertas organizaciones requieren que todos los programas comiencen con un comentario que explique su propósito, el autor, la fecha y la hora de la última modificación del mismo.

Es un complemento importante para obtener código de buena calidad.

Permite al programador comunicar sus intenciones a los lectores humanos en un lenguaje natural de alto nivel.

En lugar de forzarlos a leer código de nivel relativamente bajo.

Los programadores pueden usarla sin tener que conocer los detalles de su implementación.

# Comentarios

En un programa Java hay tres tipos de comentarios.

- Comentario de línea.
- Comentario de multilínea.
- Comentario de documentación, Javadoc.

# Comentario de Línea

Empieza con //.

Comienza con estos caracteres y termina al final de la línea.

También lo llamamos **comentario al final de una línea.**

```
// Programa para imprimir texto.
```

# Comentario de Multilínea

Empieza por /*.

Y termina por */.

```
/* Aquí empieza el bloque comentado
y aquí acaba */
```

El compilador ignora todo el texto contenido dentro del comentario.

# Comentario de Documentación

Empieza por /**.

Y termina por */.

Java dispone de la herramienta Javadoc para documentar automáticamente los programas.

En un comentario de documentación normalmente se indica el autor y la versión del software.

En Java la documentación del código es fundamental.

Resulta evidente en cuanto se consulta la documentación de las clases que se proporciona directamente en los paquetes que distribuye Sun.

También podremos hacer lo mismo nosotros.

Java promueve que el programador documente el funcionamiento de las clases dentro del propio código.

Los javadoc generan un conjunto de páginas en HTML por las que se puede navegar utilizando cualquier navegador Web.

```java
/**
* Programa HolaMundo
* @author Antonio Rivero
* @version 1.00 2015/7/9
*/
public class HolaMundo{
  public static void main(String[] args){
    System.out.println("Hola mundo");
  } //Cierre del main
} //Cierre de la clase
```

Se debe poner un comentario:

- Al principio de cada archivo con la descripción y objetivo del mismo.
- Antes de cada método, explicando para qué sirve.
- Antes de cada algoritmo, explicando qué hace el algoritmo.
- Antes de cada definición de estructura de datos, indicando cuál es su objetivo.
- Antes de cada parte significativa del programa.

El uso de comentarios hace más claro y legible un programa.

En los comentarios se debe decir qué se hace, para qué y cuál es el fin de nuestro programa.

Conviene utilizar comentarios siempre que merezca la pena hacer una aclaración sobre el programa.

# Líneas en Blanco y Espacios

Los Utilizamos para mejorar la legibilidad del programa.

Los comentarios deben ir precedidos por una línea en blanco para separarlos lógicamente de la parte precedente del programa.

# Etiquetas

Java complementa la tarea de documentación utilizando ciertas etiquetas en los comentarios.

# Orden de las Etiquetas

@author: En clases e interfaces. Se pueden poner varios. En este caso resulta apropiado hacerlo en orden cronológico.

@version: En clases y en interfaces.

@param: En métodos y constructores. Se ponen tantos como argumentos tenga el constructor o el método.Deberían aparecer en el mismo orden en que se declaran.

@return: En métodos.

@exception: En constructores y métodos. Deberían aparecer en el mismo orden en que se declaran o en orden alfabético.

@throws: Con Javadoc 1.2 es un sinónimo de @exception.

@see: Se pueden poner varios. Se recomienda empezar por los más generales e ir indicando después los más concretos.

@since. En clases. Versión desde la que está presente la clase.

@deprecated. En métodos. Indicación de que el método es obsoleto.

# Convenios de Programación en Java

Tener unas reglas para programar es importante porque:

- La mayor parte del tiempo se dedica a leer programas, por lo que resulta importante que sean fáciles de leer y de entender.
- Seguir unas reglas de programación permite mejorar la legibilidad del código escrito y entender fácilmente el código que no es de uno mismo.

- Si se da código propio a terceras personas, o se intenta explicar lo que se ha hecho, será más fácil de entender si se ha seguido un estilo consistente y similar, o igual, al de otros programadores.
- Casi nunca ocurre que todo el código que utiliza para un programa sea escrito por un solo programador, sino que se habrá desarrollado por un grupo de programadores.

# Estructura de una Clase

Cada archivo fuente contendrá un comentario inicial donde se describirá el objetivo de dicho archivo.

Sentencia de paquete, si existe.

Importación de las clases que se utilizan en esta clase.

Declaración de clases e interfaces, que deberán aparecer en el siguiente orden:

- Comentario de documentación de la clase o interfaz.
- Sentencia de declaración de la clase o interfaz.
- Atributos estáticos, primero los públicos, después los protegidos, después los de paquete y por último los privados.

