

# Lenguaje de Programación Java. Primeros pasos.



## Objetivos

En esta unidad empezaremos a utilizar el lenguaje de programación que se va a emplear para desarrollar todos los contenidos de este módulo profesional, el lenguaje Java. Tras describir sus particularidades, veremos cómo crear, compilar y ejecutar programas escritos en Java, de manera básica.

Veremos cómo crear variables, los tipos de datos y operadores. Aún no se ha visto el concepto de Programación Orientada a Objetos, pero se podrán realizar pequeños programas utilizando objetos simples, algo necesario para poder poner en práctica los contenidos de la unidad.

También estudiaremos las estructuras de control de flujo que Java pone a disposición del programador. Una vez vistas estas estructuras, se afronta el estudio de las estructuras de salto incorporadas en Java que pueden ser necesarias en determinadas circunstancias.

Utilizaremos el IDE (Entorno Integrado de Desarrollo) BlueJ para implementar nuestros programas.

# 1.- El lenguaje de programación Java.

Java es un lenguaje sencillo de aprender, con una sintaxis parecida a la de C++, pero en la que se han eliminado elementos complicados y que pueden originar errores. Java es orientado a objetos, con lo que elimina muchas preocupaciones al programador y permite la utilización de gran cantidad de bibliotecas ya definidas, evitando reescribir código que ya existe. Es un lenguaje de programación creado para satisfacer nuevas necesidades que los lenguajes existentes hasta el momento no eran capaces de solventar.

Una de las principales virtudes de Java es su independencia del hardware, ya que el código que se genera es válido para cualquier plataforma. Este código será ejecutado sobre una máquina virtual denominada **Maquina Virtual Java** ( JVM – Java Virtual Machine), que interpretará el código convirtiéndolo a código específico de la plataforma que lo soporta. De este modo el programa se escribe una única vez y puede hacerse funcionar en cualquier lugar. Lema del lenguaje: “**Write once, run everywhere**”.



Antes de que apareciera Java, el lenguaje C era uno de los más extendidos por su versatilidad. Pero cuando los programas escritos en C aumentaban de volumen, su manejo comenzaba a complicarse. Mediante las técnicas de programación estructurada y programación modular se conseguían reducir estas complicaciones, pero no era suficiente.

Fue entonces cuando la Programación Orientada a Objetos, POO, entra en escena, aproximando notablemente la construcción de programas al pensamiento humano y haciendo más sencillo todo el proceso. Los problemas se dividen en objetos que tienen propiedades e interactúan con otros objetos, de este modo, el programador puede centrarse en cada objeto para programar internamente los elementos y funciones que lo componen.

Las características principales de lenguaje Java se resumen a continuación:

- ✓ El código generado por el compilador Java es independiente de la arquitectura.
- ✓ Está totalmente orientado a objetos.
- ✓ Su sintaxis es similar a C y C++.
- ✓ Es distribuido, preparado para aplicaciones TCP/IP
- ✓ Dispone de un amplio conjunto de bibliotecas.
- ✓ Es robusto, realizando comprobaciones del código en tiempo de compilación y de ejecución.
- ✓ La seguridad está garantizada, ya que las aplicaciones Java no acceden a zonas delicadas de memoria o del sistema.



## Debes conocer

Puedes obtener una descripción detallada de las características reseñadas anteriormente a través del siguiente artículo:

[Características detalladas del lenguaje Java](#)

## 1.1.- Breve historia.

Java surgió en 1991 cuando un grupo de ingenieros de Sun Microsystems trataron de diseñar un nuevo lenguaje de programación destinado a programar pequeños dispositivos electrónicos. La dificultad de estos dispositivos es que cambian continuamente y para que un programa funcione en el siguiente dispositivo aparecido, hay que rescribir el código. Por eso la empresa Sun quería crear un lenguaje **independiente del dispositivo**.

Pero no fue hasta 1995 cuando pasó a llamarse **Java**, dándose a conocer al público como lenguaje de programación para computadores. Java pasa a ser un lenguaje totalmente independiente de la plataforma y a la vez potente y orientado a objetos. Esa filosofía y su facilidad para crear aplicaciones para redes TCP/IP ha hecho que sea uno de los lenguajes más utilizados en la actualidad.

El factor determinante para su expansión fue la incorporación de un intérprete Java en la versión 2.0 del navegador Web Netscape Navigator, lo que supuso una gran revuelo en Internet. A principios de 1997 apareció **Java 1.1** que proporcionó sustanciales mejoras al lenguaje. **Java 1.2**, más tarde rebautizado como **Java 2**, nació a finales de 1998.

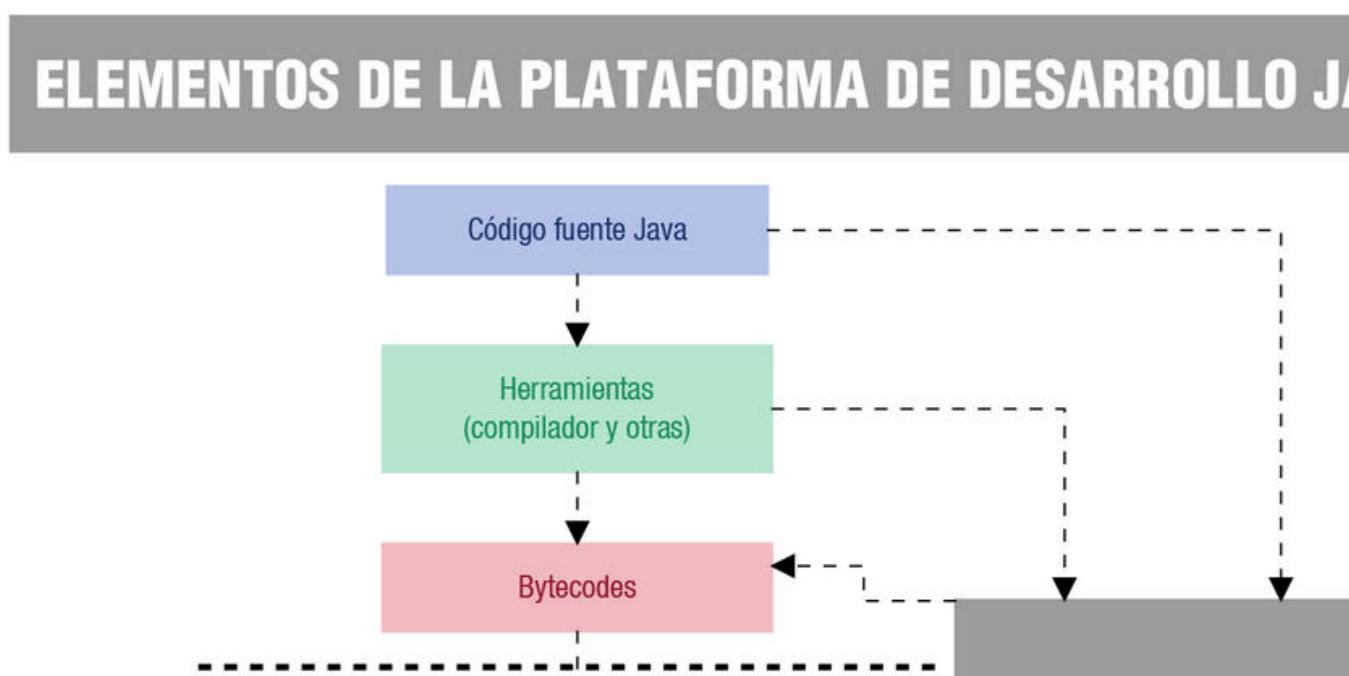
El principal objetivo del lenguaje Java es llegar a ser el **nexo universal** que conecte a los usuarios con la información, esté ésta situada en el ordenador local, en un servidor Web, en una base de datos o en cualquier otro lugar.

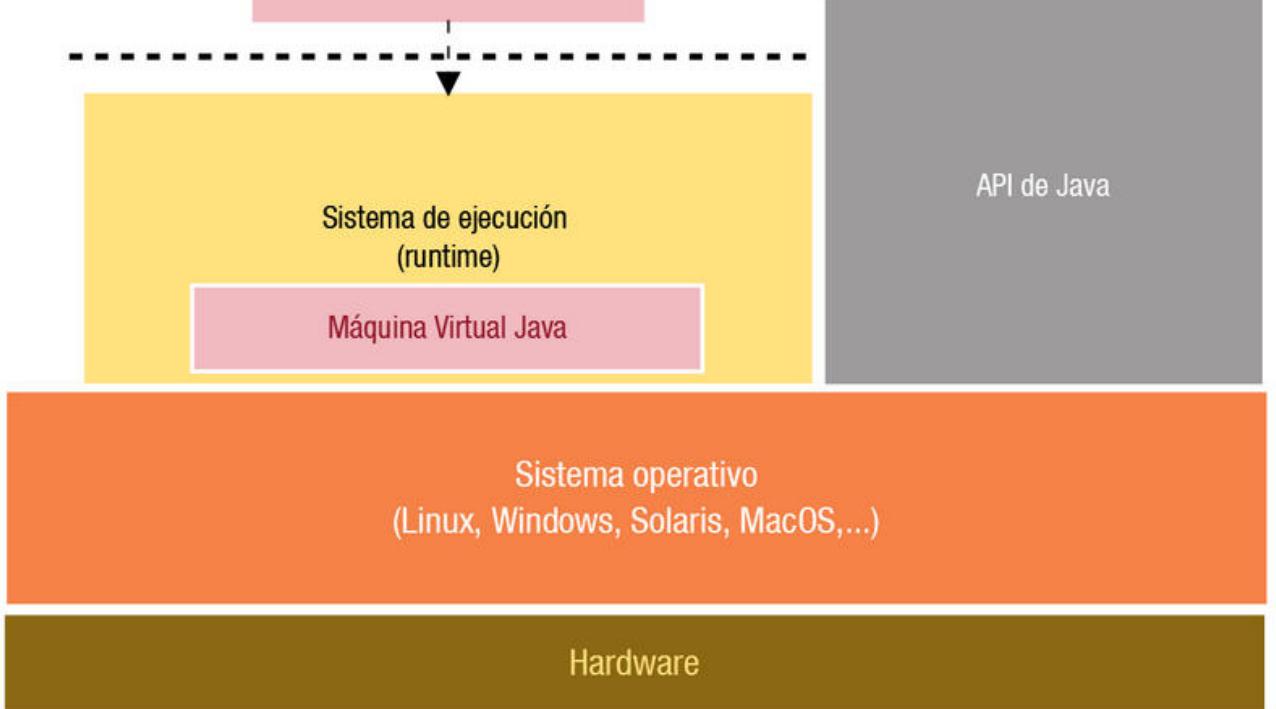
Para el desarrollo de programas en lenguaje Java es necesario utilizar un entorno de desarrollo denominado **JDK** (Java Development Kit), que provee de un compilador y un entorno de ejecución (JRE – Java Run Environment) para los bytecodes generados a partir del código fuente. Al igual que las diferentes versiones del lenguaje han incorporado mejoras, el entorno de desarrollo y ejecución también ha sido mejorado sucesivamente.

**Java 2** es la tercera versión del lenguaje, pero es algo más que un lenguaje de programación, incluye los siguientes elementos:

- ✓ Un lenguaje de programación: Java.
- ✓ Un conjunto de bibliotecas estándar que vienen incluidas en la plataforma y que son necesarias en todo entorno Java. Es el Java Core.
- ✓ Un conjunto de herramientas para el desarrollo de programas, como es el compilador de bytecodes, el generador de documentación, un depurador, etc.
- ✓ Un entorno de ejecución que en definitiva es una máquina virtual que ejecuta los programas traducidos a bytecodes.

El siguiente esquema muestra los elementos fundamentales de la plataforma de desarrollo Java 2.





## Para saber más

Si deseas conocer más sobre los orígenes del lenguaje Java, aquí te ofrecemos más información:

[Historia de Java](#)

[Línea de tiempo de la historia de Java](#)

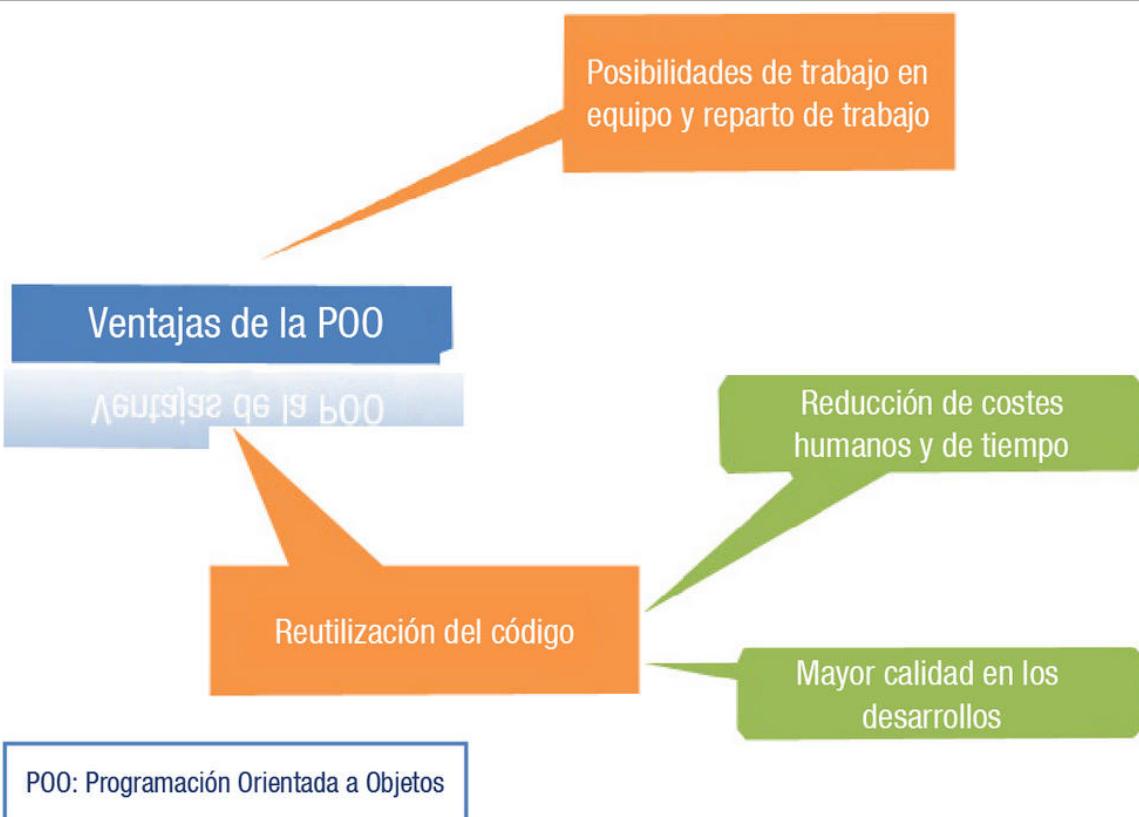
## 1.2.- La POO y Java.

En Java, los datos y el código (funciones o métodos) se combinan en entidades llamadas **objetos**. El objeto tendrá un comportamiento (su código interno) y un estado (los datos). Los objetos permiten la reutilización del código y pueden considerarse, en sí mismos, como piezas reutilizables en múltiples proyectos distintos. Esta característica permite reducir el tiempo de desarrollo de software.

Por simplificar un poco las cosas, un programa en Java será como una representación teatral en la que debemos preparar primero cada personaje, definir sus características y qué va a saber hacer. Cuando esta fase esté terminada, la obra se desarrollará sacando personajes a escena y haciéndoles interactuar.

Al emplear los conceptos de la Programación Orientada a Objetos (POO), Java incorpora las tres características propias de este paradigma: **encapsulación, herencia y polimorfismo**. Los patrones o tipos de objetos se denominan **clases** y los objetos que utilizan estos patrones o pertenecen a dichos tipos, se identifican con el nombre de **instancias**. Estos conceptos se verán más adelante en sucesivas unidades.

### VENTAJAS DE LA PROGRAMACIÓN ORIENTADA A OBJETOS



Otro ejemplo para seguir aclarando ideas, piensa en los bloques de juegos de construcción. Suponemos que conoces los cubos de plástico en varios colores y tamaños. Por una de sus caras disponen de pequeños conectores circulares y en otra de sus caras pequeños orificios en los que pueden conectarse otros bloques, con el objetivo principal de permitir construir formas más grandes. Si usas diferentes piezas del lego puedes construir aviones, coches, edificios, etc. Si te fijas bien, cada pieza es un objeto pequeño que puede unirse con otros objetos para crear objetos más grandes.

Pues bien, aproximadamente así es como funciona la programación orientada a objetos: unimos elementos pequeños para construir otros más grandes. Nuestros programas estarán formados por muchos componentes (objetos) independientes y diferentes; cada uno con una función determinada en nuestro software y que podrá comunicarse con los demás de una manera predefinida.

## 1.3.- Independencia de la plataforma y trabajo en red.

Existen dos características que distinguen a **Java** de otros lenguajes, como son la **independencia de la plataforma** y la posibilidad de trabajar en red o, mejor, la posibilidad de **crear aplicaciones que trabajan en red**.

Estas características las vamos a explicar a continuación:

- a. **Independencia:** Los programas escritos en Java pueden ser ejecutados en cualquier tipo de hardware. El código fuente es compilado, generándose el código conocido como **Java Bytecode** (instrucciones máquina simplificadas que son específicas de la plataforma Java), el bytecode será interpretado y ejecutado en la **Máquina Virtual Java (JVM)** que es un programa escrito en código nativo de la plataforma destino entendible por el hardware. Con esto se evita tener que realizar un programa diferente para cada plataforma.

Por tanto, la parte que realmente es dependiente del sistema es la Máquina Virtual Java, así como las librerías o bibliotecas básicas que permiten acceder directamente al hardware de la máquina.

- b. **Trabajo en red:** Esta capacidad del lenguaje ofrece múltiples posibilidades para la comunicación vía TCP/IP. Para poder hacerlo existen librerías que permiten el acceso y la interacción con protocolos como HTTP, FTP, etc., facilitando al programador las tareas del tratamiento de la información a través de redes.



