

TEMA 2: Primeros programas.

Uso de estructuras de control.

Sumario

.....	1
Introducción.....	2
¿Cómo funciona un programa en Java?.....	2
Herramientas para desarrollar con Java.....	5
NetBeans.....	7
Variables.....	12
Convenios y reglas para nombrar variables.....	12
Tipos de datos simples.....	17
Números enteros.....	17
Números reales.....	18
Overflow.....	19
Caracteres y cadenas.....	19
Datos lógicos.....	20
Literales en java.....	21
Constantes.....	25
Conversiones de tipo (casting).....	25
Operaciones con datos.....	27
Operaciones aritméticas.....	27
Comentarios.....	36
Vida y ámbito de las variables.....	37
Introducir datos por teclado.....	38
Estructuras Selectivas (Condicionales).....	39
Condional simple (IF).....	39
Condional doble (if else).....	41
IF ELSE IF.....	43
Condional múltiple.....	46
Estructuras repetitivas (bucles).....	51
Bucle Mientras... hacer (bucle while).....	52
Bucle Hacer... mientras (Bucle do while).....	53
Bucle “para” (Bucle for).....	55
Contadores, acumuladores.....	60
Instrucciones de salto.....	67

Introducción

¿Cómo funciona un programa en Java?

En el diseño original del lenguaje Java estaba la premisa de hacer un lenguaje altamente portable (como son todos los lenguajes interpretados) y, al mismo tiempo, altamente eficiente (como son, por regla general, los lenguajes compilados). Prueba de que lo consiguieron, o que estuvieron cerca de ello, es la ubicuidad actual del lenguaje Java en todo tipo de soportes y plataformas.

Para conseguirlo, los diseñadores de Java idearon lo que podríamos denominar una *semicompilación*, de modo que el código fuente se compila en un código binario, pero no el código binario de una máquina real, sino el de una máquina ficticia. Ese código binario se denomina **bytecode**.

Esa máquina ficticia, con su CPU, sus registros y todo su hardware, se emula mediante un software especial denominado máquina virtual de Java (JVM = Java Virtual Machine). La JVM toma el código binario del bytecode y lo interpreta, traduciéndolo en tiempo de ejecución al código binario real del sistema sobre el que se está trabajando.

La ventaja de este procedimiento es que la traducción del bytecode al código binario real es mucho más rápida que una interpretación tradicional, porque el bytecode ya es un código binario. Por lo tanto, la JVM se limita a formatear las instrucciones del bytecode para hacerlas comprensibles por la máquina real, y no tiene que realizar una traducción completa.

Por ese motivo, un programa escrito en Java y compilado en una arquitectura cualquiera, funcionará sin problemas al ejecutarlo en cualquier otra arquitectura, con el único requisito de haber instalado la JVM en el ordenador de destino.

Todo este proceso se describe en las siguientes figuras. Por ejemplo, supongamos que tenemos una plataforma Intel con un sistema Windows y el JDK instalado, y en ella desarrollamos una aplicación escrita en Java y la compilamos:



Fig. El compilador de Java convierte el código fuente en código objeto para la máquina virtual de Java (bytecode).

Hemos obtenido un programa “semicompilado” en un código binario de una máquina virtual. Nótese que hubiéramos obtenido exactamente lo mismo si estuviéramos trabajando bajo GNU/Linux, MacOS, Android o cualquier otro sistema.

Ahora, nos llevamos ese programa “semicompilado” a alguna otra plataforma, digamos un smartphone con Android y la JVM correctamente instalada. Bastará con lanzar la aplicación para que la

JVM tome el control y realice la traducción final, al modo de los antiguos intérpretes, del bytecode al código binario nativo de nuestro smartphone:

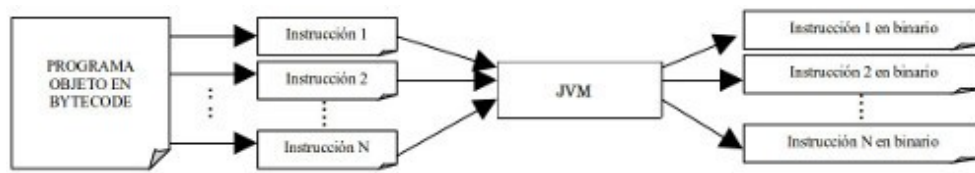


Fig. La máquina virtual de Java (JVM) interpreta el bytecode durante la ejecución del programa.

Este doble proceso de “semicompilación” seguida de “semiinterpretación” consigue el doble objetivo planteado en el diseño de Java: ser un lenguaje altamente portable al mismo tiempo que razonablemente eficiente.

Máquinas virtuales

Para entender completamente el funcionamiento de Java y su singular proceso de semicompilación, es necesario tener claro en concepto de máquina virtual, que introducimos a continuación.

Una máquina virtual es un programa informático que emula a un ordenador y puede ejecutar programas como si fuese un ordenador real. La máquina virtual puede emular un ordenador real o ficticio (esto último puede tener sentido con propósitos didácticos)

Una característica esencial de las máquinas virtuales es que los procesos que ejecutan están limitados por los recursos proporcionados por la máquina virtual. Es decir, si la máquina virtual "tiene" 1 GB de RAM, por ejemplo, los programas que ejecutemos en ella sólo pueden usar 1 GB, independientemente de que la máquina real tenga disponible más memoria física.

Uno de los usos domésticos más extendidos de las máquinas virtuales es ejecutar sistemas operativos para "probarlos". De esta forma podemos ejecutar un sistema operativo que queramos probar (GNU/Linux, por ejemplo) desde nuestro sistema operativo habitual (MacOS, por ejemplo) sin necesidad de instalarlo directamente en nuestra computadora y sin miedo a que se desconfigure el sistema operativo primario.

Tipos de máquina virtual

Las máquinas virtuales se pueden clasificar en dos grandes categorías según su funcionalidad y su grado de equivalencia a una verdadera máquina.