- Resto de los atributos, primero los públicos, después los protegidos, después los de paquete y por último los privados.
- Constructores.
- Resto de los métodos. Se sugiere agrupar los métodos por funcionalidad, no por su accesibilidad (públicos, privados, etc.). De esta forma se trata de mejorar la legibilidad de los mismos y encontrarlos mejor dentro de la clase.

# Modificadores de Acceso

Son las palabras clave como **`public`** o **`private`** que aparecen al comienzo de las declaraciones de campos y de las signaturas de los métodos.

Definen la visibilidad de:

- Un campo.
- Un constructor.
- Un método.

| Visibilidad | public | protected | default | private |
|---|---|---|---|---|
| UML | + | # | ~ | – |
| Desde la misma clase | SI | SI | SI | SI |
| Desde cualquier clase del mismo paquete | SI | SI | SI | NO |
| Desde una subclase del mismo paquete | SI | SI | SI | NO |
| Desde una subclase fuera del mismo paquete | SI | SI Herencia | NO | NO |
| Desde cualquier clase fuera del paquete | SI | NO | NO | NO |

# Visibilidad de los Campos

Los campos pueden ser privados o públicos.

***La declaración de los campos como públicos rompe con el principio de ocultamiento de la información.***

Hace que una clase que depende de esa información sea vulnerable a operaciones incorrectas, si se modifica la implementación.

Una razón más para mantener los campos como privados reside en que permiten que un objeto crezca manteniendo el control sobre su estado.

Si el acceso a los campos privados se canaliza a través de métodos **set** y **get**, entonces un objeto tiene la habilidad de asegurar que el campo nunca se configura con un valor que resulte inconsistente con su estado.

Este nivel de integridad no es posible si los campos son públicos.

# Visibilidad de los Métodos

En general los métodos de las clases son públicos, aunque no siempre es así.

A veces existen métodos privados de apoyo dentro de una clase que son usados por otros métodos de dicha clase.

# Métodos get y set

Son la forma de acceder a atributos de una clase.

Generalmente, se usan con atributos privados, ya que a los públicos se puede acceder directamente sin tener que acudir a ellos.

# Método get

Lo usamos para obtener el valor de un atributo.

El motivo es por los atributos que son privados.

```java
/**
* Devuelve el nombre del empleado.
* @return nombre del empleado.
*/
public String getNombre(){
  return nombre;
}
```

# Métodos set

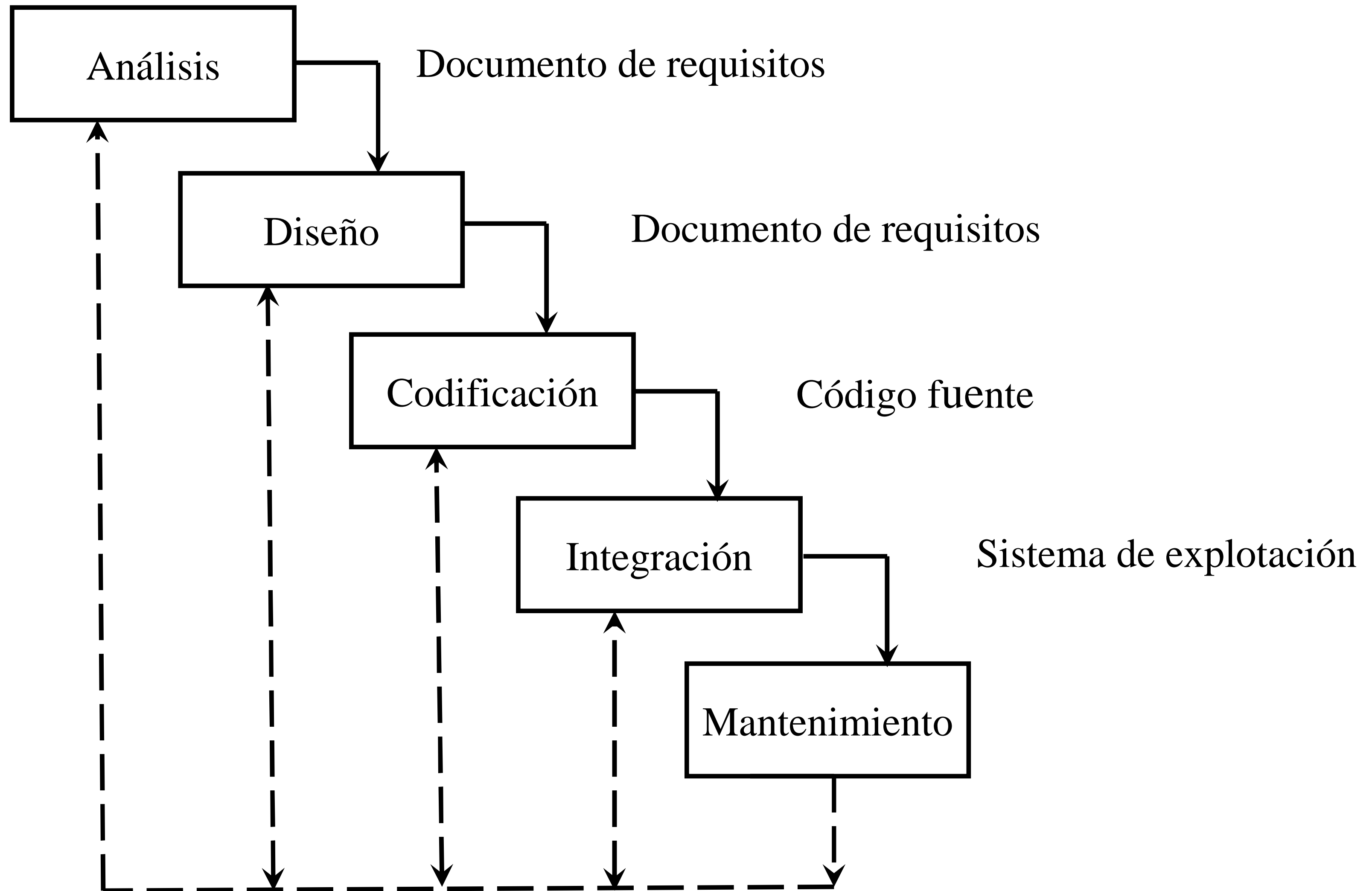
Lo usamos para modificar el valor de un atributo.