### Autoevaluación

¿Qué elemento es imprescindible para que una aplicación escrita en Java pueda ejecutarse en un ordenador?

- Que disponga de conexión a Internet y del hardware adecuado.
- Que tenga instalado un navegador web y conexión a Internet.
- Que tenga la Máquina Virtual Java adecuada instalada.

## 1.4.- Seguridad y simplicidad.

Junto a las características diferenciadoras del lenguaje Java relacionadas con la independencia y el trabajo en red, han de destacarse dos virtudes que hacen a este lenguaje uno de los más extendidos entre la comunidad de programadores: su seguridad y su simplicidad.

a. **Seguridad:** En primer lugar, los posibles accesos a zonas de memoria “sensibles” que en otros lenguajes como C y C++ podían suponer peligros importantes, se han eliminado en Java.



En segundo lugar, el código Java es comprobado y verificado para evitar que determinadas secciones del código produzcan efectos no deseados. Los test que se aplican garantizan que las operaciones, operandos, conversiones, uso de clases y demás acciones son seguras.

Y en tercer lugar, Java no permite la apertura de ficheros en la máquina local, tampoco permite ejecutar ninguna aplicación nativa de una plataforma e impide que se utilicen otros ordenadores como puente, es decir, nadie puede utilizar nuestra máquina para hacer peticiones o realizar operaciones con otra.

En definitiva, podemos afirmar que Java es un lenguaje seguro.

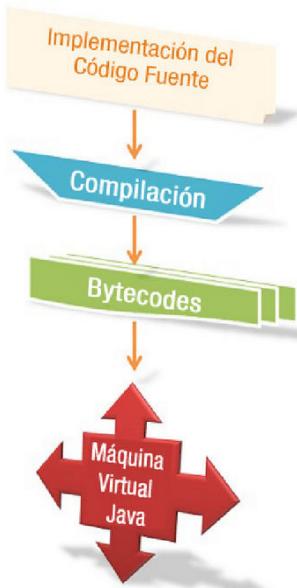
b. **Simplicidad:** Aunque Java es tan potente como C o C++, es bastante más sencillo. Posee una curva de aprendizaje muy rápida y, para alguien que comienza a programar en este lenguaje, le resulta relativamente fácil comenzar a escribir aplicaciones interesantes.

Muy relacionado con la simplicidad que aporta Java está la incorporación de un elemento muy útil como es el **Recolector de Basura (Garbage collector)**. Permite al programador liberarse de la gestión de la memoria y hace que ciertos bloques de memoria puedan reaprovecharse, disminuyendo el número de huecos libres (fragmentación de memoria).

Cuando realicemos programas, crearemos objetos, haremos que éstos interaccionen, etc. Todas estas operaciones requieren de uso de memoria del sistema, pero la gestión de ésta será realizada de manera transparente al programador. Todo lo contrario que ocurría en otros lenguajes. Podremos crear tantos objetos como solicitemos, pero nunca tendremos que destruirlos. El entorno de Java borrará los objetos cuando determine que no se van a utilizar más. Este proceso es conocido como recolección de basura.

## 1.5.- Java y los Bytecodes.

Un programa escrito en Java no es directamente ejecutable, es necesario que el código fuente sea interpretado por la Maquina Virtual Java. ¿Cuáles son los pasos que se siguen desde que se genera el código fuente hasta que se ejecuta? A continuación se detallan cada uno de ellos.



Una vez escrito el código fuente (archivos con extensión .Java), éste es precompilado generándose los códigos de bytes, Bytecodes o Java Bytecodes (archivos con extensión .class) que serán interpretados directamente por la Maquina Virtual Java y traducidos a código nativo de la plataforma sobre la que se esté ejecutando el programa.

**Bytecode:** Son un conjunto de instrucciones en lenguaje máquina que no son específicas a ningún procesador o sistema de cómputo. Un intérprete de código de bytes (bytecodes) para una plataforma específica será quien los ejecute. A estos intérpretes también se les conoce como Máquinas Virtuales Java o intérpretes Java de tiempo de ejecución.

En el proceso de precompilación, existe un verificador de códigos de bytes que se asegurará de que se cumplen las siguientes condiciones:

- ✓ El código satisface las especificaciones de la Máquina Virtual Java.
- ✓ No existe amenaza contra la integridad del sistema.
- ✓ No se producen desbordamientos de memoria.
- ✓ Los parámetros y sus tipos son adecuados.
- ✓ No existen conversiones de datos no permitidas.

Para que un bytecode pueda ser ejecutado en cualquier plataforma, es imprescindible que dicha plataforma cuente con el intérprete adecuado, es decir, la máquina virtual específica para esa plataforma. En general, la Máquina Virtual Java es un programa de reducido tamaño y gratuito para todos los sistemas operativos.

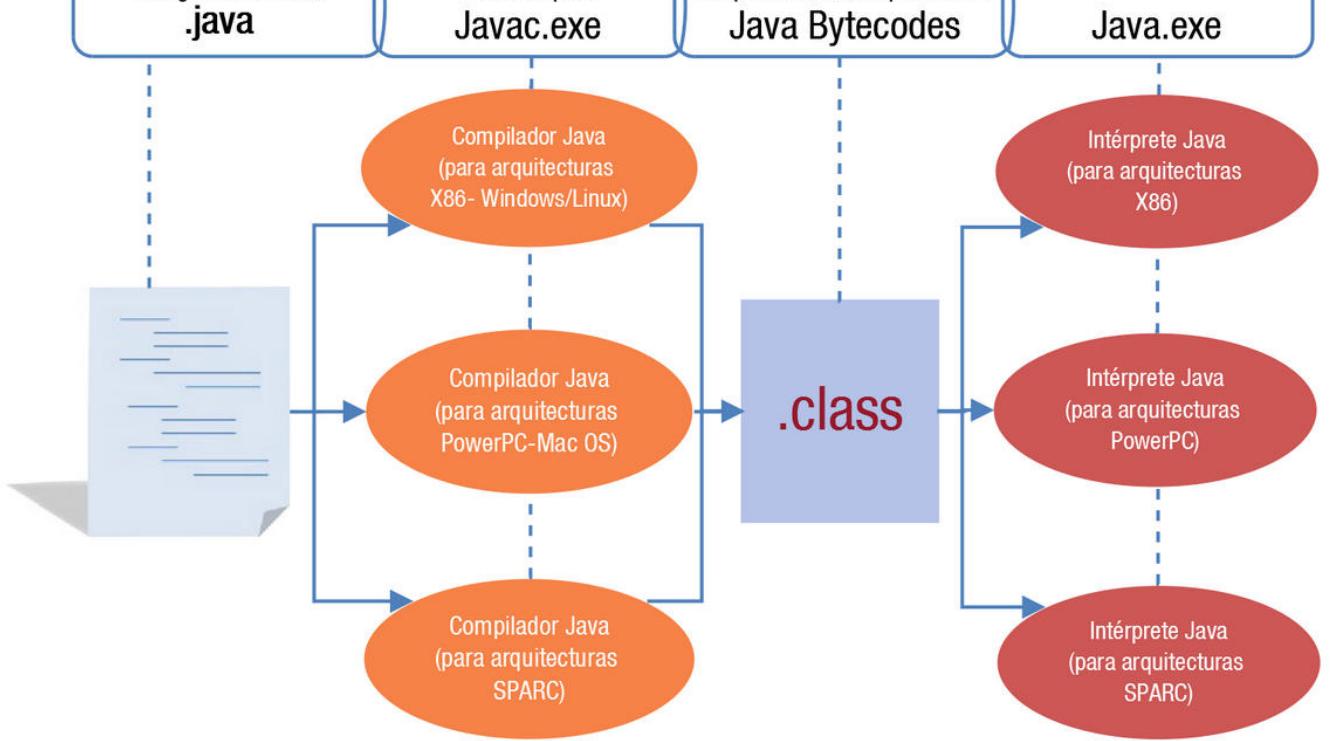
### BYTECODES EN JAVA

Código Fuente Java  
**.java**

Para compilar  
**Javac.exe**

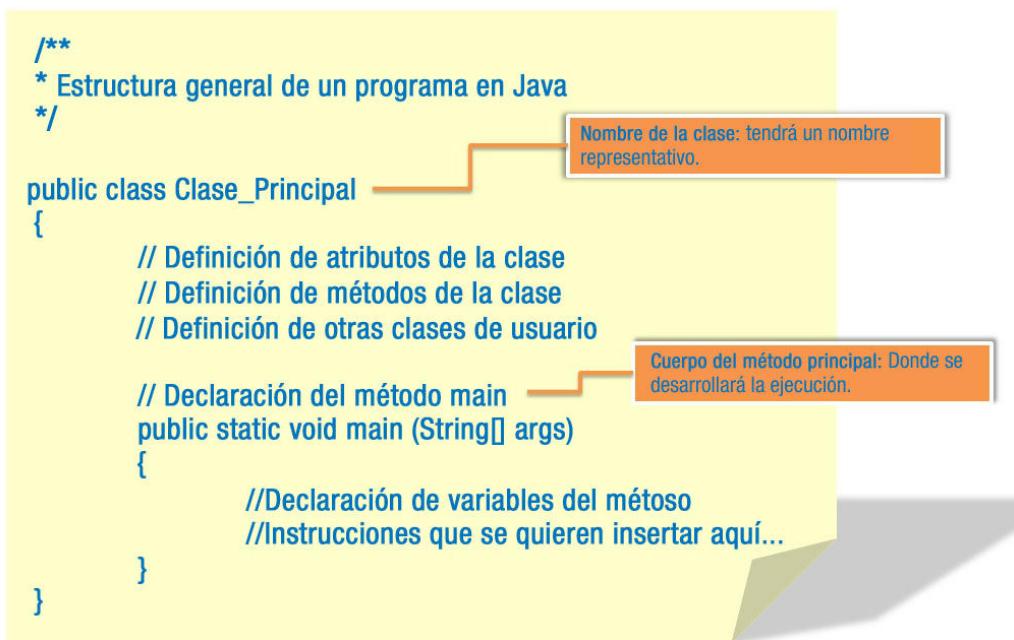
Independencia de la plataforma  
**Java Bytecodes**

**Java.exe**



## 2.- Estructura de un programa.

A continuación se presenta la estructura general de un programa realizado en Java.



Vamos a analizar cada uno de los elementos que aparecen en dicho gráfico:

- ✓ **public class Clase\_Principal:** Todos los programas han de incluir una clase como esta. Es una clase general en la que se incluyen todos los demás elementos del programa. Entre otras cosas, contiene el método o función `main()` que representa al programa principal, desde el que se llevará a cabo la ejecución del programa. Esta clase puede contener a su vez otras clases del usuario, pero sólo una puede ser `public`. El nombre del fichero `.Java` que contiene el código fuente de nuestro programa, coincidirá con el nombre de la clase que estamos describiendo en estas líneas.
- ✓ **public static void main (String[] args):** Es el método que representa al programa principal, en él se podrán incluir las instrucciones que estimemos oportunas para la ejecución del programa. Desde él se podrá hacer uso del resto de clases creadas. Todos los programas Java tienen un método `main`.
- ✓ **Comentarios:** Los comentarios se suelen incluir en el código fuente para realizar aclaraciones, anotaciones o cualquier otra indicación que el programador estime oportuna. Estos comentarios pueden introducirse de dos formas, `con // y con /* */`. Con la primera forma estaríamos estableciendo una única línea completa de comentario y, con la segunda, con `/*` comenzaríamos el comentario y éste no terminaría hasta que no insertáramos `*/`.
- ✓ **Bloques de código:** son conjuntos de instrucciones que se marcan mediante la apertura y cierre de llaves `{ }`. El código así marcado es considerado interno al bloque.
- ✓ **Punto y coma:** aunque en el ejemplo no hemos incluido ninguna línea de código que termine con punto y coma, hay que hacer hincapié en que cada línea de código ha de terminar con punto y coma `(;)`. En caso de no hacerlo, tendremos errores sintácticos.



### Recomendación

Ten en cuenta que **Java distingue entre mayúsculas y minúsculas**. Si le das a la clase

principal el nombre PrimerPrograma, el archivo .Java tendrá como identificador **PrimerPrograma.Java**, que es totalmente diferente a primerprograma.Java. Además, para Java los elementos PrimerPrograma y primerprograma serían considerados dos clases diferentes dentro del código fuente.



## Autoevaluación

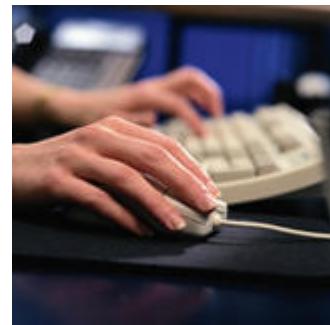
`public static void main (String[] args)` es la clase general del programa.

- Verdadero  Falso

## 2.1.- El entorno básico de desarrollo Java.

Ya conoces cómo es la estructura de un programa en Java, pero, ¿qué necesitamos para llevarlo a la práctica? La herramienta básica para empezar a desarrollar aplicaciones en Java es el **JDK (Java Development Kit o Kit de Desarrollo Java)**, que incluye un compilador y un intérprete para línea de comandos. Estos dos programas son los empleados en la precompilación e interpretación del código.

Como veremos, existen diferentes entornos para la creación de programas en Java que incluyen multitud de herramientas, pero por ahora nos centraremos en el entorno más básico, extendido y gratuito, el Java Development Kit (JDK). Según se indica en la propia página web de Oracle, JDK es un entorno de desarrollo para construir aplicaciones, applets y componentes utilizando el lenguaje de programación Java. Incluye herramientas útiles para el desarrollo y prueba de programas escritos en Java y ejecutados en la Plataforma Java.



Así mismo, junto a JDK se incluye una implementación del entorno de ejecución Java, el **JRE (Java Runtime Environment)** para ser utilizado por el JDK. El JRE incluye la Máquina Virtual de Java (JVM o MVJ – Java Virtual Machine), bibliotecas de clases y otros ficheros que soportan la ejecución de programas escritos en el lenguaje de programación Java.

Para que podamos compilar y ejecutar ficheros Java es necesario que realicemos unos pequeños ajustes en la configuración del sistema. Vamos a indicarle dónde encontrar los ficheros necesarios para realizar las labores de compilación y ejecución, en este caso **Javac.exe** y **Java.exe**, así como las librerías contenidas en la API de Java y las clases del usuario.

**La variable PATH:** Como aún no disponemos de un IDE (Integrated Development Environment - Entorno Integrado de Desarrollo) la única forma de ejecutar programas es a través de línea de comandos. Pero sólo podremos ejecutar programas directamente si la ruta hacia ellos está indicada en la variable PATH del ordenador. Es necesario que incluyamos la ruta hacia estos programas en nuestra variable PATH. Esta ruta será el lugar donde se instaló el JDK hasta su directorio **bin**.

**La variable CLASSPATH:** esta variable de entorno establece dónde buscar las clases o bibliotecas de la API de Java, así como las clases creadas por el usuario. Es decir, los ficheros **.class** que se obtienen una vez compilado el código fuente de un programa escrito en Java. Es posible que en dicha ruta existan directorios y ficheros comprimidos en los formatos **zip** o **jar** que pueden ser utilizados directamente por el JDK, contenido en su interior archivos con extensión **class**.

## Instalación y configuración

En este enlace tienes las indicaciones para descargar el JDK, instalarlo y configurar las variables del sistema PATH y CLASSPATH, que se han dado en el módulo de Entornos de Desarrollo

- [Instalación JDK en Windows 7 \(Ventana nueva\)](#)

Para poder desarrollar nuestros primeros programas en Java sólo necesitaremos un editor de texto plano y los elementos que acabamos de instalar.



## Autoevaluación

Podemos desarrollar programas escritos en Java mediante un editor de textos y a través del JRE podremos ejecutarlos.

- Verdadero
- Falso



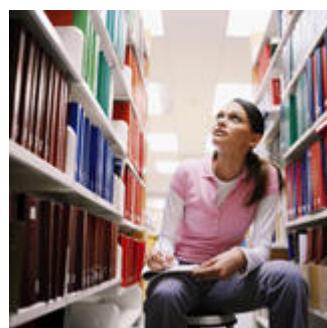
## Autoevaluación

¿Qué variable de sistema o de entorno debemos configurar correctamente para que podamos compilar directamente desde la línea de comandos nuestros programas escritos en lenguaje Java?

- CLASSPATH.
- PATH.
- Javac.exe.

## 2.2.- La API de Java.

Junto con el kit de desarrollo que hemos descargado e instalado anteriormente, vienen incluidas gratuitamente todas las bibliotecas de la API (Application Programming Interface – Interfaz de programación de aplicaciones) de Java, es lo que se conoce como Bibliotecas de Clases Java. Este conjunto de bibliotecas proporciona al programador paquetes de clases útiles para la realización de múltiples tareas dentro de un programa. Está organizada en paquetes lógicos, donde cada paquete contiene un conjunto de clases relacionadas semánticamente.



Una biblioteca de clases es un conjunto de clases de programación orientada a objetos. Esas clases contienen métodos que son útiles para los programadores. En el caso de Java cuando descargamos el JDK obtenemos la biblioteca de clases API. Utilizar las clases y métodos de las APIs de Java reduce el tiempo de desarrollo de los programas. También, existen diversas bibliotecas de clases desarrolladas por terceros que contienen componentes reutilizables de software, y están disponibles a través de la Web.



### Para saber más

Si quieres acceder a la información oficial sobre la API de Java, te proponemos el siguiente enlace (está en Inglés).

[Información oficial sobre la API de Java](#)



### Autoevaluación

Indica qué no es la API de Java:

- Un entorno integrado de desarrollo.
- Un conjunto de bibliotecas de clases.
- Una parte del JDK, incluido en el Java SE.

## 2.3.- Codificación, compilación y ejecución de aplicaciones.

Una vez que la configuración del entorno Java está completada y tenemos el código fuente de nuestro programa escrito en un archivo con extensión .Java, la compilación de aplicaciones se realiza mediante el programa **Javac** incluido en el software de desarrollo de Java.