- Máquinas virtuales de sistema (en inglés System Virtual Machine)
- Máquinas virtuales de proceso (en inglés Process Virtual Machine)

A) Máquinas virtuales de sistema

Las máquinas virtuales de sistema, también llamadas máquinas virtuales de hardware, permiten a la máquina física "dividirse" entre varias máquinas virtuales, cada una ejecutando su propio sistema operativo.

Estas máquinas virtuales permiten a varios sistemas operativos distintos pueden coexistir sobre el mismo hardware, completamente aisladas unas de otras, si bien compartirán los recursos de la máquina física. Esto permite reducir el coste total de las instalaciones necesarias para mantener los mismos servicios, dado que hay un ahorro considerable en hardware, energía, mantenimiento, espacio, etc.

La mayoría de los programas de virtualización conocidos pertenecen a esta categoría de máquina virtual. Entre los más famosos están:

- VMWare. Ha sido el software de virtualización por excelencia durante muchos años. Es software privativo, aunque tiene una versión gratuita. Funciona en Windows, Linux y Mac.
- VirtualBox de Oracle (antes Sun). Funciona también en Windows, Linux y Mac. Tiene una versión Open Source (VirtualBox OSE).
- VirtualPC. Está desarrollado por Microsoft, por lo que es una excelente opción para virtualizar sistemas Windows. No funciona bajo Linux, aunque sí bajo Mac.

B) Máquinas virtuales de proceso

Una máquina virtual de proceso, a veces llamada "máquina virtual de aplicación", se ejecuta como un proceso normal dentro de un sistema operativo y soporta un solo proceso. La máquina se inicia automáticamente cuando se lanza el proceso que se desea ejecutar y se detiene para cuando éste finaliza. Su objetivo es el de proporcionar un entorno de ejecución independiente de la plataforma de hardware y del sistema operativo, que oculte los detalles de la plataforma subyacente y permita que un programa se ejecute siempre de la misma forma sobre cualquier plataforma.

La máquina virtual Java es de este tipo. Por lo tanto, cada vez que lanzamos un programa compilado en el bytecode de Java, la JVM emula el hardware necesario para realizar la ejecución (interpretación) del código y ejecutar así la aplicación independientemente del hardware real. Otra máquina virtual con la misma filosofía y muy extendida es la del entorno .Net de Microsoft.

Herramientas para desarrollar con Java

Para hacer un programa en Java únicamente se precisa un editor de código y los JDK y JRE:

El JDK es el Java Development Kit, que traducido al español es, Herramientas de desarrollo para Java, aquí nos encontraremos con el compilador javac que es el encargado de convertir nuestro código fuente (.java) en bytecode (.class), el cual posteriormente será interpretado y ejecutado con la JVM, Java Virtual Machine por sus siglas en inglés, que nuevamente al español es La Máquina Virtual de Java, también dentro de estas herramientas encontramos los siguientes programas, javadoc(encargado de generar la documentación de nuestro código), el jvisualvm(muestra información a detalle sobre las aplicaciones que están corriendo actualmente en la JVM), entre muchas otras.

El JRE es el Java Runtime Environment, en español es el Entorno de Ejecución de Java, en palabras del propio portal de Java es la implementación de la Máquina virtual de Java que realmente ejecuta los programas de Java, esto quiere decir que aquí encontraremos todo lo necesario para ejecutar nuestras aplicaciones escritas en Java, normalmente el JRE está destinado a usuarios finales que no requieren el JDK, pues a diferencia de este no contiene los programas necesarios para crear aplicaciones en el lenguaje Java, es así, que el JRE se puede instalar sin necesidad de instalar el JDK, pero al instalar el JDK, este siempre cuenta en su interior con el JRE.

¿ Podemos instalar el JDK sin el JRE ? , la respuesta es no.

¿ Podemos instalar el JRE sin el JDK ? , la respuesta es sí si únicamente se quieren ejecutar programas. No si lo que se quiere es desarrollar y ejecutar programas. Esto es debido a que el JDK está destinado a usuarios que requieran crear aplicaciones en el lenguaje java.

¿ Entonces si instalamos el JDK el JRE y un editor de texto, como el propio que viene ya instalado en windows llamado notepad ya no tengo que instalar nada más ?, Lo cierto es que en rigor no se precisa de más pero lo cierto es que los editores de texto, son más sencillos que los IDE, y por tanto tienen menos funcionalidades integradas. La cantidad de funciones básicas depende de cada editor, pero generalmente tienen algún tipo de resaltado y formateo de código. Hoy en día es muy común que se puedan ampliar con extensiones de todo tipo, aunque eso hace que sean un poco más pesados y lentos, alejándose del concepto de editor de texto y acercándose más al de IDE. La idea es que un editor sea rápido y liviano. Ejemplos de editores: Vim, Sublime, Atom, Visual Studio Code, Notepad++

Para más detalles respecto a JRE, JDK y todos los productos java, visitar:

<http://www.oracle.com/technetwork/java/javase/terms/products/index.html>

¿ Qué es un IDE ?

Un entorno de desarrollo integrado o entorno de desarrollo interactivo, en inglés Integrated Development Environment (IDE), es una aplicación informática que proporciona servicios integrales para facilitarle al desarrollador o programador el desarrollo de software.

Normalmente, un IDE consiste de un editor de código fuente, herramientas de construcción automáticas y un depurador. La mayoría de los IDE tienen auto-completado inteligente de código

(IntelliSense). Algunos IDE contienen un compilador, un intérprete, o ambos, tales como NetBeans y Eclipse

Los IDE están diseñados para maximizar la productividad del programador proporcionando componentes muy unidos con interfaces de usuario similares. Los IDE presentan un único programa en el que se lleva a cabo todo el desarrollo.

NetBeans

NetBeans es un IDE libre, hecho principalmente para el lenguaje de programación Java. Existe además un número importante de módulos para extenderlo. NetBeans IDE2 es un producto libre y gratuito sin restricciones de uso.