La cabecera tiene:

- Un tipo de retorno **`void`.**
- Un solo parámetro formal, el nuevo valor del campo a modificar.

```java
/**
* Modifica el nombre de un empleado.
* @param nombre.
*/
public void setNombre(String nombre){
    this.nombre = nombre;
}
```

# Ciclo de Vida en Cascada



# Acoplamiento y Cohesión

Hay dos términos que son fundamentales cuando hablamos sobre la calidad de un diseño de clases:

- Acoplamiento.
- Cohesión.

# Acoplamiento

Describe la interconexión de las clases.

Debemos lograr:

- Un acoplamiento débil o bajo.
- Que cada clase sea altamente independiente.
- Se comunique con otras clases mediante una interfaz compacta y bien definida.

El grado de acoplamiento determina el grado de dificultad de realizar modificaciones en una aplicación.

En una estructura de clases fuertemente acopladas, un cambio en una clase hace necesario también cambiar otras varias clases.

Este hecho es el que tratamos de evitar porque el efecto de hacer un pequeño cambio puede rápidamente propagarse a la aplicación completa.

Además, encontrar todos los lugares en que resulta necesario hacer los cambios y realmente llevar a cabo estos cambios puede ser dificultoso y consumir demasiado tiempo.

Por otro lado, en un ***sistema débilmente acoplado***, podemos con frecuencia modificar una clase sin tener que realizar cambios en ninguna otra y la aplicación continúa funcionando.

# Cohesión

Describe cuánto se ajusta una ***unidad de código*** a una tarea lógica concreta.

En un sistema altamente cohesivo cada ***unidad de código*** es responsable de una tarea bien definida.

Un diseño de clases de buena calidad exhibe un alto grado de cohesión.

La razón principal que subyace al principio de cohesión es la ***reutilización***.

Si un método de una clase es responsable de una única cosa bien definida es más probable que pueda ser usado nuevamente en un contexto diferente.

Una ventaja complementaria es que cuando se requiere un cambio de un aspecto de una aplicación, probablemente encontremos todas las piezas de código relevantes ubicadas en la misma unidad.

# Tipos Enumerados

Un enumerado o Enum es una clase “especial” que limita la creación de objetos a los especificados explícitamente en la implementación de la clase.

La única limitación que tienen los enumerados respecto a una clase normal es que si tiene constructor, este debe de ser privado para que no se puedan crear nuevos objetos.

# toString( )

El método **toString** nos permite mostrar la información completa de un objeto, es decir, el valor de sus **atributos.**

Este método también se hereda de **java.lang.Object.**

Por lo que deberemos sobrescribir este **método**.

# printf

Imprime un mensaje por pantalla utilizando una cadena de formato que incluye las instrucciones para mezclar múltiples cadenas en la cadena final a mostrar por pantalla.