Para llevar a cabo la compilación desde la línea de comandos, escribiremos:

**Javac archivo.Java**

Donde **Javac** es el compilador de Java y **archivo.Java** es nuestro código fuente.

El resultado de la compilación será un archivo con el mismo nombre que el archivo Java pero con la **extensión class**. Esto ya es el archivo con el código en forma de bytecode. Es decir con el código precompilado. Si en el código fuente de nuestro programa figuraran más de una clase, veremos como al realizar la compilación se generarán tantos archivos con extensión .class como clases tengamos. Además, si estas clases tenían método **main** podremos ejecutar dichos archivos por separado para ver el funcionamiento de dichas clases.

Para que el programa pueda ser ejecutado, siempre y cuando esté incluido en su interior el método **main**, podremos utilizar el intérprete incluido en el kit de desarrollo.

La ejecución de nuestro programa desde la línea de comandos podremos hacerla escribiendo:

**Java archivo.class**

Donde **Java** es el intérprete y **archivo.class** es el archivo con el código precompilado.



### Ejercicio resuelto

Vamos a llevar a la práctica todo lo que hemos estado detallando a través de la creación, compilación y ejecución de un programa sencillo escrito en Java.

Observa el código que se muestra más abajo, seguro que podrás entender parte de él. Cópialo en un editor de texto, respetando las mayúsculas y las minúsculas. Puedes guardar el archivo con extensión .Java en la ubicación que prefieras. Recuerda que el nombre de la clase principal (en el código de ejemplo **MiModulo**) debe ser exactamente igual al del archivo con extensión .Java, si tienes esto en cuenta la aplicación podrá ser compilada correctamente y ejecutada.

```
/**  
 * La clase MiModulo implementa una aplicación que  
 * simplemente imprime "Módulo profesional - Programación" en pantalla.  
 */  
class MiModulo {  
  
    public static void main(String[] args) {  
        System.out.println("Módulo profesional - Programación"); // Muestra la cadena de caracteres.  
    }  
}
```

Accede a la línea de comandos y teclea, en la carpeta donde has guardado el archivo Java, el comando **para compilarlo**: `javac MiModulo.java`

El compilador genera entonces un fichero de código de bytes: `MiModulo.class`. Si visualizas ahora el contenido de la carpeta verás que en ella está el archivo `.java` y uno o varios (depende de las clases que contenga el archivo con el código fuente) archivos `.class`.

Finalmente, **para realizar la ejecución** del programa debes utilizar la siguiente sentencia:  
`java MiModulo.java`

Si todo ha ido bien, verás escrito en pantalla: "Módulo profesional – Programación".

### 3.- Tipos de aplicaciones en Java.

La versatilidad del lenguaje de programación Java permite al programador crear distintos tipos de aplicaciones. A continuación, describiremos las características más relevantes de cada uno de ellos:

#### ✓ Aplicaciones de consola:

- ◆ Son programas independientes al igual que los creados con los lenguajes tradicionales.
- ◆ Se componen como mínimo de un archivo `.class` que debe contar necesariamente con el método `main`.
- ◆ No necesitan un navegador web y se ejecutan cuando invocamos el comando Java para iniciar la Máquina Virtual de Java (JVM). De no encontrarse el método `main` la aplicación no podrá ejecutarse.
- ◆ Las aplicaciones de consola leen y escriben hacia y desde la entrada y salida estándar, sin ninguna interfaz gráfica de usuario.

#### ✓ Aplicaciones gráficas:

- ◆ Aquellas que utilizan las clases con capacidades gráficas, como Swing que es la biblioteca para la interfaz gráfica de usuario avanzada de la plataforma Java SE.
- ◆ Incluyen las instrucciones `import`, que indican al compilador de Java que las clases del paquete `Javax.swing` se incluyan en la compilación.

#### ✓ Applets:

- ◆ Son programas incrustados en otras aplicaciones, normalmente una página web que se muestra en un navegador. Cuando el navegador carga una web que contiene un applet, éste se descarga en el navegador web y comienza a ejecutarse. Esto nos permite crear programas que cualquier usuario puede ejecutar con tan solo cargar la página web en su navegador.
- ◆ Se pueden descargar de Internet y se observan en un navegador. Los applets se descargan junto con una página HTML desde un servidor web y se ejecutan en la máquina cliente.
- ◆ No tienen acceso a partes sensibles (por ejemplo: no pueden escribir archivos), a menos que uno mismo le dé los permisos necesarios en el sistema.
- ◆ No tienen un método principal.
- ◆ Son multiplataforma y pueden ejecutarse en cualquier navegador que soporte Java.

#### ✓ Servlets:

- ◆ Son componentes de la parte del servidor de Java EE, encargados de generar respuestas a las peticiones recibidas de los clientes.
- ◆ Los servlets, al contrario de los applets, son programas que están pensados para trabajar en el lado del servidor y desarrollar aplicaciones Web que interactúen con los clientes.

#### ✓ Midlets:

- ◆ Son aplicaciones creadas en Java para su ejecución en sistemas de propósito simple o dispositivos móviles. Los juegos Java creados para teléfonos móviles son midlets.
- ◆ Son programas creados para dispositivos embebidos (se dedican a una sola actividad), más específicamente para la máquina virtual Java MicroEdition (Java ME).
- ◆ Generalmente son juegos y aplicaciones que se ejecutan en teléfonos móviles.



### Autoevaluación

Un Applet es totalmente seguro ya que no puede acceder, en ningún caso, a zonas sensibles del sistema. Es decir, no podría borrar o modificar nuestros archivos.

- Verdadero
- Falso

## 4.- Entornos Integrados de Desarrollo (IDE).

En los comienzos de Java la utilización de la línea de comandos era algo habitual. El programador escribía el código fuente empleando un editor de texto básico, seguidamente, pasaba a utilizar un compilador y con él obtenía el código compilado. En un paso posterior, necesitaba emplear una tercera herramienta para el ensamblado del programa. Por último, podía probar a través de la línea de comandos el archivo ejecutable. El problema surgía cuando se producía algún error, lo que provocaba tener que volver a iniciar el proceso completo.

Los Entornos Integrados de Desarrollo son aplicaciones que ofrecen la posibilidad de llevar a cabo el proceso completo de desarrollo de software a través de un único programa. Podremos realizar las labores de edición, compilación, depuración, detección de errores, corrección y ejecución de programas escritos en Java o en otros lenguajes de programación, bajo un entorno gráfico (no mediante línea de comandos). Junto a las capacidades descritas, cada entorno añade otras que ayudan a realizar el proceso de programación, como por ejemplo: código fuente coloreado, plantillas para diferentes tipos de aplicaciones, creación de proyectos, etc.

Hay que tener en cuenta que un entorno de desarrollo no es más que una fachada para el proceso de compilación y ejecución de un programa. ¿Qué quiere decir eso? Pues que si tenemos instalado un IDE y no tenemos instalado el compilador, no tenemos nada.



### Para saber más

Si deseas conocer algo más sobre lo que son los Entornos Integrados de Desarrollo (IDE).

[Definición de Entorno Integrado de Desarrollo en Wikipedia](#)

## 4.1.- IDE's actuales.

Existen en el mercado multitud de entornos de desarrollo para el lenguaje Java, los hay de libre distribución, de pago, para principiantes, para profesionales, que consumen más recursos, que son más ligeros, más amigables, más complejos que otros, etc.

Entre los que son gratuitos o de libre distribución tenemos:

- ✓ **NetBeans**
- ✓ **Eclipse**
- ✓ **BlueJ**
- ✓ **Jgrasp**
- ✓ **Jcreator LE**

Entre los que son propietarios o de pago tenemos:

- ✓ **IntelliJ IDEA**
- ✓ **Jbuilder**
- ✓ **Jcreator**
- ✓ **JDeveloper**



### Debes conocer

Cada uno de los entornos nombrados más arriba posee características que los hacen diferentes unos de otros, pero para tener una idea general de la versatilidad y potencia de cada uno de ellos, accede a la siguiente tabla comparativa:

[Comparativa entornos para Java](#)

Pero, ¿cuál o cuáles son los más utilizados por la comunidad de programadores Java? El puesto de honor se lo disputan entre **Eclipse**, **IntelliJ IDEA** y **NetBeans**.



## 4.2.- Instalación y configuración de BlueJ.

BlueJ es un sencillo entorno integrado de desarrollo (IDE) exclusivamente diseñado para la enseñanza y el aprendizaje de Java. Se trata de un proyecto nacido en el seno de un grupo de investigación universitario integrado por miembros británicos y australianos.

Para realizar la instalación del entorno BlueJ, seguiremos los siguientes pasos básicos:

1. Descarga de la versión deseada desde la web oficial [www.bluej.org](http://www.bluej.org)
2. Seleccionar la plataforma o sistema operativo, existen versiones para Windows, Linux, MacOS, etc. Tanto en Windows como en Linux, se descarga un archivo ejecutable que se encarga de la instalación.
3. Comenzará la descarga del archivo de instalación ejecutable y una vez finalizada, lanzar éste, comenzando la instalación en nuestro equipo.
4. Finalmente, la instalación se completa y dispondremos de este entorno totalmente operativo.



### Comenzando a trabajar con BlueJ

A continuación tienes un enlace al manual de uso de BlueJ que se ha utilizado en el módulo de Entornos de Desarrollo

- [Manual BlueJ \(Ventana nueva\)](#)

## 5.- Programación en Java.



### Para saber más

Dentro de la documentación de Oracle sobre Java SE se encuentra el libro “The Java Language Specification”. Este libro está escrito por los inventores del lenguaje, y constituye una referencia técnica casi obligada sobre el mismo. Como mucha de la documentación oficial de Java, se encuentra en inglés. El enlace directo es el siguiente:

 [The Java Language Specification](#)

A partir de ahora es conveniente que utilices algún manual de apoyo para iniciarte a la programación en Java. Te proponemos el de la serie de Libros “Aprenda Informática como si estuviera en primero”, de la Escuela Superior de Ingenieros Industriales de San Sebastián (Universidad de Navarra):

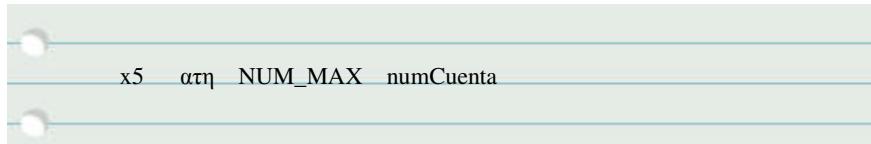
 [Manual de apoyo sobre Java](#)

## 5.1.- Variables e Identificadores.

Al nombre que le damos a la variable se le llama **identificador**. Los identificadores permiten nombrar los elementos que se están manejando en un programa.

Un **identificador** en Java es una secuencia ilimitada sin espacios de letras y dígitos UNICODE, de forma que **el primer símbolo de la secuencia debe ser una letra, un símbolo de subrayado (\_) o el símbolo dólar (\$)**.

Por ejemplo, son válidos los siguientes identificadores:



En la definición anterior decimos que un identificador es una secuencia ilimitada de caracteres Unicode. Pero... ¿qué es Unicode? Unicode es un código de caracteres o sistema de codificación, un alfabeto que recoge los caracteres de prácticamente todos los idiomas importantes del mundo. Las líneas de código en los programas se escriben usando ese conjunto de caracteres Unicode.

Esto quiere decir que en Java se pueden utilizar varios alfabetos como el Griego, Árabe o Japonés. De esta forma, los programas están más adaptados a los lenguajes e idiomas locales, por lo que son más significativos y fáciles de entender tanto para los programadores que escriben el código, como para los que posteriormente lo tienen que interpretar, para introducir alguna nueva funcionalidad o modificación en la aplicación.

El estándar Unicode originalmente utilizaba 16 bits, pudiendo representar hasta 65.536 caracteres distintos, que es el resultado de elevar dos a la potencia dieciséis. Actualmente Unicode puede utilizar más o menos bits, dependiendo del formato que se utilice: **UTF-8**, **UTF-16** ó **UTF-32**. A cada carácter le corresponde únicamente un número entero perteneciente al intervalo de **0 a 2 elevado a n**, siendo n el número de bits utilizados para representar los caracteres.

Por ejemplo, la letra ñ es el entero 164. Además, el código Unicode es “compatible” con el código ASCII, ya que para los caracteres del código ASCII, Unicode asigna como código los mismos 8 bits, a los que les añade a la izquierda otros 8 bits todos a cero. La conversión de un carácter ASCII a Unicode es inmediata.

Enlace para acceder a la documentación sobre las distintas versiones de Unicode en la página web oficial del estándar: [Documentación sobre Unicode](#)

## 5.1.1.- Convenios y reglas para nombrar variables.

A la hora de nombrar un identificador existen una serie de normas de estilo de uso generalizado que, no siendo obligatorias, se usan en la mayor parte del código Java. Estas reglas para la nomenclatura de variables son las siguientes:

- ✓ **Java distingue las mayúsculas de las minúsculas.** Por ejemplo, `Alumno` y `alumno` son variables diferentes.
- ✓ **No se suelen utilizar identificadores que comiencen con «\$» o «\_»,** además el símbolo del dólar, por convenio, no se utiliza nunca.
- ✓ **No se puede utilizar el valor booleano (true o false) ni el valor nulo (null).**
- ✓ **Los identificadores deben ser lo más descriptivos posibles.** Es mejor usar palabras completas en vez de abreviaturas crípticas. Así nuestro código será más fácil de leer y comprender. En muchos casos también hará que nuestro código se autodocumente. Por ejemplo, si tenemos que darle el nombre a una variable que almacena los datos de un cliente sería recomendable que la misma se llamara algo así como `FicheroClientes` o `ManejadorCliente`, y no algo poco descriptivo como `Cl33`.

Además de estas restricciones, en la siguiente tabla puedes ver otras convenciones, que no siendo obligatorias, sí son recomendables a la hora de crear identificadores en Java.

**Convenciones sobre identificadores en Java**

Identificador	Convención	Ejemplo
Nombre de variable	Comienza por letra minúscula, y si tienen más de una palabra se colocan juntas y el resto comenzando por mayúsculas	numAlumnos, suma
Nombre constante	En letras mayúsculas, separando las palabras con el guión bajo, por convenio el guión bajo no se utiliza en ningún otro sitio	TAM_MAX, PI
Nombre de una clase	Comienza por letra mayúscula	String, MiTipo
Nombre función	Comienza con letra minúscula	modificaValor, obtieneValor



### Autoevaluación

Un identificador es una secuencia de uno o más símbolos Unicode que cumple las siguientes condiciones. Señala la afirmación correcta.

- Todos los identificadores han de comenzar con una letra, el carácter subrayado (`_`) o el carácter dólar (`$`).
- No puede incluir el carácter espacio en blanco.
- Puede tener cualquier longitud, no hay tamaño máximo.

- Todas las anteriores son correctas.

## 5.1.2.- Palabras reservadas.

Las palabras reservadas, a veces también llamadas palabras clave o keywords , son secuencias de caracteres formadas con letras ASCII cuyo uso se reserva al lenguaje y, por tanto, **no pueden utilizarse para crear identificadores**.

Las palabras reservadas en Java son:

**Palabras clave en Java**

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Hay palabras reservadas que no se utilizan en la actualidad, como es el caso de **const** y **goto**, que apenas se utilizan en la actual implementación del lenguaje Java.

Por otro lado, puede haber otro tipo de palabras o texto en el lenguaje que aunque no sean palabras reservadas tampoco se pueden utilizar para crear identificadores.

- ✓ Es el caso de **true** y **false** que, aunque puedan parecer palabras reservadas, porque no se pueden utilizar para ningún otro uso en un programa, técnicamente son **literales booleanos**.
- ✓ Igualmente, **null** es considerado un literal, no una palabra reservada.

Normalmente, los editores y entornos integrados de desarrollo utilizan colores para diferenciar las palabras reservadas del resto del código, los comentarios, las constantes y literales, etc. De esta forma se facilita la lectura del programa y la detección de errores de sintaxis. Dependiendo de la configuración del entorno se utilizarán unos colores u otros.

## 5.1.3.- Tipos de variables.

En un programa nos podemos encontrar distintos tipos de variables. Las diferencias entre una variable y otra dependerán de varios factores, por ejemplo, el tipo de datos que representan, si su valor cambia o no a lo largo de todo el programa, o cuál es el papel que llevan a cabo en el programa. De esta forma, el lenguaje de programación Java define los siguientes tipos de variables:

- a. **Variables de tipos primitivos y variables referencia**, según el tipo de información que contengan. En función de a qué grupo pertenezca la variable, tipos primitivos o tipos referenciados, podrá tomar unos valores u otros, y se podrán definir sobre ella unas operaciones u otras.
- b. **Variables y constantes**, dependiendo de si su valor cambia o no durante la ejecución del programa. La definición de cada tipo sería:
  - ✓ Variables. Sirven para almacenar los datos durante la ejecución del programa, pueden estar formadas por cualquier tipo de dato primitivo o referencia. Su valor puede cambiar varias veces a lo largo de todo el programa.
  - ✓ Constantes o variables finales. Son aquellas variables cuyo valor no cambia a lo largo de todo el programa.
- c. **Variables miembro y variables locales**, en función del lugar donde aparezcan en el programa. La definición concreta sería:
  - ✓ Variables miembro. Son las variables que se crean dentro de una clase, fuera de cualquier método. Pueden ser de tipos primitivos o referencias, variables o constantes. En un lenguaje puramente orientado a objetos como es Java, todo se basa en la utilización de objetos, los cuales se crean usando clases. En la siguiente unidad veremos los distintos tipos de variables miembro que se pueden usar.
  - ✓ Variables locales. Son las variables que se crean y usan dentro de un método o, en general, dentro de cualquier bloque de código. La variable deja de existir cuando la ejecución del bloque de código o el método finaliza. Al igual que las variables miembro, las variables locales también pueden ser de tipos primitivos o referencias.

```
/**  
 * Aplicación ejemplo de tipos de variables  
 */  
public class ejemplovariables {  
    final double PI = 3.1415926536; // PI es una constante  
    int x; // x es una variable miembro  
           // de clase ejemplovariables  
  
    int obtenerX(int x) { // x es un parámetro  
        int valorantiguo = this.x; // valorantiguo es una variable local  
        return valorantiguo;  
    }  
  
    // el método main comienza la ejecución de la aplicación  
    public static void main(String[] args) {  
        // aquí iría el código de nuestra aplicación  
  
    } // fin del método main  
}  
// fin de la clase ejemplovariables
```



### Autoevaluación

**Relaciona los tipos de variables con la característica correspondiente, escribiendo el número asociado a la característica en el hueco correspondiente.**

### Ejercicio de relacionar

Las variables...	Relación	Tienen la característica de que ...
Locales.	<input type="radio"/>	1. Una vez inicializadas su valor nunca cambia.
Miembro.	<input type="radio"/>	2. Van dentro de un método.
Constantes.	<input checked="" type="radio"/>	3. Van dentro de una clase.