La página principal de NetBeans:

<https://netbeans.org/>

Crear un proyecto en Netbeans:

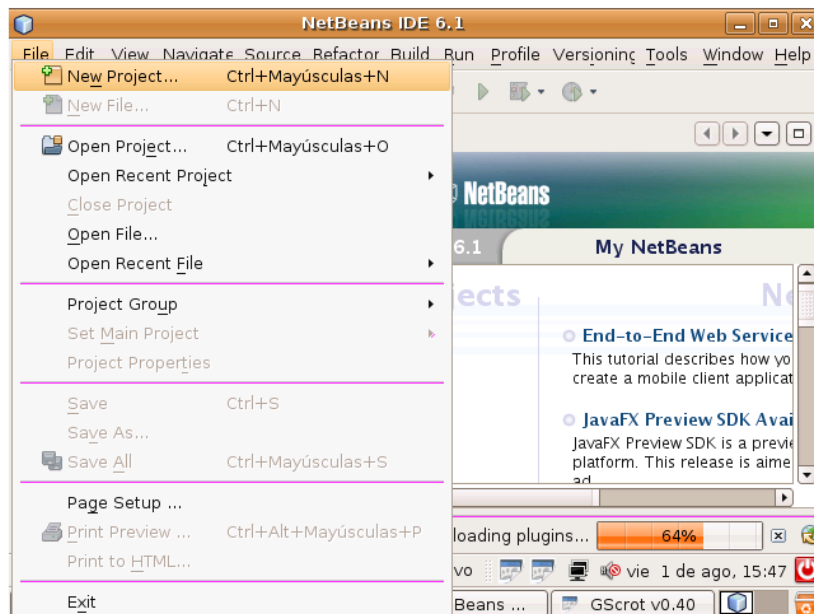
NetBeans nos permite crear proyectos java de diversos tipos(Escritorio, Web, Moviles). También tiene soporte para otros lenguajes. Estos proyectos poseen una estructura definida, que permiten al IDE un manejo adecuado de los mismos.

1. Crear un proyecto nuevo es muy sencillo, lo podemos realizar desde el menu/ File New

Project. o Seleccionando el icono

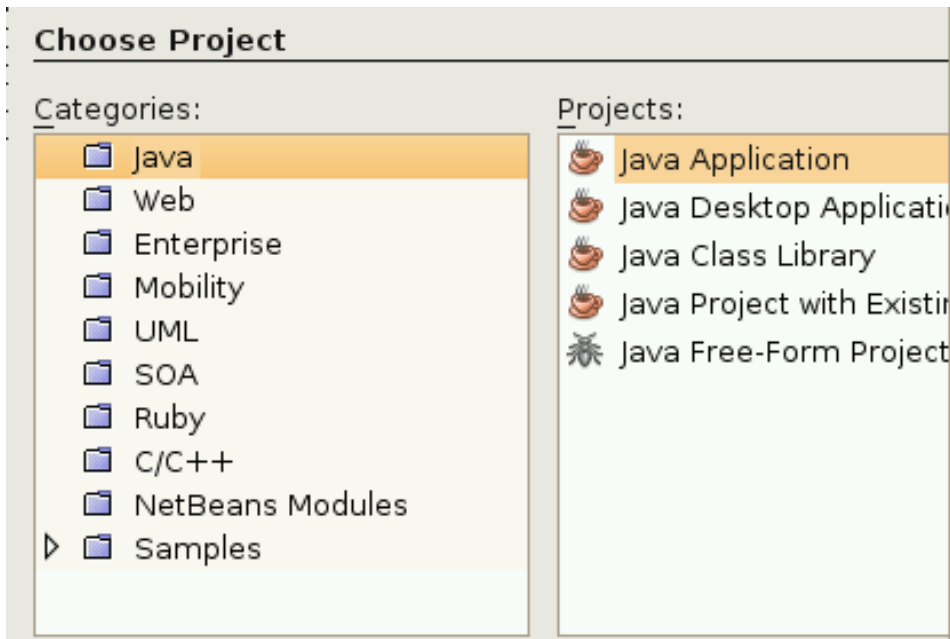


- Menu de Proyecto Nuevo



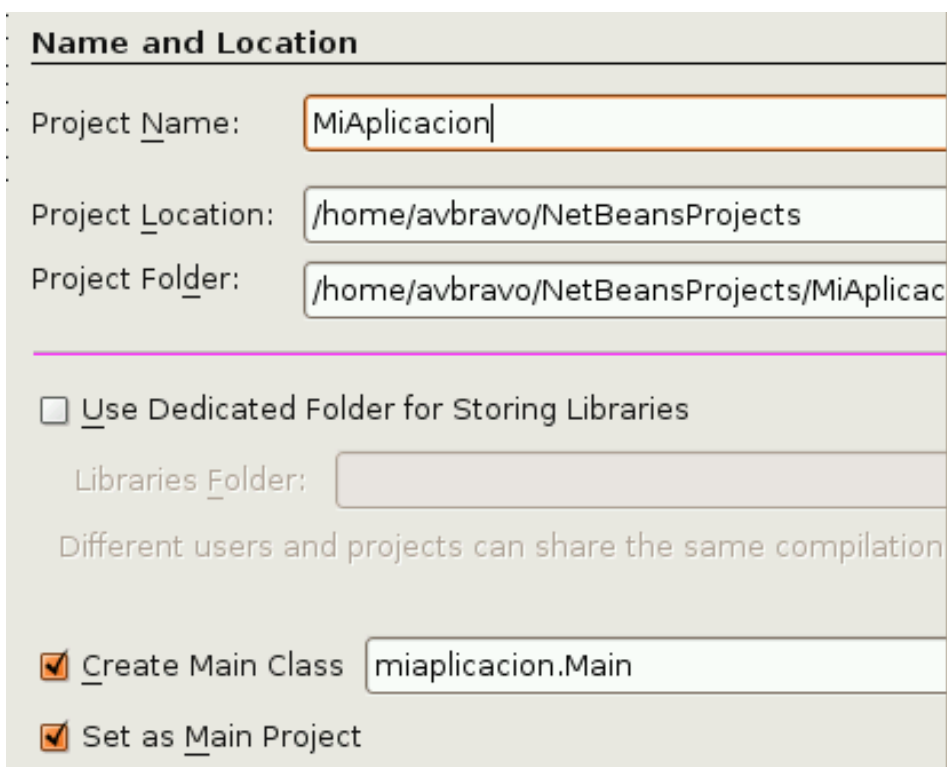
- Ahora seleccionamos el tipo de proyecto

Escogemos Java. Java Application. Ya que deseamos crear un proyecto de escritorio.