Es una función especial porque recibe un número variable de parámetros.

El primer parámetro es fijo y es la cadena de formato.

En ella se incluye texto a imprimir literalmente y *marcas* a reemplazar por texto que se obtiene de los parámetros adicionales.

Se llama con tantos parámetros como marcas haya en la cadena de formato más uno, la propia cadena de formato.

El siguiente ejemplo muestra cómo se imprime el valor de la variable **`contador`**.

```
printf("El valor es %d.\n", contador);
```

El símbolo **`%`** denota el comienzo de la marca de formato.

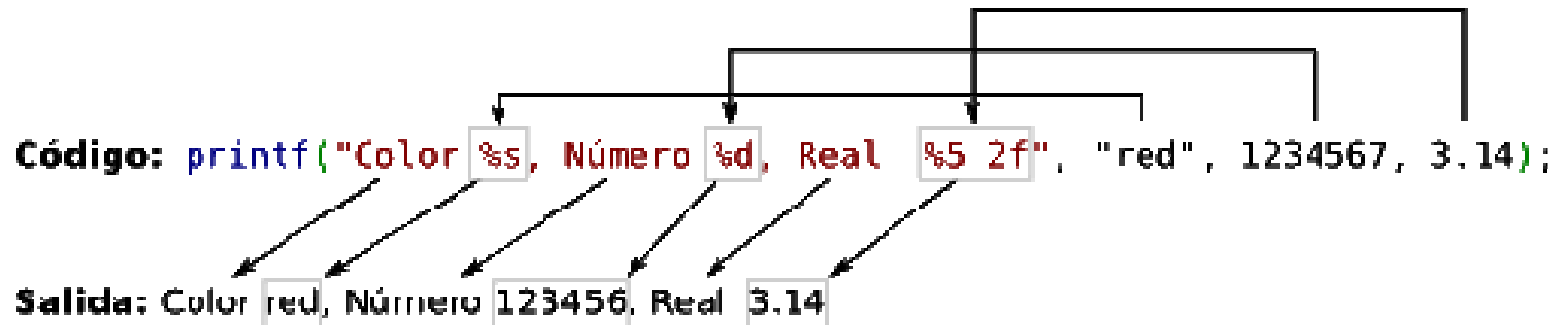
La marca **`%d`** se reemplaza por el valor de la variable **`contador`** y se imprime la cadena resultante.

El símbolo `\n` representa un salto de línea.

La salida, por defecto, se justifica a la derecha del ancho total que le hayamos dado al campo, que por defecto tiene como longitud la longitud de la cadena.

Si en la cadena de formato aparecen varias marcas, los valores a incluir se toman en el mismo orden en el que aparecen.

La siguiente figura muestra un ejemplo en el que la cadena de formato tiene tres marcas, **`%s`**, **`%d`** y **`%5.2f`**, que se procesan utilizando respectivamente la cadena “red”, el entero **`1234567`** y el número real **`3.14`**.



Las marcas en la cadena de formato deben tener la siguiente estructura.

Los campos entre corchetes son optativos.

```
%[parameter][flags][width][.precision][length]type
```

Toda marca comienza por el símbolo **%** y termina con su tipo.

Cada uno de los nombres:

- *parameter*
- *flags*
- *width*
- *precision*
- *length*
- *type*

Representa un conjunto de valores posibles que se explican a continuación.

| Parameter | Descripción |
| --- | --- |
| `n$` | Se reemplaza “n” por un número para cambiar el orden en el que se procesan los argumentos.<br>Por ejemplo `%3$d` se refiere al tercer argumento independientemente del lugar que ocupa en la cadena de formato. |

| Flags | Descripción |
| --- | --- |
| número | Rellena con espacios o con ceros, ver siguiente flag a la izquierda hasta el valor del número. |
| 0 | Se rellena con ceros a la izquierda hasta el valor dado por el flag anterior. Por ejemplo `%03d` imprime un número justificado con ceros hasta tres dígitos. |
| + | Imprimir el signo de un número |
| - | Justifica el campo a la izquierda (por defecto ya hemos dicho que se justifica a la derecha) |
| # | Formato alternativo. Para reales se dejan ceros al final y se imprime siempre la coma. Para números que no están en base 10, se añade un prefijo denotando la base. |

| Width | Descripción |
|---|---|
| número | Tamaño del ancho del campo donde se imprimirá el valor. |
| * | Igual que el caso anterior, pero el número a utilizar se pasa como parámetro justo antes del valor. Por ejemplo printf("%*d", 5, 10) imprime el número 10, pero con un ancho de cinco dígitos (es decir, rellenará con 3 espacios en blanco a la izquierda). |