**Enviar**

## 5.2.- Tipos de datos.

---

En los lenguajes fuertemente tipados, a todo dato (constante, variable o expresión) le corresponde un tipo que es conocido antes de que se ejecute el programa.

El tipo limita el valor de la variable o expresión, las operaciones que se pueden hacer sobre esos valores, y el significado de esas operaciones. Esto es así porque **un tipo de dato no es más que una especificación de los valores que son válidos para esa variable , y de las operaciones que se pueden realizar con ellos.**

Debido a que el tipo de dato de una variable se conoce durante la revisión que hace el compilador para detectar errores, o sea en tiempo de compilación, esta labor es mucho más fácil, ya que no hay que esperar a que se ejecute el programa para saber qué valores va a contener esa variable. Esto se consigue con un control muy exhaustivo de los tipos de datos que se utilizan, lo cual tiene sus ventajas e inconvenientes, por ejemplo cuando se intenta asignar un valor de un tipo, a una variable de otro tipo. Sin embargo, en Java, puede haber conversión entre ciertos tipos de datos, como veremos posteriormente.

Ahora no es el momento de entrar en detalle sobre la conversión de tipos, pero sí debemos conocer con exactitud de qué tipos de datos dispone el lenguaje Java. Ya hemos visto que las variables, según la información que contienen, se pueden dividir en variables de tipos primitivos y variables referencia. Pero ¿qué es un tipo de dato primitivo? ¿Y un tipo referencia? Esto es lo que vamos a ver a continuación. Los tipos de datos en Java se dividen principalmente en dos categorías:

- ✓ **Tipos de datos sencillos o primitivos.** Representan valores simples que vienen predefinidos en el lenguaje; contienen valores únicos, como por ejemplo un carácter o un número.
- ✓ **Tipos de datos referencia.** Se definen con un nombre o referencia (puntero) que contiene la dirección en memoria de un valor o grupo de valores. Dentro de este tipo tenemos por ejemplo los vectores o arrays, que son una serie de elementos del mismo tipo, o las clases, que son los modelos o plantillas a partir de los cuales se crean los objetos.

## 5.2.1.- Tipos de datos primitivos.

Los tipos primitivos son aquéllos datos sencillos que constituyen los tipos de información más habituales: números, caracteres y valores lógicos o booleanos. Al contrario que en otros lenguajes de programación orientados a objetos, **en Java no son objetos**.

Una de las mayores ventajas de tratar con tipos primitivos en lugar de con objetos, es que el compilador de Java puede optimizar mejor su uso. Otra importante característica, es que cada uno de los tipos primitivos tiene **idéntico** tamaño y comportamiento en todas las versiones de Java y para cualquier tipo de ordenador. Esto quiere decir que no debemos preocuparnos de cómo se representarán los datos en distintas plataformas, y asegura la **portabilidad** de los programas, a diferencia de lo que ocurre con otros lenguajes.

Tipo	Descripción	Bytes	Rango
<b>byte</b>	Entero muy corto	1	-128 a 127
<b>short</b>	Entero corto	2	-32,768 a 32,767
<b>int</b>	Entero	4	-2,147,483,648 a 2,147,483,647
<b>long</b>	Entero largo	8	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807
<b>float</b>	Número con punto flotante de precisión individual con hasta 7 dígitos significativos	4	+/-1.4E-45 (+/-1.4 times $10^{-45}$ ) a +/-3.4E38 (+/-3.4 times $10^{38}$ )
<b>double</b>	Número con punto flotante de precisión doble con hasta 16 dígitos significativos	8	+/-4.9E-324 (+/-4.9 times $10^{-324}$ ) a +/-1.7E308 (+/-1.7 times $10^{308}$ )
<b>char</b>	Carácter UNICODE	2	
<b>boolean</b>	Valor Verdadero o Falso	1	true o false

A la hora de elegir el tipo de dato que vamos a utilizar ¿qué criterio seguiremos para elegir un tipo de dato u otro? Pues deberemos tener en cuenta cómo es la información que hay que guardar, si es de tipo texto, numérico, ... y, además, qué rango de valores puede alcanzar. En este sentido, hay veces que aunque queramos representar un número sin decimales, tendremos que utilizar datos de tipo real.

Por ejemplo, el tipo de dato **int** no podría representar la población mundial del planeta, ya que el valor máximo que alcanza es de 2.147.483.647, siendo éste el número máximo de combinaciones posibles con 32 bits, teniendo en cuenta que la representación de los números enteros en Java utiliza complemento a 2. Si queremos representar el valor correspondiente a la población mundial del planeta, cerca de 7 mil millones de habitantes, tendríamos que utilizar al menos un tipo de dato **long**, o si tenemos problemas de espacio un tipo **float**.

La mayoría de los programadores en Java emplean el tipo **double** cuando trabajan con datos de tipo real, es una forma de asegurarse de que los errores cometidos por las sucesivas aproximaciones sean menores.

## 5.2.2.- Declaración e inicialización.

Llegados a este punto cabe preguntarnos ¿cómo se crean las variables en un programa? ¿Qué debo hacer antes de usar una variable en mi programa? Pues bien, como podrás imaginar, debemos crear las variables antes de poder utilizarlas en nuestros programas, indicando qué nombre va a tener y qué tipo de información va a almacenar, en definitiva, debemos **declarar la variable**.

Las variables se pueden declarar en cualquier bloque de código, dentro de llaves. Y lo hacemos indicando su identificador y el tipo de dato, separadas por comas si vamos a declarar varias a la vez, por ejemplo:

```
int numAlumnos = 15;  
double radio = 3.14, importe = 102.95;
```

De esta forma, estamos declarando `numAlumnos` como una variable de tipo `int`, y otras dos variables `radio` e `importe` de tipo `double`. Aunque no es obligatorio, hemos aprovechado la declaración de las variables para inicializarlas a los valores 15, 3.14 y 102.95 respectivamente.

Si la variable va a permanecer inalterable a lo largo del programa, la declararemos como `constante`, utilizando la palabra reservada `final` de la siguiente forma:

```
final double PI = 3.1415926536;
```

En ocasiones puede que al declarar una variable no le demos valor, ¿qué crees que ocurre en estos casos? Pues que el compilador le asigna un valor por defecto, aunque depende del tipo de variable que se trate:

- ✓ Las **variables miembro** sí se inicializan automáticamente, si no les damos un valor. Cuando son de tipo numérico, se inicializan por defecto a `0`, si son de tipo carácter, se inicializan al carácter `null` (`\0`), si son de tipo `boolean` se les asigna el valor por defecto `false`, y si son tipo referenciado se inicializan a `null`.
- ✓ Las **variables locales** no se inicializan automáticamente. Debemos asignarles nosotros un valor antes de ser usadas, ya que si el compilador detecta que la variable se usa antes de que se le asigne un valor, produce un error. Por ejemplo en este caso:

```
int p;  
int q = p; // error
```

Y también en este otro, ya que se intenta usar una variable local que podría no haberse inicializado:

```
int p;
```

```
if ( . . . )
    p = 5 ;
    int q = p; // error
```

En el ejemplo anterior la instrucción **if** hace que si se cumple la condición que hay entre paréntesis (cualquiera que indiquemos), entonces el programa asignará el valor 5 a la variable **p**; sino se cumple la condición, **p** quedará sin inicializar. Pero si **p** no se ha inicializado, no tendría valor para asignárselo a **q**. Por ello, el compilador detecta ese posible problema y produce un error del tipo “**La variable podría no haber sido inicializada**”, independientemente de si se cumple o no la condición del **if**.



## Autoevaluación

De las siguientes, señala cuál es la afirmación correcta:

- La declaración de una variable consiste en indicar el tipo que va a tener seguido del nombre y su valor
- Java no tiene restricción de tipos
- Todos los tipos tienen las mismas operaciones a realizar con ellos: suma, resta, multiplicación, etc.
- Todas las anteriores son incorrectas.

## 5.2.3.- Tipos referenciados.

A partir de los ocho tipos datos primitivos, se pueden construir otros tipos de datos. Estos tipos de datos se llaman **tipos referenciados** o **referencias**, porque se utilizan para almacenar la dirección de los datos en la memoria del ordenador.

```
int[] arrayDeEnteros;  
Cuenta cuentaCliente;
```

En la primera instrucción declaramos una lista de números del mismo tipo, en este caso, enteros. En la segunda instrucción estamos declarando la variable u objeto `cuentaCliente` como una referencia de tipo `Cuenta`.

Cualquier aplicación de hoy en día necesita no perder de vista una cierta cantidad de datos. Cuando el conjunto de datos utilizado tiene características similares se suelen agrupar en estructuras para facilitar el acceso a los mismos, son los llamados **datos estructurados**. Son datos estructurados los **arrays, listas, árboles**, etc. Pueden estar en la memoria del programa en ejecución, guardados en el disco como ficheros, o almacenados en una base de datos.

Además de los ocho tipos de datos primitivos que ya hemos descrito, Java proporciona un tratamiento especial a los textos o cadenas de caracteres mediante el tipo de dato `String`. Java crea automáticamente un nuevo objeto de tipo `String` cuando se encuentra una cadena de caracteres encerrada entre comillas dobles. En realidad se trata de objetos, y por tanto son tipos referenciados, pero se pueden utilizar de forma sencilla como si fueran variables de tipos primitivos:

```
String mensaje;  
mensaje= "El primer programa";
```

A continuación se muestra un ejemplo de declaración de distintos tipos de variables:

```
public class ejemplotipos {  
  
    // el método main inicia la ejecución de la aplicación  
    public static void main(String[] args) {  
        // Código de la aplicación  
        int i = 10;  
        double d = 3.14;  
        char c1 = 'a';  
        char c2 = 65;  
        boolean encontrado = true;  
        String msj = "Bienvenido a Java";  
  
        System.out.println("La variable i es de tipo entero y su valor es: " + i);  
        System.out.println("La variable f es de tipo double y su valor es: " + d);  
        System.out.println("La variable c1 es de tipo carácter y su valor es: " + c1);  
        System.out.println("La variable c2 es de tipo carácter y su valor es: " + c2);  
        System.out.println("La variable encontrado es de tipo booleano y su valor es: " + encontrado);  
        System.out.println("La variable msj es de tipo String y su valor es: " + msj);  
    } // fin del método main
```

```
 } // fin del método main  
 } // fin de la clase ejemplotipos|
```

Para mostrar por pantalla un mensaje utilizamos `System.out`, conocido como la salida estándar del programa. Este método lo que hace es escribir un conjunto de caracteres a través de la línea de comandos.

Podemos utilizar `System.out.print` o `System.out.println`. En el segundo caso lo que hace el método es que justo después de escribir el mensaje, sitúa el cursor al principio de la línea siguiente.

El texto en color gris que aparece entre caracteres // son comentarios que permiten documentar el código, pero no son tenidos en cuenta por el compilador y, por tanto, no afectan a la ejecución del programa.

## 5.2.4.- Tipos enumerados.

Los **tipos de datos enumerados** son una forma de declarar una variable con un conjunto restringido de valores. Por ejemplo, los días de la semana, las estaciones del año, los meses, etc. Es como si definiéramos nuestro propio tipo de datos.

La forma de declararlos es con la palabra reservada `enum`, seguida del nombre de la variable y la lista de valores que puede tomar entre llaves. A los valores que se colocan dentro de las llaves se les considera como constantes, van separados por comas y deben ser valores únicos.

La lista de valores se coloca entre llaves, porque un tipo de datos `enum` no es otra cosa que una especie de clase en Java, y todas las clases llevan su contenido entre llaves.

Al considerar Java este tipo de datos como si de una clase se tratara, no sólo podemos definir los valores de un tipo enumerado, sino que también podemos definir operaciones a realizar con él y otro tipo de elementos, lo que hace que este tipo de dato sea más versátil y potente que en otros lenguajes de programación.

En el siguiente ejemplo puedes comprobar el uso que se hace de los tipos de datos enumerados. Tenemos una variable `Dias` que almacena los días de la semana. Para acceder a cada elemento del tipo enumerado se utiliza el nombre de la variable seguido de un punto y el valor en la lista. Más tarde veremos que podemos añadir métodos y campos o variables en la declaración del tipo enumerado, ya que como hemos comentado un tipo enumerado en Java tiene el mismo tratamiento que las clases.

```
10  public class tiposenumerados {
11      public enum Dias {Lunes, Martes, Miercoles, Jueves, Viernes, Sabado, Domingo}
12
13      public static void main(String[] args) {
14          // código de la aplicación
15          Dias diaactual = Dias.Martes;
16          Dias diasiguiente = Dias.Miercoles;
17
18          System.out.print("Hoy es: ");
19          System.out.println(diaactual);
20          System.out.println("Mañana\nes\n"+diasiguiente);
21
22      } // fin main
23
24  } // fin tiposenumerados
```

En este ejemplo hemos utilizado el método `System.out.print`. Como podrás comprobar si lo ejecutas, la instrucción número 18 escribe el texto que tiene entre comillas pero no salta a la siguiente línea, por lo que el la instrucción número 19 escribe justo a continuación.

Sin embargo, también podemos escribir varias líneas usando una única sentencia. Así lo hacemos en la instrucción número 20, la cual imprime como resultado tres líneas de texto. Para ello hemos utilizado un carácter especial, llamado **carácter escape** (`\`). Este carácter sirve para darle ciertas órdenes al compilador, en lugar de que salga impreso en pantalla. Después del carácter de escape viene otro carácter que indica la orden a realizar, juntos reciben el nombre de **secuencia de escape**. La secuencia de escape `\n` recibe el nombre de **carácter de nueva línea**. Cada vez que el compilador se encuentra en un texto ese carácter, el resultado es que mueve el cursor al principio de la línea siguiente. En el próximo apartado vamos a ver algunas de las secuencias de escape más utilizadas.

## 5.3.- Literales de tipos primitivos.

Un **literal**, **valor literal** o **constante literal** es un valor concreto para los tipos de datos primitivos del lenguaje, el tipo `String` o el tipo `null`.

Los **literales booleanos** tienen dos únicos valores que puede aceptar el tipo: `true` y `false`. Por ejemplo, con la instrucción `boolean encontrado = true;` estamos declarando una variable de tipo booleana a la cual le asignamos el valor literal `true`.

Los **literales enteros** se pueden representar en tres notaciones:

- ✓ **Decimal**: por ejemplo `20`. Es la forma más común.
- ✓ **Octal**: por ejemplo `024`. Un número en octal siempre empieza por cero, seguido de dígitos octales (del 0 al 7).
- ✓ **Hexadecimal**: por ejemplo `0x14`. Un número en hexadecimal siempre empieza por `0x` seguido de dígitos hexadecimales (del 0 al 9, de la 'a' a la 'f' o de la 'A' a la 'F').

Las constantes literales de tipo `long` se le debe añadir detrás una `L` ó `L`, por ejemplo `873L`, si no se considera por defecto de tipo `int`. Se suele utilizar `L` para evitar la confusión de la ele minúscula con 1.

Los **literales reales** o en coma flotante se expresan con coma decimal o en notación científica, o sea, seguidos de un exponente `e` ó `E`. El valor puede finalizarse con una `f` o una `F` para indicar el formato `float` o con una `d` o una `D` para indicar el formato `double` (por defecto es `double`). Por ejemplo, podemos representar un mismo literal real de las siguientes formas: `13.2`, `13.2D`, `1.32e1`, `0.132E2`. Otras constantes literales reales son por ejemplo: `.54`, `31.21E-5`, `2.f`, `6.022137e+23f`, `3.141e-9d`.

Un **literal carácter** puede escribirse como un carácter entre comillas simples como `'a'`, `'ñ'`, `'Z'`, `'p'`, etc. o por su código de la tabla Unicode, anteponiendo la secuencia de escape `\` si el valor lo ponemos en octal o `\u` si ponemos el valor en hexadecimal. Por ejemplo, si sabemos que tanto en ASCII como en UNICODE, la letra A (mayúscula) es el símbolo número 65, y que 65 en octal es 101 y 41 en hexadecimal, podemos representar esta letra como `\101` en octal y `\u0041` en hexadecimal. Existen unos caracteres especiales que se representan utilizando secuencias de escape:

Secuencias de escape en Java

Secuencia de escape	Significado	Secuencia de escape	Significado
<code>\b</code>	Retroceso	<code>\r</code>	Retorno de carro
<code>\t</code>	Tabulador	<code>\\"</code>	Carácter comillas dobles
<code>\n</code>	Salto de línea	<code>\'</code>	Carácter comillas simples
<code>\f</code>	Salto de página	<code>\\\</code>	Barra diagonal

Normalmente, los objetos en Java deben ser creados con la orden `new`. Sin embargo, los **literales String** no lo necesitan ya que son objetos que se crean implícitamente por Java.