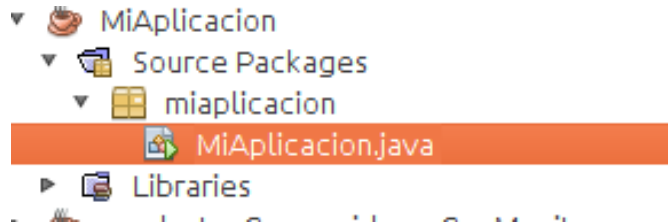
Presionamos el botón siguiente.

- Ahora le damos el nombre al proyecto



Presionamos el boton FINISH

- El IDE genera la estructura del proyecto
- Se crea la clase principal



Código que genera el IDE

```
/**
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package miaplicacion;

/**
 *
 * @author carlos
 */
public class MiAplicacion {

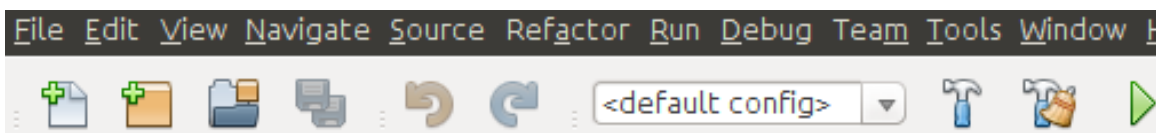
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
    }

}
```

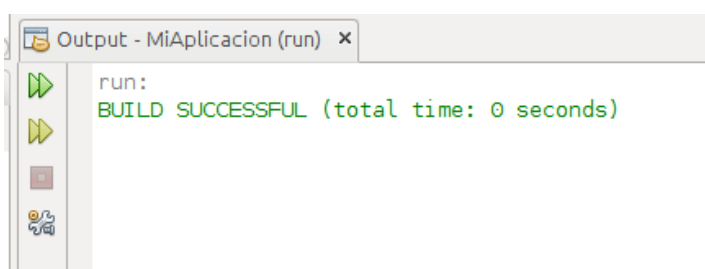
Si ejecutáramos ahora la aplicación, pulsando F6 o sobre la tecla verde:



que está en la barra de herramientas:



Observamos que no nos muestra mensajes en pantalla después del run:



Vamos a realizar el clásico Hola Mundo de programación. Para ello modificamos en el método main del fichero MiAplicacion.java:

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package miaplicacion;

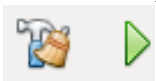
/**
 *
 * @author carlos
 */
public class MiAplicacion {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        System.out.println("Hola Mundo!, mi nombre es Juan Carlos");
    }

}
```

Si ahora pulsamos en run obtendremos el mensaje en pantalla.

Una opción que usaremos mucho es “Clean and build project” que accedemos mediante Mayusc+F11 o con el icono que está a la izquierda de la tecla verde de run:



Lo que hace es borrar las clases que han sido compiladas previamente y otros artefactos, luego reconstruye el proyecto entero desde cero:

- Los directorios de salida que han sido generados son borrados (en la mayoría de los casos son los directorios son build y dist)
- Los directorios build y dist son añadidos al directorio del proyecto (PROJECT_HOME).
- Todos los archivos fuentes son compilados en archivos .class y son colocados en el directorio PROJECT_HOME/build.
- Un archivo JAR conteniendo todo el proyecto es creado en el directorio PROJECT_HOME/build.
- Si has especificado cualquier librería para el proyecto (adicional al JDK) un directorio lib es creado en el directorio dist. Las librerías son copiadas en dist/lib.

- El archivo manifiesto en el JAR es actualizado para incluir entradas que designan la clase principal (main) y cualquier librería que están en la ruta de clases (classpath) del proyecto

Muchas veces haremos uso de Clean and build y luego Run Project

El aprendizaje de Netbeans lo iremos realizando a medida que hagamos programas.

● **Práctica 1:** Crear un proyecto NetBeans que muestre en pantalla el mensaje:
Hola Mundo! Soy nombre apellido1 apellido2 y este es mi primer programación
Tomar una captura de pantalla de la ejecución del programa y parte del código

Observa que si la línea no la terminas con punto y coma: “;” el programa no compila. Esto es algo que vamos a tener que hacer **SIEMPRE** Todas las sentencias en Java terminan en punto y coma

Observa también esta instrucción: `public static void main(String[] args) {`

Para todo programa que realicemos debe haber una sentencia como esa. Es el método main()
El método main() es el punto de partida de todo programa.

Variables

Un programa maneja datos para hacer cálculos, presentarlos en informes por pantalla o impresora, solicitarlos al usuario, guardarlos en disco, etc. Para poder manejar esos datos, el programa los guarda en variables.

Una variable es una zona en la memoria del computador con un valor que puede ser almacenado para ser usado más tarde en el programa. Las variables vienen determinadas por:

- Un nombre, que permite al programa acceder al valor que contiene en memoria. Debe ser un identificador válido.

- Un tipo de dato, que especifica qué clase de información guarda la variable en esa zona de memoria

- Un rango de valores que puede admitir dicha variable.

Convenios y reglas para nombrar variables.

El compilador aceptará los nombres que sean creados que cumplan:

- Contiene cualquier carácter Unicode, pero no puede comenzar con un número
- No debe contener los símbolos que se utilicen como operadores (+ , - , ? , % , / , * , ! , < , > , ~ , etc). En general en general, no contener símbolos especiales excepto el subrayado (" _ ")

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.

Como regla general, empezarán por una letra.

No se puede utilizar el valor booleano (true o false) ni el valor nulo (null). Todas ellas son palabras reservadas

Palabras reservadas 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

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.

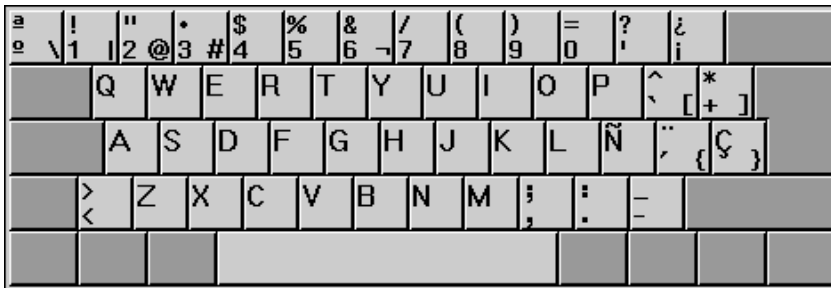
Identificador	Convención	Ejemplo
Nombre de variable.	Comienza por letra minúscula, y si tienen más de una palabra se colocan juntas y el resto comenzando por mayúsculas.	numAlumnos, suma
Nombre de constante.	En letras mayúsculas, separando las palabras con el guión bajo, por convenio el guión bajo no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

Motivos para no usar ñ, á, é, í, ó, ú y otros símbolos españoles en los nombres de variables.

- Trabajar con un equipo internacional.

Cuando trabajas en un equipo internacional, utilizar según qué caracteres puede ser una pesadilla para otros miembros del equipo.

Veamos un teclado como el que pudiéramos usar nosotros habitualmente:



Y tu compañero de trabajo Norteamericano usa este:



Para tu compañero va a ser un problema acceder a la Ñ. Y si trabajas con equipos que tienen alfabetos muy diferentes el problema crece exponencialmente:

```
int 入玉口千 = 0xFF800000;  
int 人王口千 = 0xF8000000;
```

Las anteriores son dos variables diferentes. ¿no sería muy sencillo que alguien que no hable japonés se equivocara de variable ?

En ejemplo con español:

```
Decimal año = 2000; // 365 días agrupados en una unidad  
Decimal ano = 2016; // Parte anatómica del ser humano
```

Para un hispanoablante nativo no hay duda léxico-semántica. Quizás si para nosotros pudiera ser complicada la distinción del anterior ejemplo en japonés para otros sin dominio del español pudiera ser complicado nuestra anterior definición.

```
bool flag = false; // это не является необходимым
```

El comentario anterior está en ruso. Fijarse en la dificultad que puede inferir hasta encontrar el correcto significado de lo que pudiera haber escrito un compañero ruso de nuestro equipo de trabajo.

- El problema de buscar la exactitud aumentando la complejidad de codificar

La siguiente definición de variable es perfectamente aceptada por el compilador.

```
double  $\pi$  = 3.14159265;
```

Aún siendo π el símbolo correcto para esa constante, escribirlo en el código es incómodo. No es nada habitual un teclado que disponga de ese símbolo

- La codificación y los sistemas de versionado.

Veremos la utilidad de disponer de repositorios de versiones de código en la nube, que nos permite trabajar desde múltiples sitios con diferentes aproximaciones para resolver un proyecto de código y retornar a puntos de trabajo que hemos determinado como seguros previamente si las cosas no van por el camino que queríamos. Adicionalmente de la gran versatilidad que tiene para soportar múltiples programadores trabajando sobre el mismo código

Se han dado casos en que al subir el código al sistema de control de versiones, los archivos con *caracteres extraños* se corrompían. Si bien, esto es algo que se puede controlar configurando adecuadamente el sistema de control de versiones, este puede ser un motivo adicional para no usar caracteres de este tipo

● **Práctica 2:** Dados los siguientes identificadores, indicar si son válidos o no. Justificar las respuestas. Si algún caso son válidos pero no son recomendables por algún motivo detallarlo.

mi variable
num_de_cte
____programa
3tema
cierto?
númerodeCliente
jose~
año
PI
int

Las variables responde a un tipo de datos como recién hemos dicho. El momento en el que establecemos ese tipo de datos nos hace una diferenciación entre lenguajes de programación que soporten tipado estático y tipado dinámico

Tipado estático

Se dice de un lenguaje de programación que usa un tipado estático cuando la comprobación de tipificación se realiza durante la compilación, y no durante la ejecución. Ejemplos de lenguajes que usan tipado estático son C, C++, Java . Comparado con el tipado dinámico, el estático permite que los errores de tipificación sean detectados antes, y que la ejecución del programa sea más eficiente y segura.

Tipado dinámico

Se dice de un lenguaje de programación que usa un tipado dinámico cuando la comprobación de tipificación se realiza durante su ejecución en vez de durante la compilación. Ejemplos de lenguajes que usan tipado dinámico son Perl, Python y Lisp. Comparado con el tipado estático, o sistema de enlazado temprano, el tipado dinámico es más flexible (debido a las limitaciones teóricas de la decidibilidad de ciertos problemas de análisis de programas estáticos, que impiden el mismo nivel de flexibilidad que se consigue con el tipado estático), a pesar de ejecutarse más lentamente y ser más propensos a contener errores de programación.

Tipos de datos simples

Los datos, son representaciones de los objetos del mundo real. Esos objetos pueden ser simples (por ejemplo, la edad de una persona, o el número de trabajadores de una empresa) o complejos (por ejemplo, la flota de camiones de una empresa de transportes).

Nosotros nos referiremos ahora a los tipos de datos simples. Son importantes porque los datos más complejos se fundamentan en ellos, y es necesario informar a Java de cuáles son los tipos que vamos a usar porque necesita saberlo para reservar los recursos necesarios para almacenarlos (principalmente, memoria RAM)

Un último concepto antes de continuar: se entiende por tipo de datos el dominio en el que un determinado dato puede tomar un valor. Así, determinada variable puede ser de tipo “número entero”, lo que significa que sólo puede contener números sin decimales, o puede ser de tipo “cadena alfanumérica”, que significa que puede contener un número indefinido de caracteres alfanuméricos.

En Java, se llama “tipo de datos primitivo” a lo que en otros lenguajes se llama “tipo de datos simple” (en realidad, estos tipos primitivos son clases, pero como aún no hemos visto las clases, no te preocupes de ello por el momento). Cada tipo de datos, además, tiene asociado un conjunto de operaciones para manipularlos.

Cada tipo de datos, insistimos, dispone de una representación interna diferente en el ordenador; por eso es importante distinguir entre tipos de datos a la hora de programar.

Los tipos primitivos de Java son:

- Números enteros
- Números reales
- Caracteres
- Lógicos

Así, por ejemplo, en el caso de un programa de gestión de nóminas, la edad de los empleados será un dato de tipo número entero, mientras que el dinero que ganan al mes será un dato de tipo número real.

Números enteros

Es probablemente el tipo más sencillo de entender. Los datos de tipo entero sólo pueden tomar como valores:

..., -4, -3, -2, -1, 0, 1, 2, 3, 4, ...

Como el ordenador tiene una memoria finita, la cantidad de valores enteros que puede manejar también es finita y depende del número de bits que emplee para ello (recuerda que el ordenador, internamente, representa todos los datos en binario).

Además, los enteros pueden ser con signo y sin signo. Si tienen signo, se admiten los números negativos; si no lo tienen, los números sólo pueden ser positivos (sería más correcto llamarlos números naturales).

(Los enteros con signo se almacenan en binario en complemento a uno o en complemento a dos. No nos detendremos en este libro a detallar en qué consisten esas representaciones).

Por lo tanto:

- Si se utilizan 8 bits para codificar los números enteros, el rango de valores permitido irá de 0 a 255 (sin signo) o de -128 a +127 (con signo).
- Si se utilizan 16 bits para codificar los números enteros, el rango será de 0 a 65535 (sin signo) o de -32768 a 32767 (sin signo).
- Si se utilizan 32 bits, el rango será de 0 a más de 4 mil millones (sin signo), o de menos dos mil millones a más dos mil millones (aproximadamente) con signo.
- Si se utilizan 64, 128 bits o más, se pueden manejar números enteros mayores. Puedes calcular los rangos de números resultantes y sentir escalofríos.

Los tipos enteros primitivos en Java son:

- **byte**: entero de 8 bits con signo.
- **short**: entero de 16 bits con signo.
- **int**: entero de 32 bits con signo.
- **long**: entero de 64 bits con signo.

Estas representaciones son independientes de la plataforma, a diferencia de lo que ocurre con otros lenguajes, en los que un tipo de datos puede tener una longitud distinta en cada sistema.

Con la siguiente instrucción crearíamos una variable de tipo entero de 32 bits y lo mostraríamos en pantalla:

```
int numeroEntero;  
numeroEntero = 131072;  
System.out.println( numeroEntero );
```

Números reales

El tipo de dato primitivo llamado *número real* permite representar números con decimales. La cantidad de decimales de un número real puede ser infinita, pero al ser el ordenador una máquina finita es necesario establecer un número máximo de dígitos decimales significativos.

La representación interna de los números reales se denomina coma flotante (también existe la representación en coma fija, pero no es habitual). La coma flotante es una generalización de la notación científica convencional, consistente en definir cada número con una mantisa y un exponente.

Internamente, el ordenador reserva varios bits para la mantisa y otros más para el exponente. Como en el caso de los números reales, la magnitud de los números que el ordenador pueda manejar estará directamente relacionada con el número de bits reservados para su almacenamiento.

Java dispone de dos tipos primitivos para manejar números reales:

- **float**: coma flotante de 32 bits (1 bit de signo, 8 de exponente y 24 de mantisa)
- **double**: coma flotante de 64 bits (1 bit de signo, 11 de exponente y 52 de mantisa)

Overflow

Cuando se realizan operaciones con números (tanto enteros como reales), es posible que el resultado de una de ellas dé lugar a un número fuera del rango máximo permitido. Por ejemplo, si tenemos un dato de tipo entero sin signo de 8 bits cuyo valor sea 250 y le sumamos 10, el resultado es 260, que sobrepasa el valor máximo (255).

En estos casos, estamos ante un caso extremo denominado *overflow* o desbordamiento. Los ordenadores pueden reaccionar de forma diferente ante este problema, dependiendo del sistema operativo y del lenguaje utilizado. Algunos lo detectan como un error de ejecución del programa, mientras que otros lo ignoran, convirtiendo el número desbordado a un número dentro del rango permitido pero que, obviamente, no será el resultado correcto de la operación, por lo que el programa probablemente fallará.

En el caso de Java, la JVM proporcionará un error en tiempo de ejecución si se produce un desbordamiento. Ese error puede capturarse mediante una excepción para tratarlo adecuadamente. Veremos más adelante como hacer todo esto.

Caracteres y cadenas

El tipo de dato *carácter* sirve para representar datos alfanuméricos. El conjunto de elementos que puede representar está estandarizado según diferentes tablas de código (ASCII o Unicode). El estándar más antiguo es ASCII, que consiste en una combinación de 7 u 8 bits asociada a un carácter alfanumérico concreto, pero que está siendo sustituido por sus limitaciones en la representación de caracteres no occidentales.

Java proporciona el tipo **char**, de 16 bits, para manejar caracteres.

Las combinaciones de 7 bits del ASCII clásico dan lugar a un total de 127 valores distintos (desde 0000000 hasta 1111111). En Java, se siguen reservando esos 127 valores para la codificación ASCII. El resto, sirve para almacenar los caracteres en formato Unicode. Los caracteres válidos que pueden almacenarse en una variable tipo `char` son:

- Las letras minúsculas: 'a', 'b', 'c' ... 'z'
- Las letras mayúsculas: 'A', 'B', 'C' ... 'Z'
- Los dígitos: '1', '2', '3' ...
- Caracteres especiales o de otros idiomas: '\$', '%', '¿', '!', 'ç'...

Nótese que no es lo mismo el valor entero 3 que el carácter '3'. Para distinguirlos, usaremos siempre comillas para escribir los caracteres.

Los datos tipo carácter sólo pueden contener UN carácter. Una generalización del tipo carácter es el tipo cadena de caracteres, utilizado para representar series de varios caracteres. Éste, sin embargo, es un objeto complejo y será estudiado más adelante. Sin embargo, las cadenas se utilizan tan a menudo que no podremos evitar usarlas en algunos ejercicios antes de estudiarlas a fondo.

Datos lógicos

El tipo dato *lógico*, también llamado booleano en honor a George Boole (un matemático británico que desarrolló en el siglo XIX una rama del álgebra llamada lógica o de Boole) es un dato que sólo puede tomar un valor entre dos posibles.


Esos dos valores son:

- Verdadero (en inglés, *true*)
- Falso (en inglés, *false*)

Este tipo de datos se utiliza para representar alternativas del tipo sí/no. En algunos lenguajes, el valor false se representa con el número 0 y el valor true con el número 1 (o con cualquier número distinto de 0).

Lo que ahora es importante comprender es que los datos lógicos contienen información binaria. Esto ya los hace bastante notables, pero la mayor utilidad de los datos lógicos viene por otro lado: son el resultado de todas las operaciones lógicas y relacionales, como veremos en el siguiente epígrafe.

En Java, los datos booleanos se manejan mediante el tipo **boolean**.

 **Práctica 3:** Determinar que tipo de datos es más apropiado en cada caso para guardar la siguiente información:

1. Si un empleado está casado o no.
2. 999999.
3. Día de la semana
4. Día del año.
5. Sexo: con dos valores posibles 'V' o 'M'
6. Milisegundos transcurridos desde el 01/01/1970 hasta nuestros días.
7. Almacenar el total de una factura
8. Población mundial del planeta tierra.

Literales en java

Un literal Java es un valor de tipo entero, real, lógico, carácter, cadena de caracteres o un valor nulo (null) que puede aparecer dentro de un programa.

Por ejemplo: **150**, **12.4**, **“ANA”**, **null**, **‘t’**.

LITERAL JAVA DE TIPO ENTERO

Puede expresarse en decimal (base 10), octal (base 8) y hexadecimal (base 16).