| Precision | Descripción |
|---|---|
| número | Tamaño de la parte decimal para números reales.<br>Número de caracteres a imprimir para cadenas de texto |
| * | Igual que el caso anterior, pero el número a utilizar se pasa como parámetro justo antes del valor.<br>`printf("%.*s", 3, "abcdef"`<br>imprime “abc”. |

| Length | Descripción |
|---|---|
| hh | Convertir variable de tipo **`char`** a entero e imprimir |
| h | Convertir variable de tipo **`short`** a entero e imprimir |
| l | Para enteros, se espera una variable de tipo **`long`**. |
| ll | Para enteros, se espera una variable de tipo **`long long`**. |
| L | Para reales, se espera una variable de tipo **`long double`**. |
| z | Para enteros, se espera un argumento de tipo **`size_t.`** |

| Type | Descripción |
|---|---|
| **%c** | Imprime el carácter ASCII correspondiente. |
| **%d**, **%i** | Conversión decimal con signo de un entero. |
| **%x**, **%X** | Conversión hexadecimal sin signo. |
| **%p** | Dirección de memoria (puntero). |
| **%e**, **%E** | Conversión a coma flotante con signo en notación científica. |
| **%f**, **%F** | Conversión a coma flotante con signo, usando punto decimal |
| **%g**, **%G** | Conversión a coma flotante, usando la notación que requiera menor espacio. |
| **%o** | Conversión octal sin signo de un entero. |
| **%u** | Conversión decimal sin signo de un entero. |
| **%s** | Cadena de caracteres (terminada en '\0'). |
| **%%** | Imprime el símbolo %. |

# Palabras Clave static y final

En los programas en alguna ocasión utilizaremos constantes.

Son valores matemáticos como el número Pi, o valores propios de programa que nunca cambian.

Lo correcto será declararlos como constantes en lugar de declararlos como variables.

Supongamos que queremos usar una constante como el número Pi y que usamos esta declaración:

```
private double PI = 3.1416;
```

Podemos invocar el atributo PI para que te muestre el valor por pantalla.

Pero una declaración de este tipo presenta varios problemas.

En primer lugar, lo declarado no funciona realmente como constante, sino como variable con un valor inicial.

En segundo lugar, cada vez que creamos un objeto estamos usando un espacio de memoria para almacenar el valor 3.1416.

Si tuviéramos diez objetos, tendríamos diez espacios de memoria ocupados con la misma información, lo cual resulta ineficiente.

Para resolver estos problemas, podemos declarar constantes en Java usando esta sintaxis:

```
// Sintaxis:

private static double PI = 3.1416;
```

# static

Los atributos miembros de una clase pueden ser atributos de clase o atributos de instancia.

Son atributos de clase si se usa la palabra clave **`static`**.

En ese caso la variable es única para todas las instancias (objetos) de la clase.

Ocupando un único lugar en memoria.

A las variables de clase se les llama variables estáticas.

Si no se usa **`static`**, el sistema crea un lugar nuevo para esa variable con cada instancia.

La variable es diferente para cada objeto.

En el caso de una constante no tiene sentido crear un nuevo lugar de memoria por cada objeto de una clase que se cree.

Cuando usamos **`static final`** se dice que creamos una constante de clase.

Un atributo común a todos los objetos de esa clase.

# final

Indica que una variable es de tipo constante.

No admitirá cambios después de su declaración y asignación de valor.

Determina que un atributo no puede ser sobreescrito o redefinido.

No funcionará como una variable “tradicional”, sino como una constante.

Toda constante declarada con `final` ha de ser inicializada en el mismo momento de declararla.

`final` también se usa como palabra clave en otro contexto.

Una clase `final` es aquella que no puede tener clases que la hereden.

Por estilo y para una mayor claridad en el código usaremos letras mayúsculas en los identificadores.

```
private static final double PI=3.1416;
```

Un atributo de clase lo tenemos que declarar obligatoriamente en la cabecera de clase.

Junto a los campos, debajo de la signatura de clase.

Si tratamos de incorporarlo en un método obtendremos un error.

Por otro lado, **`final`** sí puede ser usado dentro de métodos y también dentro de un método **`main`**.

En la cabecera de clase usaremos **`static final`** para definir aquellas variables comunes a todos los objetos de una clase.

```java
public class Constantes{

   public static final float PI = 3.141592f;
   public static final float E = 2.728281f;

   public static void main(String[] args){
     System.out.println("PI = " + Constantes.PI);
     System.out.println("E = " + Constantes.E);
   }
}
```