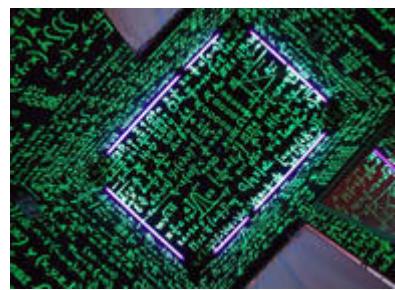
Los **literales de cadenas de caracteres** se indican entre comillas dobles. En el ejemplo anterior “El primer programa” es un literal de tipo cadena de caracteres. Al construir una cadena de caracteres se puede incluir cualquier carácter Unicode excepto un carácter de retorno de carro, por ejemplo en la siguiente instrucción utilizamos la secuencia de escape `\` para escribir dobles comillas dentro del mensaje:

```
String texto = "Juan dijo: \"Hoy hace un día fantástico...\"";
```

En el ejemplo anterior de tipos enumerados ya estábamos utilizando secuencias de escape, para introducir un salto de línea en una cadena de caracteres, utilizando el carácter especial \n.

## 5.4.- Operadores y expresiones

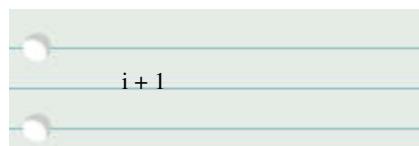
Los **operadores** llevan a cabo operaciones sobre un conjunto de datos u operandos, representados por literales y/o identificadores. Los operadores pueden ser unarios, binarios o terciarios, según el número de operandos que utilicen sean uno, dos o tres, respectivamente. Los operadores actúan sobre los tipos de datos primitivos y devuelven también un tipo de dato primitivo.



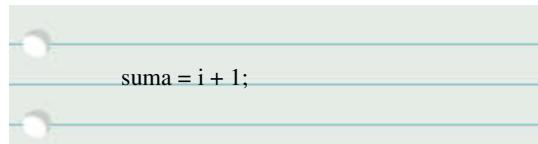
Los operadores se combinan con los literales y/o identificadores para formar expresiones. Una **expresión** es una combinación de operadores y operandos que se evalúa produciendo un único resultado de un tipo determinado.

El resultado de una expresión puede ser usado como parte de otra expresión o en una sentencia o instrucción. Las expresiones, combinadas con algunas palabras reservadas o por sí mismas, forman las llamadas **sentencias o instrucciones**.

Por ejemplo, pensemos en la siguiente expresión Java:



Con esta expresión estamos utilizando un operador aritmético para sumarle una cantidad a la variable *i*, pero es necesario indicar al programa qué hacer con el resultado de dicha expresión:



Que lo almacene en una variable llamada **suma**, por ejemplo. En este caso ya tendríamos una acción completa, es decir, una sentencia o instrucción.

Más ejemplos de sentencias o instrucciones los tenemos en las declaraciones de variables, vistas en apartados anteriores, o en las estructuras básicas del lenguaje como sentencias condicionales o bucles, que veremos en unidades posteriores.

Como curiosidad comentar que las expresiones de asignación, al poder ser usadas como parte de otras asignaciones u operaciones, son consideradas tanto expresiones en sí mismas como sentencias.

## 5.4.1.- Operadores aritméticos.

Los operadores aritméticos son aquellos operados que combinados con los operandos forman expresiones matemáticas o aritméticas.

### Operadores aritméticos básicos

Operador	Operación Java	Expresión Java	Resultado
-	Operador unario de cambio de signo	-10	-10
+	Adición	1.2 + 9.3	10.5
-	Sustracción	312.5 – 12.3	300.2
*	Multiplicación	1.7 * 1.2	1.02
/	División (entera o real)	0.5 / 0.2	2.5
%	Resto de la división entera	25 % 3	1

El resultado de este tipo de expresiones depende de los operandos que utilicen:

### Resultados de las operaciones aritméticas en Java

Tipo de los operandos	Resultado
Un operando de tipo <code>long</code> y ninguno real ( <code>float</code> o <code>double</code> )	<code>long</code>
Ningún operando de tipo <code>long</code> ni real ( <code>float</code> o <code>double</code> )	<code>int</code>
Al menos un operando de tipo <code>double</code>	<code>double</code>
Al menos un operando de tipo <code>float</code> y ninguno <code>double</code>	<code>float</code>

Otro tipo de operadores aritméticos son los operadores unarios incrementales y decrementales. Producen un resultado del mismo tipo que el operando, y podemos utilizarlos con **notación prefija**, si el operador aparece antes que el operando, o **notación postfija**, si el operador aparece después del operando. En la tabla puedes ver un ejemplo de utilización de cada operador incremental.

### Operadores incrementales en Java

Tipo operador	Expresión Java	
<code>++ (incremental)</code>	Prefija: <code>x=3;</code> <code>y=++x;</code> <code>// x vale 4 e y vale 4</code>	Postfija: <code>x=3;</code> <code>y=x++;</code> <code>// x vale 4 e y vale 3</code>
<code>--(decremental)</code>	<code>5-- // el resultado es 4</code>	

En el siguiente ejemplo vemos un programa básico que utiliza operadores aritméticos. Observa que usamos `System.out.printf` para mostrar por pantalla un texto formateado. El texto entre dobles comillas son

los argumentos del método `printf` y si usamos más de uno, se separan con comas. Primero indicamos cómo queremos que salga el texto, y después el texto que queremos mostrar. Fíjate que con el primer `%s` nos estamos refiriendo a una variable de tipo `String`, o sea, a la primera cadena de texto, con el siguiente `%s` a la segunda cadena y con el último `%s` a la tercera. Con `%f` nos referimos a un argumento de tipo `float`, etc.

```
public class operadoresaritmeticos {  
    public static void main(String[] args) {  
        short x = 7;  
        int y = 5;  
        float f1 = 13.5f;  
        float f2 = 8f;  
        System.out.println("El valor de x es " + x + ", y es " + y);  
        System.out.println("El resultado de x + y es " + (x + y));  
        System.out.println("El resultado de x - y es " + (x - y));  
        System.out.printf("%s\n%s%s\n", "División entera:", "x / y = ", (x/y));  
        System.out.println("Resto de la división entera: x % y = " + (x % y));  
        System.out.println("El valor de f1 es " + f1 + ", f2 es " + f2);  
        System.out.println("El resultado de f1 / f2 es " + (f1 / f2));  
    } // fin de main  
} // fin de la clase operadoresaritmeticos
```

```
El valor de x es 7, y es 5  
El resultado de x + y es 12  
El resultado de x - y es 2  
División entera:  
x / y = 1  
Resto de la división entera: x % y = 2  
El valor de f1 es 13.5, f2 es 8.0  
El resultado de f1 / f2 es 1.6875
```

## 5.4.2.- Operadores de asignación.

El principal operador de esta categoría es el operador asignación “=”, que permite al programa darle un valor a una variable, y ya hemos utilizado varias ocasiones en esta unidad. Además de este operador, Java proporciona otros operadores de asignación combinados con los operadores aritméticos, que permiten abreviar o reducir ciertas expresiones.

Por ejemplo, el operador “+=” suma el valor de la expresión a la derecha del operador con el valor de la variable que hay a la izquierda del operador, y almacena el resultado en dicha variable. En la siguiente tabla se muestran todos los operadores de asignación compuestos que podemos utilizar en Java

**Operadores de asignación combinados en Java**

Operador	Ejemplo en Java	Expresión equivalente
<code>+=</code>	<code>op1 += op2</code>	<code>op1 = op1 + op2</code>
<code>-=</code>	<code>op1 -= op2</code>	<code>op1 = op1 - op2</code>
<code>*=</code>	<code>op1 *= op2</code>	<code>op1 = op1 * op2</code>
<code>/=</code>	<code>op1 /= op2</code>	<code>op1 = op1 / op2</code>
<code>%=</code>	<code>op1 %= op2</code>	<code>op1 = op1 % op2</code>

Un ejemplo de operadores de asignación combinados lo tenemos a continuación:

```
public class operadoresasignacion {  
  
    // clase principal main que inicia la aplicación  
    public static void main(String[] args) {  
        int x;  
        int y;  
        x = 5; // operador asignación  
        y = 3; // operador asignación  
  
        //operadores de asignación combinados  
        System.out.printf("El valor de x es %d y el valor de y es %d\n", x,y);  
        x += y;  
        // podemos utilizar indistintamente printf o println  
        System.out.println(" Suma combinada: x += y " + " ..... x vale " + x);  
        x = 5;  
        x -= y;  
        System.out.println(" Resta combinada: x -= y " + " ..... x vale " + x);  
        x = 5;  
        x *= y;  
        System.out.println(" Producto combinado: x *= y " + " ..... x vale " + x);  
        x = 5;  
        x /= y;  
        System.out.println(" División combinada: x /= y " + " ..... x vale " + x);  
        x = 5;  
        x %= y;  
        System.out.println(" Resto combinada: x %= y " + " ..... x vale " + x);  
    } // fin main  
} // fin operadoresasignacion
```

```
El valor de x es 5 y el valor de y es 3
Suma combinada: x += y ..... x vale 8
Resta combinada: x -= y ..... x vale 2
Producto combinado: x *= y ..... x vale 15
Division combinada: x /= y ..... x vale 1
Resto combinada: x %= y ..... x vale 2
```



## Para saber más

En el siguiente enlace tienes información interesante sobre cómo se pueden utilizar los caracteres especiales incluidos en la orden `printf` (en inglés):

[Orden printf](#)

## 5.4.3.- Operador condicional.

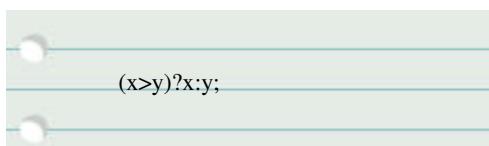
El operador condicional “? :” sirve para evaluar una condición y devolver un resultado en función de si es verdadera o falsa dicha condición. Es el único operador ternario de Java, y como tal, necesita tres operandos para formar una expresión.

El primer operando se sitúa a la izquierda del símbolo de interrogación, y siempre será una expresión booleana, también llamada condición. El siguiente operando se sitúa a la derecha del símbolo de interrogación y antes de los dos puntos, y es el valor que devolverá el operador condicional si la condición es verdadera. El último operando, que aparece después de los dos puntos, es la expresión cuyo resultado se devolverá si la condición evaluada es falsa.

### Operador condicional en Java

Operador	Expresión en Java
?:	condición ? exp1 : exp2

Por ejemplo, en la expresión:



Se evalúa la condición de si `x` es mayor que `y`, en caso afirmativo se devuelve el valor de la variable `x`, y en caso contrario se devuelve el valor de `y`.

El operador condicional se puede sustituir por la sentencia `if...then...else` que veremos en la siguiente unidad de Estructuras de control.

## 5.4.4.- Operadores de relación.

Los operadores relacionales se utilizan para comparar datos de tipo primitivo (numérico, carácter y booleano). El resultado se utilizará en otras expresiones o sentencias, que ejecutarán una acción u otra en función de si se cumple o no la relación.

Estas expresiones en Java dan siempre como resultado un valor booleano true o false. En la tabla siguiente aparecen los operadores relacionales en Java.

Operadores relacionales en Java

Operador	Ejemplo en Java	Significado
==	op1 == op2	op1 igual a op2
!=	op1 != op2	op1 distinto de op2
>	op1 > op2	op1 mayor que op2
<	op1 < op2	op1 menor que op2
>=	op1 >= op2	op1 mayor o igual que op2
<=	op1 <= op2	op1 menor o igual que op2

Hasta ahora hemos visto ejemplos que creaban variables y se inicializaban con algún valor. Pero ¿y si lo que queremos es que el usuario **introduzca** un valor al programa? Entonces debemos agregarle interactividad a nuestro programa, por ejemplo utilizando la clase **Scanner**. Aunque no hemos visto todavía qué son las clases y los objetos, de momento vamos a pensar que la clase **Scanner** nos va a permitir leer los datos que se escriben por teclado, y que para usarla es necesario importar el paquete de clases que la contiene. El código del ejemplo lo tienes a continuación. El programa se quedará esperando a que el usuario escriba algo en el teclado y pulse la tecla intro. En ese momento se convierte lo leído a un valor del tipo **int** y lo guarda en la variable indicada. Además de los operadores relacionales, en este ejemplo utilizamos también el operador condicional, que compara si los números son iguales. Si lo son, devuelve la cadena iguales y sino, la cadena distintos.

```
import java.util.Scanner;
//importamos el paquete necesario para poder usar la clase Scanner

public class ejemplorelacionales {
    // método principal que inicia la aplicación
    public static void main( String args[] )
    {
        // clase Scanner para petición de datos
        Scanner teclado = new Scanner( System.in );
        int x, y;
        String cadena;
        boolean resultado;
        System.out.print( "Introducir primer número: " );
        x = teclado.nextInt(); // pedimos el primer número al usuario
        System.out.print( "Introducir segundo número: " );
        y = teclado.nextInt(); // pedimos el segundo número al usuario
        // realizamos las comparaciones
        cadena=(x==y)?"iguales":"distintos";
        System.out.printf("Los números %d y %d son %s\n",x,y,cadena);
        resultado=(x!=y);
        System.out.println("x != y // es " + resultado);
        resultado=(x < y);
```

```
----- ~ ~ ~
System.out.println("x < y // es " + resultado);
resultado=(x > y );
System.out.println("x > y // es " + resultado);
resultado=(x <= y );
System.out.println("x <= y // es " + resultado);
resultado=(x >= y );
System.out.println("x >= y // es " + resultado);
} // fin método main
} // fin clase ejemplorelacionales
```



## Autoevaluación

Señala cuáles son los operadores relacionales en Java.

- ==, !=, >, <, >=, <=.
- =, !=, >, <, >=, <=.
- ==, !=, >, <, =>, =<.
- ==, !=, >, <, >=, <=.

## 5.4.5.- Operadores lógicos.

Los operadores lógicos realizan operaciones sobre valores booleanos, o resultados de expresiones relacionales, dando como resultado un valor booleano.

Los operadores lógicos los podemos ver en la tabla que se muestra a continuación. Existen ciertos casos en los que el segundo operando de una expresión lógica no se evalúa para ahorrar tiempo de ejecución, porque con el primero ya es suficiente para saber cuál va a ser el resultado de la expresión.

Por ejemplo, en la expresión `a && b` si `a` es falso, no se sigue comprobando la expresión, puesto que ya se sabe que la condición de que ambos sean verdadero no se va a cumplir. En estos casos es más conveniente colocar el operando más propenso a ser falso en el lado de la izquierda. Igual ocurre con el operador `||`, en cuyo caso es más favorable colocar el operando más propenso a ser verdadero en el lado izquierdo.

### Operadores lógicos en Java

Operador	Ejemplo en Java	Significado
!	<code>!op</code>	Devuelve true si el operando es false y viceversa.
&	<code>op1 &amp; op2</code>	Devuelve true si <code>op1</code> y <code>op2</code> son true
	<code>op1   op2</code>	Devuelve true si <code>op1</code> u <code>op2</code> son true
^	<code>op1 ^ op2</code>	Devuelve true si sólo uno de los operandos es true
<code>&amp;&amp;</code>	<code>op1 &amp;&amp; op2</code>	Igual que <code>&amp;</code> , pero si <code>op1</code> es false ya no se evalúa <code>op2</code>
<code>  </code>	<code>op1    op2</code>	Igual que <code> </code> , pero si <code>op1</code> es true ya no se evalúa <code>op2</code>

En el siguiente código puedes ver un ejemplo de utilización de operadores lógicos:

```
public class operadoreslogicos {
    public static void main(String[] args) {
        // TODO code application logic here
        System.out.println("OPERADORES LÓGICOS");

        System.out.println("Negacion:\n ! false es : " + (! false));
        System.out.println(" ! true es : " + (! true));

        System.out.println("Operador AND (&):\n false & false es : " + (false & false));
        System.out.println(" false & true es : " + (false & true));
        System.out.println(" true & false es : " + (true & false));
        System.out.println(" true & true es : " + (true & true));

        System.out.println("Operador OR (|):\n false | false es : " + (false | false));
        System.out.println(" false | true es : " + (false | true));
        System.out.println(" true | false es : " + (true | false));
        System.out.println(" true | true es : " + (true | true));

        System.out.println("Operador OR Exclusivo (^):\n false ^ false es : " + (false ^ false));
        System.out.println(" false ^ true es : " + (false ^ true));
        System.out.println(" true ^ false es : " + (true ^ false));
        System.out.println(" true ^ true es : " + (true ^ true));

        System.out.println("Operador &&:\n false && false es : " + (false && false));
        System.out.println(" false && true es : " + (false && true));
    }
}
```

```
System.out.println("Operador &&:\n false && false es : " + (false && false));
System.out.println(" false && true es : " + (false && true));
System.out.println(" true && false es : " + (true && false));
System.out.println(" true && true es : " + (true && true));

System.out.println("Operador ||:\n false || false es : " + (false || false));
System.out.println(" false || true es : " + (false || true));
System.out.println(" true || false es : " + (true || false));
System.out.println(" true || true es : " + (true || true));
} // fin main
} // fin operadoreslogicos
```

## OPERADORES LÓGICOS

Negacion:

```
! false es : true
! true es : false
```

Operador AND (&):

```
false & false es : false
false & true es : false
true & false es : false
true & true es : true
```

Operador OR (|):

```
false | false es : false
false | true es : true
true | false es : true
true | true es : true
```

Operador OR Exclusivo (^):

```
false ^ false es : false
false ^ true es : true
true ^ false es : true
true ^ true es : false
```

Operador &&:

```
false && false es : false
false && true es : false
true && false es : false
true && true es : true
```

Operador ||:

```
false || false es : false
false || true es : true
true || false es : true
true || true es : true
```

## 5.4.6.- Operadores de bits.

Los operadores a nivel de bits se caracterizan porque realizan operaciones sobre números enteros (o `char`) en su representación binaria, es decir, sobre cada dígito binario.

En la tabla tienes los operadores a nivel de bits que utiliza Java.

**Operadores a nivel de bits en Java**

Operador	Ejemplo en Java	Significado
<code>~</code>	<code>~op</code>	Realiza el complemento binario de op (invierte el valor de cada bit)
<code>&amp;</code>	<code>op1 &amp; op2</code>	Realiza la operación AND binaria sobre op1 y op2
<code> </code>	<code>op1   op2</code>	Realiza la operación OR binaria sobre op1 y op2
<code>^</code>	<code>op1 ^ op2</code>	Realiza la operación OR-exclusivo (XOR) binaria sobre op1 y op2
<code>&lt;&lt;</code>	<code>op1 &lt;&lt; op2</code>	Desplaza op2 veces hacia la izquierda los bits de op1
<code>&gt;&gt;</code>	<code>op1 &gt;&gt; op2</code>	Desplaza op2 veces hacia la derecha los bits de op1
<code>&gt;&gt;&gt;</code>	<code>op1 &gt;&gt;&gt; op2</code>	Desplaza op2 (en positivo) veces hacia la derecha los bits de op1



### Para saber más

Los operadores de bits raramente los vas a utilizar en tus aplicaciones de gestión. No obstante, si sientes curiosidad sobre su funcionamiento, puedes ver el siguiente enlace dedicado a este tipo de operadores:

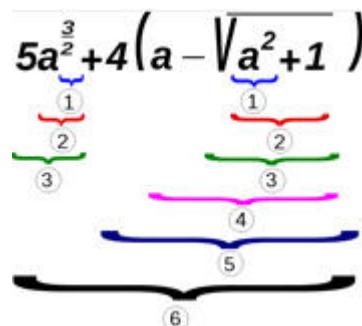
[Operadores de bits](#)

## 5.4.7.- Precedencia de operadores.

El orden de precedencia de los operadores determina la secuencia en que deben realizarse las operaciones cuando en una expresión intervienen operadores de distinto tipo.

Las reglas de precedencia de operadores que utiliza Java coinciden con las reglas de las expresiones del álgebra convencional. Por ejemplo:

- ✓ La **multiplicación, división y resto** de una operación se evalúan primero. Si dentro de la misma expresión tengo varias operaciones de este tipo, empezaré evaluándolas de izquierda a derecha.
- ✓ La **suma y la resta** se aplican después que las anteriores. De la misma forma, si dentro de la misma expresión tengo varias sumas y restas empezaré evaluándolas de izquierda a derecha.



A la hora de evaluar una expresión es necesario tener en cuenta la **asociatividad** de los operadores. La asociatividad indica qué operador se evalúa antes, en condiciones de igualdad de precedencia. Los operadores de asignación, el operador condicional (?:), los operadores incrementales (++, --) y el casting son asociativos por la derecha. El resto de operadores son asociativos por la izquierda, es decir, que se empiezan a calcular en el mismo orden en el que están escritos: de izquierda a derecha. Por ejemplo, en la expresión siguiente:

10 / 2 \* 5

Realmente la operación que se realiza es  $(10 / 2) * 5$ , porque ambos operadores, división y multiplicación, tienen igual precedencia y por tanto se evalúan primero el que antes nos encontramos por la izquierda, que es la división. El resultado de la expresión es 25. Si fueran asociativos por la derecha, puedes comprobar que el resultado sería diferente, primero multiplicaríamos  $2 * 5$  y luego dividiríamos entre 10, por lo que el resultado sería 1. En esta otra expresión:

x = y = z = 1

Realmente la operación que se realiza es  $x = (y = (z = 1))$ . Primero asignamos el valor de 1 a la variable z, luego a la variable y, para terminar asignando el resultado de esta última asignación a x. Si el operador asignación fuera asociativo por la izquierda esta operación no se podría realizar, ya que intentaríamos asignar a x el valor de y, pero y aún no habría sido inicializada.

En la tabla se detalla el orden de precedencia y la asociatividad de los operadores que hemos comentado en este apartado. Los operadores se muestran de mayor a menor precedencia.

### Orden de precedencia de operadores en Java

Operador	Tipo	Asociatividad
++ --	Unario, notación postfija	Derecha

Operador	Tipo	Asociatividad
<code>++ -- + - (cast) ! ~</code>	Unario, notación prefija	Derecha
<code>* / %</code>	Aritméticos	Izquierda
<code>+ -</code>	Aritméticos	Izquierda
<code>&lt;&lt;&gt;&gt;&gt;</code>	Bits	Izquierda
<code>&lt;&lt;= &gt;=</code>	Relacionales	Izquierda
<code>== !=</code>	Relacionales	Izquierda
<code>&amp;</code>	Lógico, Bits	Izquierda
<code>^</code>	Lógico, Bits	Izquierda
<code> </code>	Lógico, Bits	Izquierda
<code>&amp;&amp;</code>	Lógico	Izquierda
<code>  </code>	Lógico	Izquierda
<code>?:</code>	Operador condicional	Derecha
<code>= += -= *= /= %=</code>	Asignación	Derecha



## Reflexiona

¿Crees que es una buena práctica de programación utilizar paréntesis en expresiones aritméticas complejas, aún cuando no sean necesarios?

## 5.5.- Conversión de tipo.

Imagina que queremos dividir un número entre otro ¿tendrá decimales el resultado de esa división? Podemos pensar que siempre que el denominador no sea divisible entre el divisor, tendremos un resultado con decimales, pero no es así. Si el denominador y el divisor son variables de tipo entero, el resultado será entero y no tendrá decimales. Para que el resultado tenga decimales necesitaremos hacer una **conversión de tipo**.

Las conversiones de tipo se realizan para hacer que el resultado de una expresión sea del tipo que nosotros deseamos, en el ejemplo anterior para hacer que el resultado de la división sea de tipo real y, por ende, tenga decimales. Existen dos tipos de conversiones:

- ✓ **Conversiones automáticas.** Cuando a una variable de un tipo se le asigna un valor de otro tipo numérico con menos bits para su representación, se realiza una conversión automática. En ese caso, el valor se dice que es promocionado al tipo más grande (el de la variable), para poder hacer la asignación. También se realizan conversiones automáticas en las operaciones aritméticas, cuando estamos utilizando valores de distinto tipo, el valor más pequeño se promociona al valor más grande, ya que el tipo mayor siempre podrá representar cualquier valor del tipo menor (por ejemplo, de `int` a `long` o de `float` a `double`).
- ✓ **Conversiones explícitas.** Cuando hacemos una conversión de un tipo con más bits a un tipo con menos bits. En estos casos debemos indicar que queremos hacer la conversión de manera expresa, ya que se puede producir una pérdida de datos y hemos de ser conscientes de ello. Este tipo de conversiones se realiza con el **operador cast**. El operador `cast` es un operador unario que se forma colocando delante del valor a convertir el tipo de dato entre paréntesis. Tiene la misma precedencia que el resto de operadores unarios y se asocia de izquierda a derecha.

Debemos tener en cuenta que **un valor numérico nunca puede ser asignado a una variable de un tipo menor en rango, si no es con una conversión explícita**. Por ejemplo:

```
int a;
byte b;
a = 12;      // no se realiza conversión alguna
b = 12;      // se permite porque 12 está dentro
              // del rango permitido de valores para b
b = a;       // error, no permitido (incluso aunque
              // 12 podría almacenarse en un byte)
byte b = (byte) a; // Correcto, forzamos conversión explícita
```

En el ejemplo anterior vemos un caso típico de error de tipos, ya que estamos intentando asignarle a `b` el valor de `a`, siendo `b` de un tipo más pequeño. Lo correcto es promocionar `a` al tipo de datos `byte`, y entonces asignarle su valor a la variable `b`.

**Tabla de Conversión de Tipos de Datos Primitivos**

		Tipo destino								
		boolean	char	byte	short	int	long	float	double	
Tipo origen		boolean	-	N	N	N	N	N	N	

char	N	-	C	C	CI	CI	CI	CI
byte	N	C	-	CI	CI	CI	CI	CI
short	N	C	C	-	CI	CI	CI	CI
int	N	C	C	C	-	CI	CI*	CI
long	N	C	C	C	C	-	CI*	CI*
float	N	C	C	C	C	C	-	CI
double	N	C	C	C	C	C	C	-

## Explicación de los símbolos utilizados

**N:** Conversión no permitida (un **boolean** no se puede convertir a ningún otro tipo y viceversa).

**CI:** Conversión implícita o automática. Un asterisco indica que puede haber posible pérdida de datos.

**C:** Casting de tipos o conversión explícita.

El asterisco indica que puede haber una posible pérdida de datos, por ejemplo al convertir un número de tipo **int** que usa los 32 bits posibles de la representación, a un tipo **float**, que también usa 32 bits para la representación, pero 8 de los cuales son para el exponente.

En cualquier caso, las conversiones de números en coma flotante a números enteros siempre necesitarán un **Casting**, y deberemos tener mucho cuidado debido a la pérdida de precisión que ello supone.

## Reglas de Promoción de Tipos de Datos

Cuando en una expresión hay datos o variables de distinto tipo, el compilador realiza la promoción de unos tipos en otros, para obtener como resultado el tipo final de la expresión. Esta promoción de tipos se hace siguiendo unas reglas básicas en base a las cuales se realiza esta promoción de tipos, y resumidamente son las siguientes:

- ✓ Si uno de los operandos es de tipo **double**, el otro es convertido a **double**.
- ✓ En cualquier otro caso:
  - ◆ Si el uno de los operandos es **float**, el otro se convierte a **float**
  - ◆ Si uno de los operandos es **long**, el otro se convierte a **long**
  - ◆ Si no se cumple ninguna de las condiciones anteriores, entonces ambos operandos son convertidos al tipo **int**.

## Tabla sobre otras consideraciones

Conversiones de números en Coma flotante (float, double) a enteros (int)	Conversiones entre caracteres (char)
Cuando convertimos números en coma flotante a números enteros, la parte decimal se trunca (redondeo a cero). Si queremos hacer otro tipo de redondeo, podemos utilizar, entre otras, las siguientes funciones:	Como un tipo <b>char</b> lo que guarda en carácter, los caracteres pueden ser tanto con signo como sin signo.

## Conversiones de números en Coma flotante (float, double) a enteros (int)

- ✓ **Math.round(num):** Redondeo al siguiente número entero.
- ✓ **Math.ceil(num):** Mínimo entero que sea mayor o igual a num.
- ✓ **Math.floor(num):** Entero mayor, que sea inferior o igual a num.

```
double num=3.5;  
x=Math.round(num); // x = 4  
y=Math.ceil(num); // y = 4  
z=Math.floor(num); // z = 3
```

## Conversiones entre caracte

### Ejemplo:

```
int num;  
char c;  
  
num = (int) 'A'; //num = 65  
c = (char) 65; // c = 'A'  
c = (char) ((int) 'A' + 1); // c = 'B'
```



## Debes conocer

En el siguiente enlace viene información importante sobre cómo se producen las conversiones de tipos en Java, tanto automáticas como explícitas:

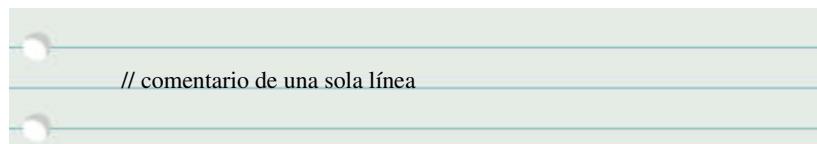
[Conversión de Tipos de Datos en Java](#)

## 5.6.- Comentarios.

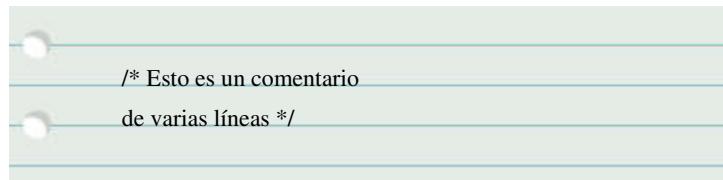
Los comentarios son muy importantes a la hora de describir qué hace un determinado programa. A lo largo de la unidad los hemos utilizado para documentar los ejemplos y mejorar la comprensión del código. Para lograr ese objetivo, es normal que cada programa comience con unas líneas de comentario que indiquen, al menos, una breve descripción del programa, el autor del mismo y la última fecha en que se ha modificado.

Todos los lenguajes de programación disponen de alguna forma de introducir comentarios en el código. En el caso de Java, nos podemos encontrar los siguientes tipos de comentarios:

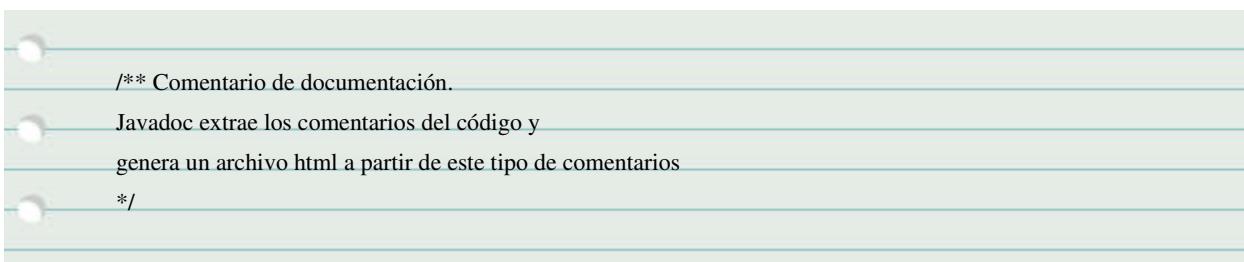
- ✓ **Comentarios de una sola línea.** Utilizaremos el delimitador // para introducir comentarios de sólo una línea.



- ✓ **Comentarios de múltiples líneas.** Para introducir este tipo de comentarios, utilizaremos una barra inclinada y un asterisco (\*), al principio del párrafo y un asterisco seguido de una barra inclinada (\*)/ al final del mismo.



- ✓ **Comentarios Javadoc.** Utilizaremos los delimitadores /\*\* y \*/. Al igual que con los comentarios tradicionales, el texto entre estos delimitadores será ignorado por el compilador. Este tipo de comentarios se emplean para generar documentación automática del programa. A través del programa javadoc, incluido en JavaSE , se recogen todos estos comentarios y se llevan a un documento en formato .html.



### Reflexiona

Una buena práctica de programación es añadir en la última llave que delimita cada bloque de código, un comentario indicando a qué clase o método pertenece esa llave.



### Para saber más

Si quieres ir familiarizándote con la información que hay en la web de Oracle, en el siguiente enlace puedes encontrar más información sobre la herramienta Javadoc incluida en el Kit de Desarrollo de Java SE (en inglés):

[Página oficial de Oracle sobre la herramienta Javadoc](#)

## 6.- Estructuras de control

Este epígrafe lo utilizaremos para reafirmar cuestiones que son obvias y que en el transcurso de anteriores unidades se han dado por sabidas. Aunque, a veces, es conveniente recordar. Lo haremos como un conjunto de FAQ:

- ✓ **¿Cómo se escribe un programa sencillo?** Si queremos que un programa sencillo realice instrucciones o sentencias para obtener un determinado resultado, es necesario colocar éstas una detrás de la otra, exactamente en el orden en que deben ejecutarse.
- ✓ **¿Podrían colocarse todas las sentencias una detrás de otra, separadas por puntos y comas en una misma línea?** Claro que sí, pero no es muy recomendable. Cada sentencia debe estar escrita en una línea, de esta manera tu código será mucho más legible y la localización de errores en tus programas será más sencilla y rápida. De hecho, cuando se utilizan herramientas de programación, los errores suelen asociarse a un número o números de línea.
- ✓ **¿Puede una misma sentencia ocupar varias líneas en el programa?** Sí. Existen sentencias que, por su tamaño, pueden generar varias líneas. Pero siempre finalizarán con un punto y coma.
- ✓ **¿En Java todas las sentencias se terminan con punto y coma?** Efectivamente. Si detrás de una sentencia ha de venir otra, pondremos un punto y coma. Escribiendo la siguiente sentencia en una nueva línea. Pero en algunas ocasiones, sobre todo cuando utilizamos estructuras de control de flujo, detrás de la cabecera de una estructura de este tipo no debe colocarse punto y coma. No te preocupes, lo entenderás cuando analicemos cada una de ellas.
- ✓ **¿Qué es la sentencia nula en Java?** La sentencia nula es una línea que no contiene ninguna instrucción y en la que sólo existe un punto y coma. Como su nombre indica, esta sentencia no hace nada.
- ✓ **¿Qué es un bloque de sentencias?** Es un conjunto de sentencias que se encierra entre llaves y que se ejecutaría como si fuera una única orden. Sirve para agrupar sentencias y para clarificar el código. Los bloques de sentencias son utilizados en Java en la práctica totalidad de estructuras de control de flujo, clases, métodos, etc. La siguiente tabla muestra dos formas de construir un bloque de sentencias.

### Bloques de sentencias.

Bloque de sentencias 1	Bloque de sentencias 2
{sentencia1; sentencia2;...; sentencia N;}	{ sentencia1; sentencia2; ...; sentenciaN; }

- ✓ **¿En un bloque de sentencias, éstas deben estar colocadas con un orden exacto?** En ciertos casos sí, aunque si al final de su ejecución se obtiene el mismo resultado, podrían ocupar diferentes posiciones en nuestro programa.

Construyas de una forma o de otra tus programas, debes tener en cuenta siempre en Java las siguientes premisas:

- ✓ Declara cada variable antes de utilizarla.
- ✓ Inicializa con un valor cada variable la primera vez que la utilices.
- ✓ No es recomendable usar variables no inicializadas en nuestros programas, pueden provocar errores o resultados imprevistos.



## Autoevaluación

Indica qué afirmación es correcta:

- Para crear un bloque de sentencias, es necesario delimitar éstas entre llaves. Este bloque funcionará como si hubiéramos colocado una única orden
- La sentencia nula en Java, se puede representar con un punto y coma sólo en una única línea.
- Para finalizar en Java cualquier sentencia, es necesario hacerlo con un punto y coma.
- Todas las afirmaciones son correctas

## 6.1.- Estructuras de selección.

¿Cómo conseguimos que nuestros programas puedan tomar decisiones? Para comenzar, lo haremos a través de las estructuras de selección. Estas estructuras constan de una sentencia especial de decisión y de un conjunto de secuencias de instrucciones.

El funcionamiento es sencillo, la sentencia de decisión será evaluada y ésta devolverá un valor (verdadero o falso), en función del valor devuelto se ejecutará una secuencia de instrucciones u otra. Por ejemplo, si el valor de una variable es mayor o igual que 5 se imprime por pantalla la palabra APROBADO y, si es menor, se imprime SUSPENSO. Para este ejemplo, la comprobación del valor de la variable será la sentencia especial de decisión. La impresión de la palabra APROBADO será una secuencia de instrucciones y la impresión de la palabra SUSPENSO será otra. Cada secuencia estará asociada a cada uno de los resultados que puede arrojar la evaluación de la sentencia especial de decisión.



### Recomendación

En el lenguaje de programación C, verdadero o falso se representan mediante un literal entero. 0 representará Falso y 1 o cualquier otro valor, representará Verdadero. Como sabes, en Java las variables de tipo booleano sólo podrán tomar los valores true (verdadero) o false (falso).

La evaluación de las sentencias de decisión o expresiones que controlan las estructuras de selección, devolverán siempre un valor verdadero o falso.

Las estructuras de selección se dividen en:

1. Estructuras de selección simples o estructura if.
2. Estructuras de selección compuestas o estructura if-else.
3. Estructuras de selección basadas en el **operador condicional**.
4. Estructuras de selección múltiples o estructura switch.

A continuación, detallaremos las características y funcionamiento de cada una de ellas. Es importante que a través de los ejemplos que vamos a ver, puedas determinar en qué circunstancias utilizar cada una de estas estructuras. Aunque un mismo problema puede ser resuelto con diferentes estructuras e incluso, con diferentes combinaciones de éstas.



## 6.1.1.- Estructura if / if-else.

La estructura **if** es una estructura de selección o estructura condicional, en la que se evalúa una expresión lógica o sentencia de decisión y en función del resultado, se ejecuta una sentencia o un bloque de éstas.

La estructura **if** puede presentarse de las siguientes formas:

### Estructura if simple.

Estructura if simple.	
<p><b>Sintaxis:</b></p> <pre>if (expresión-lógica)     sentencia1;</pre>	<p><b>Sintaxis:</b></p> <pre>if (expresión-lógica) {     sentencia1;     sentencia2;     ...     sentenciaN; }</pre>

### Funcionamiento:

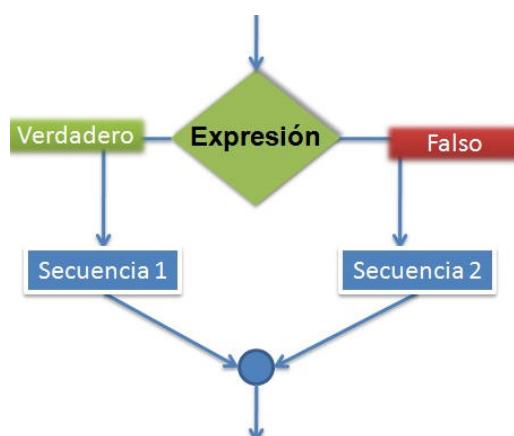
Si la evaluación de la expresión-lógica ofrece un resultado verdadero, se ejecuta la **sentencia1** o bien el bloque de sentencias asociado. Si el resultado de dicha evaluación es falso, no se ejecutará ninguna instrucción asociada a la estructura condicional.

Haciendo una interpretación cercana al pseudocódigo tendríamos que si se cumple la condición (expresión lógica), se ejecutará un conjunto de instrucciones y si no se cumple, se ejecutará otro conjunto de instrucciones.

Hay que tener en cuenta que la cláusula **else** de la sentencia **if** no es obligatoria. En algunos casos no necesitaremos utilizarla, pero sí se recomienda cuando es necesario llevar a cabo alguna acción en el caso de que la expresión lógica no se cumpla.

En aquellos casos en los que no existe cláusula **else**, si la expresión lógica es falsa, simplemente se continuarán ejecutando las siguientes sentencias que aparezcan bajo la estructura condicional **if**.

Los condicionales **if** e **if-else** pueden anidarse, de tal forma que dentro de un bloque de sentencias puede incluirse otro **if** o **if-else**. El nivel de anidamiento queda a criterio del programador, pero si éste es demasiado profundo podría provocar problemas de eficiencia y legibilidad en el código. En otras ocasiones, un nivel de anidamiento excesivo puede denotar la necesidad de utilización de otras estructuras de selección más adecuadas.



Cuando se utiliza anidamiento de este tipo de estructuras, es necesario poner especial atención en saber a qué `if` está asociada una cláusula `else`. Normalmente, un `else` estará asociado con el `if` inmediatamente superior o más cercano que exista dentro del mismo bloque y que no se encuentre ya asociado a otro `else`.

Para completar la información que debes saber sobre las estructuras `if` e `if-else`, revisa el siguiente ejemplo. En él podrás analizar el código de un programa que realiza el cálculo de la nota de un examen de tipo test. Además de calcular el valor de la nota, se ofrece como salida la calificación no numérica de dicho examen. Para obtenerla, se combinarán las diferentes estructuras condicionales aprendidas hasta ahora.

Presta especial atención a los comentarios incorporados en el código fuente, así como a la forma de combinar las estructuras condicionales y a las expresiones lógicas utilizadas en ellas.

```
package sentencias_condicionales;
/**
 * Ejemplos de utilización de diferentes estructuras condicionales simples, completas y anidamiento de
 */
public class Sentencias_condicionales {

    /*Vamos a realizar el cálculo de la nota de un examen de tipo test. Para ello, tendremos en cuenta
     * total de pregunta, los aciertos y los errores. Dos errores anulan una respuesta correcta.
     * Finalmente, se muestra por pantalla la nota obtenida, así como su calificación no numérica.
     * La obtención de la calificación no numérica se ha realizado combinando varias estructuras condicionales
     * lógicas compuestas, así como anidamiento.      */
    public static void main(String[] args) {
        // Declaración e inicialización de variables
        int num_aciertos = 12;
        int num_errores = 3;
        int num_preguntas = 20;
        float nota = 0;
        String calificacion="";

        //Procesamiento de datos
        nota = ((num_aciertos - (num_errores/2))*10)/num_preguntas;

        if (nota < 5)
        {
            calificacion="INSUFICIENTE";
        }
        else
        {
            if (nota >= 5 && nota <6)
                calificacion="SUFICIENTE";
            if (nota >= 6 && nota <7)
                calificacion="BIEN";
            if (nota >= 7 && nota <9)
                calificacion="NOTABLE";
            if (nota >= 9 && nota <=10)
                calificacion="SOBRESALIENTE";
        }

        //Salida de información
        System.out.println ("La nota obtenida es: " + nota);
        System.out.println ("y la calificación obtenida es: " + calificacion);
    }
}
```



## Autoevaluación

¿Cuándo se mostrará por pantalla el mensaje incluido en el siguiente fragmento de código?

```
If(numero % 2 == 0);  
System.out.print("El número es par /n");
```

- Nunca.
- Siempre.
- Cuando el resto de la división entre 2 del contenido de la variable **numero**, sea cero.

## 6.1.2.- Estructura switch.

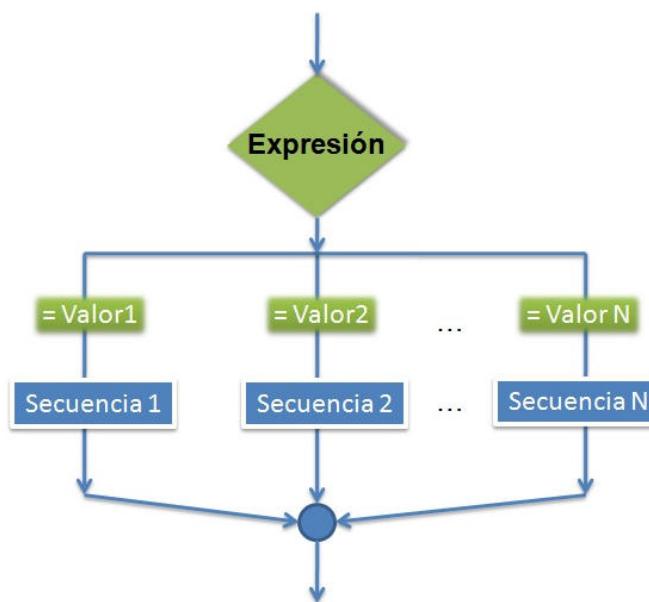
¿Qué podemos hacer cuando nuestro programa debe elegir entre más de dos alternativas?, una posible solución podría ser emplear estructuras **if** anidadas, aunque no siempre esta solución es la más eficiente. Cuando estamos ante estas situaciones podemos utilizar la estructura de selección múltiple **switch**. En la siguiente tabla se muestra tanto la sintaxis, como el funcionamiento de esta estructura.

### Estructura switch.

Estructura switch	
<b>Sintaxis:</b> <pre>switch (expresion) {     case valor1:         sentencia1_1;         sentencia1_2;         ....         break;         ....     case valorN:         sentenciaN_1;         sentenciaN_2;         ....         break;     default:         sentencias-default; }</pre>	<b>Condiciones:</b> <ul style="list-style-type: none"><li>✓ Donde expresión debe ser del tipo <b>char</b>, <b>byte</b>, <b>short</b> o <b>int</b>, y las constantes de cada <b>case</b> deben ser de este tipo o de un tipo compatible.</li><li>✓ La expresión debe ir entre paréntesis.</li><li>✓ Cada <b>case</b> llevará asociado un valor y se finalizará con dos puntos.</li><li>✓ El bloque de sentencias asociado a la cláusula <b>default</b> puede finalizar con una sentencia de ruptura <b>break</b> o no.</li></ul>
<b>Funcionamiento:</b> <ul style="list-style-type: none"><li>✓ Las diferentes alternativas de esta estructura estarán precedidas de la cláusula <b>case</b> que se ejecutará cuando el valor asociado al <b>case</b> coincida con el valor obtenido al evaluar la expresión del <b>switch</b>.</li><li>✓ En las cláusulas <b>case</b>, no pueden indicarse expresiones condicionales, rangos de valores o listas de valores. (otros lenguajes de programación sí lo permiten). Habrá que asociar una cláusula <b>case</b> a cada uno de los valores que deban ser tenidos en cuenta.</li><li>✓ La cláusula <b>default</b> será utilizada para indicar un caso por defecto, las sentencias asociadas a la cláusula <b>default</b> se ejecutarán si ninguno de los valores indicados en las cláusulas <b>case</b> coincide con el resultado de la evaluación de la expresión de la estructura <b>switch</b>.</li><li>✓ La cláusula <b>default</b> puede no existir, y por tanto, si ningún <b>case</b> ha sido activado finalizaría el <b>switch</b>.</li><li>✓ Cada cláusula <b>case</b> puede llevar asociadas una o varias sentencias, sin necesidad de delimitar dichos bloques por medio de llaves.</li><li>✓ En el momento en el que el resultado de la evaluación de la expresión coincide con alguno de los valores asociados a las cláusulas <b>case</b>, se ejecutarán todas las instrucciones asociadas hasta la aparición de una sentencia <b>break</b> de ruptura. (la sentencia <b>break</b> se analizará en epígrafes posteriores)</li></ul>	

En resumen, se ha de comparar el valor de una expresión con un conjunto de constantes, si el valor de la expresión coincide con algún valor de dichas constantes, se ejecutarán los bloques de instrucciones

asociados a cada una de ellas. Si no existiese coincidencia, se ejecutarían una serie de instrucciones por defecto.



En el siguiente fragmento de código en el que se resuelve el cálculo de la nota de un examen de tipo test, utilizando la estructura switch.

```
public class condicional_switch {  
  
    public static void main(String[] args) {  
        // Declaración e inicialización de variables  
        int num_aciertos = 17;  
        int num_errores = 3;  
        int num_preguntas = 20;  
        int nota = 0;  
        String calificacion="";  
  
        //Procesamiento de datos  
        nota = ((num_aciertos - (num_errores/2))*10)/num_preguntas;  
  
        switch (nota) {  
            case 5: calificacion="SUFICIENTE";  
                break;  
            case 6: calificacion="BIEN";  
                break;  
            case 7: calificacion="NOTABLE";  
                break;  
            case 8: calificacion="NOTABLE";  
                break;  
            case 9: calificacion="SOBRESALIENTE";  
                break;  
            case 10: calificacion="SOBRESALIENTE";  
                break;  
            default: calificacion="INSUFICIENTE";  
        }  
  
        //Salida de información  
        System.out.println ("La nota obtenida es: " + nota);  
        System.out.println ("y la calificación obtenida es: " + calificacion);  
    }  
}
```

## 6.2.- Estructuras de repetición.

Nuestros programas ya son capaces de controlar su ejecución teniendo en cuenta determinadas condiciones, pero aún hemos de aprender un conjunto de estructuras que nos permita repetir una secuencia de instrucciones determinada. La función de estas estructuras es repetir la ejecución de una serie de instrucciones teniendo en cuenta una condición.

A este tipo de estructuras se las denomina **estructuras de repetición**, estructuras repetitivas, **bucles** o estructuras iterativas. En Java existen cuatro clases de bucles:



- ✓ Bucle **for** (repite para)
- ✓ Bucle **for/in** (repite para cada)
- ✓ Bucle **While** (repite mientras)
- ✓ Bucle **Do While** (repite hasta)

Los bucles **for** y **for/in** se consideran bucles **controlados por contador**. Por el contrario, los bucles **while** y **do...while** se consideran bucles **controlados por sucesos**.

La utilización de unos bucles u otros para solucionar un problema dependerá en gran medida de las siguientes preguntas:

- ✓ ¿Sabemos a priori cuántas veces necesitamos repetir un conjunto de instrucciones?
- ✓ ¿Sabemos si hemos de repetir un conjunto de instrucciones si una condición satisface un conjunto de valores?
- ✓ ¿Sabemos hasta cuándo debemos estar repitiendo un conjunto de instrucciones?
- ✓ ¿Sabemos si hemos de estar repitiendo un conjunto de instrucciones mientras se cumpla una condición?



### Recomendación

Estudia cada tipo de estructura repetitiva, conoce su funcionamiento y podrás llegar a la conclusión de que algunos de estos bucles son equivalentes entre sí. Un mismo problema, podrá ser resuelto empleando diferentes tipos de bucles y obtener los mismos resultados.

Estas y otras preguntas tendrán su respuesta en cuanto analicemos cada una de estructuras repetitivas en detalle.

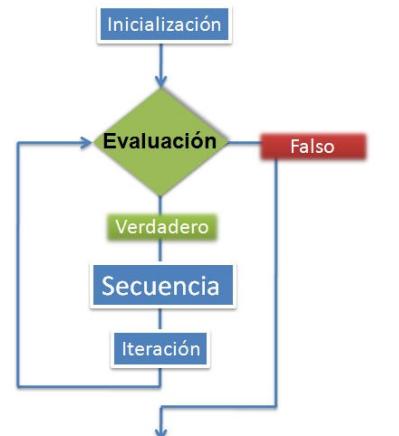
## 6.2.1.- Estructura for.

Hemos indicado anteriormente que el bucle **for** es un bucle controlado por contador. Este tipo de bucle tiene las siguientes características:

- ✓ Se ejecuta un número determinado de veces.
- ✓ Utiliza una variable contadora que controla las iteraciones del bucle.

En general, existen tres operaciones que se llevan a cabo en este tipo de bucles:

- ✓ Se inicializa la variable contadora.
- ✓ Se evalúa el valor de la variable contadora, por medio de una comparación de su valor con el número de iteraciones especificado.
- ✓ Se modifica o actualiza el valor del contador a través de incrementos o decrementos de éste, en cada una de las iteraciones.



### Recomendación

La inicialización de la variable contadora debe realizarse correctamente para garantizar que el bucle se lleve a cabo, al menos, la primera repetición de su código interno.

La condición de terminación del bucle debe variar en el interior del mismo, de no ser así, podemos caer en la creación de un bucle infinito. Cuestión que se debe evitar por todos los medios.

Es necesario estudiar el número de veces que se repite el bucle, pues debe ajustarse al número de veces estipulado.

En la siguiente tabla, podemos ver la especificación de la estructura **for**:

### Estructura repetitiva **for**.

#### Estructura repetitiva **for**

##### Sintaxis:

```
for (inicialización; condición; iteración)
    sentencia;
```

(estructura **for** con una única sentencia)

Donde **inicialización** es una expresión en la que se inicializa una variable de control, que será la encargada de controlar el final del bucle.

Donde **condición** es una expresión que evaluará la variable

## Estructura repetitiva for

### Sintaxis:

```
for ( inicialización; condición; iteración )
{
    sentencia1;
    sentencia2;
    ...
    sentenciaN;
}
```

(estructura for con un bloque de sentencias)

de control. Mientras la condición sea verdadera, el cuerpo del bucle estará repitiéndose.

Cuando la condición no se cumpla, terminará la ejecución del bucle. Donde iteración indica la manera en la que la variable de control va cambiando en cada iteración del bucle. Podrá ser mediante incremento o decremento, y no solo de uno en uno.

A continuación se muestra un ejemplo de utilización del bucle for para la impresión por pantalla de la tabla de multiplicar del siete.

```
public class repetitiva_for {
    /* En este ejemplo se utiliza la estructura repetitiva for
     * para representar en pantalla la tabla de multiplicar del siete
     */
    public static void main(String[] args) {
        // Declaración e inicialización de variables
        int numero = 7;
        int contador;
        int resultado=0;

        //Salida de información
        System.out.println ("Tabla de multiplicar del " + numero);
        System.out.println (".....");

        //Utilizamos ahora el bucle for
        for (contador=1; contador<=10;contador++)
        /* La cabecera del bucle incorpora la inicialización de la variable de control, la cond
         * incremento de dicha variable de uno en uno en cada iteración del bucle.
         * En este caso contador++ incrementará en una unidad el valor de dicha variable.
         */
        {
            resultado = contador * numero;
            System.out.println(numero + " x " + contador + " = " + resultado);
            /* A través del operador + aplicado a cadenas de caracteres, concatenamos los valor
             * caracteres que necesitamos para representar correctamente la salida de cada mult
             */
        }
    }
}
```



## Autoevaluación

Cuando construimos la cabecera de un bucle for, podemos prescindir de alguno de los tres elementos que la forman e incluso, podemos utilizar más de una variable contadora separando éstas por comas. Pero, ¿Qué conseguiremos si construimos un bucle de la siguiente forma?

```
for (;;) { //instrucciones }
```

- Un bucle infinito.
- Nada, dará un error.
- Un bucle que se ejecutaría una única vez.

## 6.2.2.- Estructura while.

El bucle `while` es la primera de las estructuras de repetición controladas por sucesos que vamos a estudiar. La utilización de este bucle responde al planteamiento de la siguiente pregunta: ¿Qué podemos hacer si lo único que sabemos es que se han de repetir un conjunto de instrucciones mientras se cumpla una determinada condición?

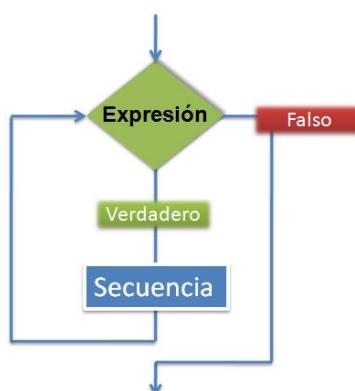
La característica fundamental de este tipo de estructura repetitiva estriba en ser útil en aquellos casos en los que las instrucciones que forman el cuerpo del bucle podría ser necesario ejecutarlas o no. Es decir, en el bucle `while` siempre se evaluará la condición que lo controla, y si dicha condición es cierta, el cuerpo del bucle se ejecutará una vez, y se seguirá ejecutando mientras la condición sea cierta. Pero si en la evaluación inicial de la condición ésta no es verdadera, el cuerpo del bucle no se ejecutará.

Es imprescindible que en el interior del bucle `while` se realice alguna acción que modifique la condición que controla la ejecución del mismo, en caso contrario estaríamos ante un bucle infinito.

### Estructura repetitiva `while`.

Estructura repetitiva <code>while</code>	
<b>Sintaxis:</b> <pre>while (condición)       sentencia;</pre>	<b>Sintaxis:</b> <pre>while (condición) {       sentencia1;       ...       sentenciaN; }</pre>
<b>Funcionamiento:</b> Mientras la condición sea cierta, el bucle se repetirá, ejecutando la/s instrucción/es de su interior. En el momento en el que la condición no se cumpla, el control del flujo del programa pasará a la siguiente instrucción que exista justo detrás del bucle <code>while</code> . La condición se evaluará siempre al principio, y podrá darse el caso de que las instrucciones contenidas en él no lleguen a ejecutarse nunca si no se satisface la condición de partida.	

En la siguiente imagen puedes ver un diagrama de flujo que representa el funcionamiento de este tipo de estructura repetitiva.



En el siguiente ejemplo se muestra la utilización del bucle `while` para la impresión por pantalla de la tabla de multiplicar del siete. Como podrás comprobar, el resultado de este bucle es totalmente equivalente al obtenido utilizando el bucle `for`.

```
public class repetitiva_while {
    public static void main(String[] args) {
        // Declaración e inicialización de variables
        int numero = 7;
        int contador;
        int resultado=0;

        //Salida de información
        System.out.println ("Tabla de multiplicar del " + numero);
        System.out.println (".....");

        //Utilizamos ahora el bucle while
        contador = 1; //inicializamos la variable contadora
        while (contador <= 10) //Establecemos la condición del bucle
        {
            resultado = contador * numero;
            System.out.println(numero + " x " + contador + " = " + resultado);
            //Modificamos el valor de la variable contadora, para hacer que el
            //bucle pueda seguir iterando hasta llegar a finalizar
            contador++;
        }
    }
}
```



## Autoevaluación

Utilizando el siguiente fragmento de código estamos construyendo un bucle infinito.  
¿Verdadero o Falso?

while (true) System.out.println("Imprimiendo desde dentro del bucle \n");

- Verdadero  Falso

## 6.2.3.- Estructura do-while.

La segunda de las estructuras repetitivas controladas por sucesos es **do-while**. En este caso, la pregunta que nos planteamos es la siguiente: ¿Qué podemos hacer si lo único que sabemos es que se han de ejecutar, al menos una vez, un conjunto de instrucciones y seguir repitiéndose hasta que se cumpla una determinada condición?

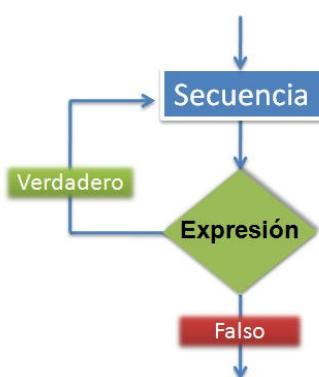
La característica fundamental de este tipo de estructura repetitiva estriba en ser útil en aquellos casos en los que las instrucciones que forman el cuerpo del bucle necesitan ser ejecutadas, al menos, una vez y repetir su ejecución hasta que la condición sea verdadera. Por tanto, en esta estructura repetitiva siempre se ejecuta el cuerpo del bucle una primera vez.

Es imprescindible que en el interior del bucle se realice alguna acción que modifique la condición que controla la ejecución del mismo, en caso contrario estaríamos ante un bucle infinito.

### Estructura repetitiva do-while.

Estructura repetitiva do-while	
<b>Sintaxis:</b> <pre>do   sentencia;   while (condición);</pre>	<b>Sintaxis:</b> <pre>do {   sentencia1;   ...   sentenciaN; } while (condición);</pre>
<b>Funcionamiento:</b> El cuerpo del bucle se ejecuta la primera vez, a continuación se evaluará la condición y, si ésta es falsa, el cuerpo del bucle volverá a repetirse. El bucle finalizará cuando la evaluación de la condición sea verdadera. En ese momento el control del flujo del programa pasará a la siguiente instrucción que exista justo detrás del bucle do-while. La condición se evaluará siempre después de una primera ejecución del cuerpo del bucle, por lo que no se dará el caso de que las instrucciones contenidas en él no lleguen a ejecutarse nunca.	

En la siguiente imagen puedes ver un diagrama de flujo que representa el funcionamiento de este tipo de estructura repetitiva.



A continuación puedes analizar un ejemplo de utilización del bucle **do-while** para la impresión por pantalla de la tabla de multiplicar del siete. Como podrás comprobar, el resultado de este bucle es totalmente equivalente al obtenido utilizando el bucle **for** y el bucle **while**.

```
public class repetitiva_dowhile {  
    public static void main(String[] args) {  
        // Declaración e inicialización de variables  
        int numero = 7;  
        int contador;  
        int resultado=0;  
  
        //Salida de información  
        System.out.println ("Tabla de multiplicar del " + numero);  
        System.out.println (".....");  
  
        //Utilizamos ahora el bucle do-while  
        contador = 1; //inicializamos la variable contadora  
        do  
        {  
            resultado = contador * numero;  
            System.out.println(numero + " x " + contador + " = " + resultado);  
            //Modificamos el valor de la variable contadora, para hacer que el  
            //bucle pueda seguir iterando hasta llegar a finalizar  
            contador++;  
        }while (contador <= 10); //Establecemos la condición del bucle  
    }  
}
```

## 6.3.- Estructuras de salto.

---

¿Saltar o no saltar? he ahí la cuestión. En la gran mayoría de libros de programación y publicaciones de Internet, siempre se nos recomienda que prescindamos de sentencias de salto incondicional, es más, se desaconseja su uso por provocar una mala estructuración del código y un incremento en la dificultad para el mantenimiento de los mismos. Pero Java incorpora ciertas sentencias o estructuras de salto que es necesario conocer y que pueden sernos útiles en algunas partes de nuestros programas.

Estas estructuras de salto corresponden a las sentencias `break`, `continue`, las **etiquetas de salto** y la sentencia `return`. Pasamos ahora a analizar su sintaxis y funcionamiento.

## 6.3.1.- Sentencias break y continue.

Se trata de dos instrucciones que permiten modificar el comportamiento de otras estructuras o sentencias de control, simplemente por el hecho de estar incluidas en algún punto de su secuencia de instrucciones.

La **sentencia break** incidirá sobre las estructuras de control **switch**, **while**, **for** y **do-while** del siguiente modo:

- ✓ Si aparece una sentencia **break** dentro de la secuencia de instrucciones de cualquiera de las estructuras mencionadas anteriormente, dicha estructura terminará inmediatamente.
- ✓ Si aparece una sentencia **break** dentro de un bucle anidado sólo finalizará la sentencia de iteración más interna, el resto se ejecuta de forma normal.

Es decir, que **break** sirve para romper el flujo de control de un bucle, aunque no se haya cumplido la condición del bucle. Si colocamos un **break** dentro del código de un bucle, cuando se alcance el **break**, automáticamente se saldrá del bucle pasando a ejecutarse la siguiente instrucción inmediatamente después de él.

En la siguiente imagen, puedes apreciar cómo se utilizaría la sentencia **break** dentro de un bucle **for**.

```
6  public class sentencia_break {  
7  public static void main(String[] args) {  
8      // Declaración de variables  
9      int contador;  
10  
11  
12      //Procesamiento y salida de información  
13  
14      for (contador=1;contador<=10;contador++)  
15      {  
16          if (contador==7)  
17              break;  
18          System.out.println ("Valor: " + contador);  
19      }  
20      System.out.println ("Fin del programa");  
21      /* El bucle sólo se ejecutará en 6 ocasiones, ya que cuando  
22      * la variable contador sea igual a 7 encontraremos un break que  
23      * romperá el flujo del bucle, transfiriéndonos a la sentencia que  
24      * imprime el mensaje de Fin del programa.  
25      */  
26  }
```

La **sentencia continue** incidirá sobre las sentencias o estructuras de control **while**, **for** y **do-while** del siguiente modo:

- ✓ Si aparece una sentencia **continue** dentro de la secuencia de instrucciones de cualquiera de las sentencias anteriormente indicadas, dicha sentencia dará por terminada la iteración actual y se ejecuta una nueva iteración, evaluando de nuevo la expresión condicional del bucle.
- ✓ Si aparece en el interior de un bucle anidado solo afectará a la sentencia de iteración más interna, el resto se ejecutaría de forma normal.

Es decir, la sentencia **continue** forzará a que se ejecute la siguiente iteración del bucle, sin tener en cuenta las instrucciones que pudiera haber después del **continue**, y hasta el final del código del bucle.

En la siguiente imagen, puedes apreciar cómo se utiliza la sentencia **continue** en un bucle **for** para imprimir por pantalla sólo los números pares.

```
4  * Uso de la sentencia continue  
5  */  
6  public class sentencia_continue {  
7  public static void main(String[] args) {  
8      // Declaración de variables
```

```
9 // Declaración de variables
10 int contador;
11
12 System.out.println ("Imprimiendo los números pares que hay del 1 al 10..
13 //Procesamiento y salida de información
14
15 for (contador=1;contador<=10;contador++)
16 {
17     if (contador % 2 != 0) continue;
18     System.out.print(contador + " ");
19 }
20 /* Las iteraciones del bucle que generarán la impresión de cada uno
21 * de los números pares, serán aquellas en las que el resultado de
22 * calcular el resto de la división entre 2 de cada valor de la variable
23 * contador, sea igual a 0.
24 */
25 }
26
27 }
```



## Autoevaluación

La instrucción `break` puede utilizarse en las estructuras de control `switch`, `while`, `for` y `do-while`, no siendo imprescindible utilizarla en la cláusula `default` de la estructura `switch`. ¿Verdadero o Falso?

- Verdadero  Falso

## 6.3.2.- Etiquetas.

Ya lo indicábamos al comienzo del epígrafe dedicado a las estructuras de salto, los saltos incondicionales y en especial, saltos a una etiqueta son totalmente desaconsejables. No obstante, Java permite asociar etiquetas cuando se va a realizar un salto. De este modo puede conseguirse algo más de legibilidad en el código.

Las estructuras de salto **break** y **continue**, pueden tener asociadas etiquetas. Es a lo que se llama un **break etiquetado** o un **continue etiquetado**. Pero sólo se recomienda su uso cuando se hace necesario salir de bucles anidados hacia diferentes niveles.

¿Y cómo se crea un salto a una etiqueta? En primer lugar, crearemos la etiqueta mediante un **identificador seguido de dos puntos (:)** . A continuación, se escriben las sentencias Java asociadas a dicha etiqueta encerradas entre llaves. Por así decirlo, la creación de una etiqueta es como fijar un punto de salto en el programa para poder saltar a él desde otro lugar de dicho programa.

¿Cómo se lleva a cabo el salto? Es sencillo, en el lugar donde vayamos a colocar la sentencia **break** o **continue**, añadiremos detrás el identificador de la etiqueta. Con ello, conseguiremos que el salto se realice a un lugar determinado. La sintaxis será **break <etiqueta>**.

Quizá a aquellos que habéis programado en HTML os suene esta herramienta, ya que tiene cierta similitud con las anclas que pueden crearse en el interior de una página web, a las que nos llevará el hiperenlace o link que hayamos asociado.

También para aquellos y aquellas que habéis creado alguna vez archivos por lotes o archivos batch bajo MSDOS es probable que también os resulte familiar el uso de etiquetas, pues la sentencia **GOTO** que se utilizaba en este tipo de archivos, hacía saltar el flujo del programa al lugar donde se ubicaba la etiqueta que se indicara en dicha sentencia.

A continuación, te ofrecemos un ejemplo de declaración y uso de etiquetas en un bucle. Como podrás apreciar, las sentencias asociadas a cada etiqueta están encerradas entre llaves para delimitar así su ámbito de acción.

```
3  * Uso de etiquetas en bucle
4  */
5  public class etiquetas {
6  public static void main(String[] args) {
7
8      for (int i=1; i<3; i++) //Creamos cabecera del bucle
9      {
10         bloque_uno: { //Creamos primera etiqueta
11             bloque_dos:{ //Creamos segunda etiqueta
12                 System.out.println("Iteración: "+i);
13                 if (i==1) break bloque_uno; //Llevamos a cabo el primer salto
14                 if (i==2) break bloque_dos; //Llevamos a cabo el segundo salto
15             }
16             System.out.println("después del bloque dos");
17         }
18         System.out.println("después del bloque uno");
19     }
20     System.out.println("Fin del bucle");
21 }
22 }
```

### 6.3.3.- Sentencia Return.

Ya sabemos como modificar la ejecución de bucles y estructuras condicionales múltiples, pero ¿Podríamos modificar la ejecución de un método? ¿Es posible hacer que éstos detengan su ejecución antes de que finalice el código asociado a ellos? Sí es posible, a través de la sentencia **return** podremos conseguirlo.



La sentencia **return** puede utilizarse de dos formas:

- ✓ Para terminar la ejecución del método donde esté escrita, con lo que transferirá el control al punto desde el que se hizo la llamada al método, continuando el programa por la sentencia inmediatamente posterior.
- ✓ Para devolver o retornar un valor, siempre que junto a **return** se incluya una expresión de un tipo determinado. Por tanto, en el lugar donde se invocó al método se obtendrá el valor resultante de la evaluación de la expresión que acompañaba al método.

En general, una sentencia **return** suele aparecer al final de un método, de este modo el método tendrá una entrada y una salida. También es posible utilizar una sentencia **return** en cualquier punto de un método, con lo que éste finalizará en el lugar donde se encuentre dicho **return**. No será recomendable incluir más de un **return** en un método y por regla general, deberá ir al final del método, como hemos comentado.

El valor de retorno es opcional, si lo hubiera debería de ser del mismo tipo o de un tipo compatible al tipo del valor de retorno definido en la cabecera del método, pudiendo ser desde un entero a un objeto creado por nosotros. Si no lo tuviera, el tipo de retorno sería **void**, y **return** serviría para salir del método sin necesidad de llegar a ejecutar todas las instrucciones que se encuentran después del **return**.



#### Para saber más

En el siguiente archivo java encontrarás el código de un programa que obtiene la suma de dos números, empleando para ello un método sencillo que retorna el valor de la suma de los números que se le han pasado como parámetros. Presta atención a los comentarios y fíjate en las conversiones a entero de la entrada de los operandos por consola.

```
import java.io.*;

public class sentencia_return {

    private static BufferedReader stdin = new BufferedReader( new InputStreamReader(System.in) );

    public static int suma(int numero1, int numero2)
    {
        int resultado;
        resultado = numero1 + numero2;
        return resultado; //Mediante return devolvemos el resultado de la suma
    }

    public static void main(String[] args) throws IOException {
        //Declaración de variables
        String input; //Esta variable recibirá la entrada de teclado
        int primer_numero, segundo_numero; //Estas variables almacenarán los operandos

        // Solicitamos que el usuario introduzca dos números por consola
        System.out.print ("Introduce el primer operando:");
        input = stdin.readLine(); //Leemos la entrada como cadena de caracteres
        primer_numero = Integer.parseInt(input); //Transformamos a entero lo introducido
    }
}
```

```
System.out.print ("Introduce el segundo operando: ");
input = stdin.readLine(); //Leemos la entrada como cadena de caracteres
segundo_numero = Integer.parseInt(input); //Transformamos a entero lo introducido

//Imprimimos los números introducidos
System.out.println ("Los operandos son: " + primer_numero + " y " + segundo_numero);
System.out.println ("obteniendo su suma... ");

//Invocamos al método que realiza la suma, pasándole los parámetros adecuados
System.out.println ("La suma de ambos operandos es: " + suma(primer_numero,segundo_numero));

}
```



## Autoevaluación

¿Qué afirmación es correcta?

- Con `return`, se puede finalizar la ejecución del método en el que se encuentre.
- Con `return`, siempre se retornará un valor del mismo tipo o de un tipo compatible al definido en la cabecera del método.
- Con `return`, puede retornarse un valor de un determinado tipo y suele hacerse al final del método. Además, el resto de respuestas también son correctas.