El signo + al principio es opcional y el signo – será obligatorio si el número es negativo.

El tipo de un literal entero es int a no ser que su valor absoluto sea mayor que el de un int ó se especifique el sufijo l ó L en cuyo caso será de tipo long.

Así: **123L** es de tipo long

Literal Java de tipo entero en Base decimal

Está formado por 1 o más dígitos del 0 al 9.


El primer dígito debe ser distinto de cero.

Por ejemplo:

1234 literal java entero de tipo int

1234L literal java entero de tipo long

Si un literal tiene tantas cifras que supera los límites de un número entero. Debemos especificar que es de tipo long agregándole la letra L

 **Práctica 4:** Introducir en un código java de NetBeans la siguiente línea:

long variable = 123400000000;

¿qué mensaje muestra el IDE?

Literal Java de tipo entero en Base octal

Está formado por 1 o más dígitos del 0 al 7.

El primer dígito debe ser cero.

Por ejemplo:

01234

025

Literal Java de tipo entero en Base hexadecimal

Está formado por 1 o más dígitos del 0 al 9 y letras de la A a la F (mayúsculas o minúsculas).


Debe comenzar por 0x ó 0X.

Por ejemplo:

0x1A2

0x430

0xf4

 **Práctica 5:** Introducir en un código java de NetBeans las siguientes línea:

int numero = 0x10;

System.out.println(numero);

¿qué número aparece en pantalla?

Pon ahora la siguiente línea:

System.out.println(067);

¿ qué número muestra ahora ? ¿67?

LITERAL JAVA DE TIPO REAL

Son números en base 10, que deben llevar un parte entera, un punto decimal y una parte fraccionaria. Si la parte entera es cero, puede omitirse.

El signo + al principio es opcional y el signo – será obligatorio si el número es negativo.

Por ejemplo:

12.2303

-3.44

+10.33

0.456

.96

También pueden representarse utilizando la notación científica. En este caso se utiliza una E (mayúscula o minúscula) seguida del exponente (positivo o negativo). El exponente está formado por dígitos del 0 al 9.

Por ejemplo, el número en notación científica $14 \cdot 10^{-3}$ se escribe: 14E-3

Más ejemplos de literal Java de tipo real:

2.15E2 $\rightarrow 2.15 * 10^2 \rightarrow 2.15 * 100 \rightarrow 215$

.0007E4 $\rightarrow 0.0007 * 10^4 \rightarrow 0.0007 * 10000 \rightarrow 7.0$

-50.445e-10 $\rightarrow -50.445 * 10^{-10} \rightarrow -0.0000000050445$

El tipo de estos literales es siempre double, a no ser que se añada el sufijo F ó f para indicar que es float.

Por ejemplo:

2.15 literal real de tipo double

2.15F literal real de tipo float

También se pueden utilizar los sufijos d ó D para indicar que el literal es de tipo double:

12.002d literal real de tipo double

LITERAL JAVA DE TIPO CARÁCTER

Contiene un solo carácter encerrado entre comillas simples.

Es de tipo char.

Las secuencias de escape se consideran literales de tipo carácter.

Por ejemplo:

'a'

'4'

'\n'

LITERAL JAVA DE CADENAS DE CARACTERES

Está formado por 0 ó más caracteres encerrados entre comillas dobles.

Pueden incluir secuencias de escape.

Por ejemplo:

“Esto es una cadena de caracteres”

“Pulsa \"C\" para continuar” → observar que la barra invertida nos escapa las comillas

“” -> cadena vacía

“T” -> cadena de un solo carácter, es diferente a ‘t’ que es un carácter

Las cadenas de caracteres en Java son objetos de tipo String.

Secuencias de escape en Java

Acabamos de ver que para poder introducir dentro de una cadena de texto el símbolo de las comillas dobles debemos escaparlo mediante la barra invertida: “\” Veamos códigos de escape habituales

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

 **Práctica 6:** Escribe un programa que muestre tu horario de clase. Puedes usar espacios o tabuladores para alinear el texto.

Constantes

Los programas de ordenador contienen ciertos valores que no deben cambiar durante su ejecución. Estos valores se llaman constantes.

Podemos decir que una constante es una posición de memoria que se referencia con un identificador, conocido como nombre de la constante, donde se almacena el valor de un dato que no puede cambiar durante la ejecución del programa.

Una constante en Java se declara de forma similar a una variable, anteponiendo la palabra **final**

Ejemplo:

final double PI = 3.141591;

● **Práctica 7:** Introducir en un código java de NetBeans las siguientes línea:

final int MAXIMO = 23;

MAXIMO = 12;

¿muestra el IDE algo diferente ? ¿se puede ejecutar el programa?

Conversiones de tipo (casting)

Java es un lenguaje fuertemente tipado, por lo que suele reaccionar mal ante el intento de mezclar tipos de datos distintos en la misma expresión.

En general, en estos casos se puede hablar de dos tipos de conversión de datos:

A) Conversiones implícitas: se realizan de forma automática al mezclar tipos de datos. En Java solo puede hacerse si la variable receptora del resultado tiene más precisión que las variables situadas en la expresión. Por ejemplo, puede asignarse un int a un long, pero no al revés:

long a = 1, b = 7;

int x = 3, y = 6;

a = x; // Esta asignación funcionará

y = b; // Esta asignación fallará

B) Conversiones explícitas: el programador especifica mediante el código la conversión de un tipo en otro, indicando el nuevo tipo entre paréntesis durante la asignación. Este proceso se denomina *casting* y se muestra en el siguiente ejemplo:

int x = 5;

byte y;

y = (byte)x; // La variable entera x se convertirá a byte

Como un int puede contener números más grandes que un byte, esta última conversión puede suponer la pérdida de parte de la información y debe ser, en general, evitada.

● **Práctica 8:** Introducir en un código java de NetBeans las siguientes líneas:

```
int entero;  
double real = 3.5;  
entero = (int)real;  
System.out.println(entero);
```

¿ genera algún error ? ¿ qué número muestra ?

● **Práctica 9:** Sea: a, b enteros; c,d reales; e,f carácter; g de tipo lógico. ¿ qué asignaciones son válidas y cuáles no?. Justifica por qué no

- | | | |
|--------------|-----------------|-------------|
| a) a = 20; | b) b = 5500; | c) e = 'f'; |
| d) g = true; | f) b = 12; | g) c = 0; |
| h) e = f; | i) g = "false"; | j) f = '0'; |
| k) d = c | l) a = 12.56; | m) f = g; |

Operaciones con datos

Los datos que participan en una operación se llaman operandos, y el símbolo de la operación se denomina operador. Por ejemplo, en la operación entera $5 + 3$, los datos 5 y 3 son los operandos y "+" es el operador.

Podemos clasificar las operaciones básicas con datos en dos grandes grupos: las operaciones aritméticas y las operaciones lógicas.

Operaciones aritméticas

Son análogas a las operaciones matemáticas convencionales, aunque cambian los símbolos. Se emplean, en general, con datos de tipo entero o real:

Operación	Operador
suma	+
resta	-
multiplicación	*
división	/
módulo (resto)	%

Veamos la operación módulo (%) . Sirve para calcular el resto de la división entera. Es decir, si divides un número entero entre otro (por ejemplo, 5 entre 2), el cociente será otro número entero (2), y el resto será 1. Pues bien, el operador / te proporciona el cociente, y el operador % te proporciona el resto. Es un operador muy útil en una gran diversidad de circunstancias

El tipo del resultado de cada operación dependerá del tipo de los operandos. Por ejemplo, si sumamos dos números enteros, el resultado será otro número entero. En cambio, si sumamos dos números reales, el resultado será un número real. Aquí tenemos algunos ejemplos de operaciones aritméticas con números enteros y reales:

Operandos	Operador	Operación	Resultado
35 y 9 (enteros)	+	$35 + 9$	44 (entero)
35 y 9 (enteros)	-	$35 - 9$	26 (entero)
35 y 9 (enteros)	*	$35 * 9$	315 (entero)
35 y 9 (enteros)	/	$35 / 9$	3 (entero)
35 y 9 (enteros)	%	$35 \% 9$	8 (entero)
8,5 y 6,75 (reales)	+	$8,5 + 6,75$	15,25 (real)

8,5 y 6,75 (reales)	-	8,5 - 6,75	1,75 (real)
8,5 y 6,75 (reales)	*	8,5 * 6,75	57,375 (real)
8,5 y 6,75 (reales)	/	8,5 / 6,75	1,259 (real)

Nótese que el operador "-" también se usa para preceder a los números negativos, como en el álgebra convencional.

Hay, además, otros dos operadores aritméticos muy utilizados:

- Operador incremento (++): Se utiliza para aumentar en una unidad el valor de una variable numérica entera. Por ejemplo, **$x++$ es equivalente a $x = x + 1$.**
- Operador decremento (--): Se utiliza para disminuir en una unidad el valor de una variable numérica entera. La expresión **$x--$ es equivalente a $x = x - 1$.**

los operadores unarios incrementales y decrementales que acabamos de ver podemos utilizarlos con notación **prefija, si el operador aparece antes que el operando: $++x$; , o notación postfija, si el operador aparece después del operando $x++$; .**

Observa la siguiente línea de código:

$y = x++$;

Si lo pensamos nos damos cuenta que en esa línea se están realizando dos operaciones. Por un lado se está asignando el contenido de la variable x a la variable y . Por otro lado se está incrementando en 1 el valor de x .

Pues bien, la diferencia ente prefijo y postfijo se entiende mejor haciendo una comparación con el equivalente de las dos líneas que representarían las dos operaciones que hemos realizado:

	Prefijo	Postfijo
Expresión:	$y = x++$;	$y = ++x$;
Equivale a:	$y=x$; $x=x+1$;	$x=x+1$; $y=x$;

● **Práctica 10:** Introducir en un código java de NetBeans las siguientes líneas:

```
int x,y;
```

```
x=3;
```

```
y=++x;
```

```
System.out.println("Resultado con prefijo: x = " + x + " y = " + y);
```

```
x=3;
```

```
y=x++;
```

```
System.out.println("Resultado con postfijo: x = " + x + " y = " + y);
```

¿ qué resultado sale para los valores x e y ?

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
----------	-----------------	-----------------------

+=	op1 += op2	op1 = op1 + op2
----	------------	-----------------

-=	op1 -= op2	op1 = op1 - op2
----	------------	-----------------

*=	op1 *= op2	op1 = op1 * op2
----	------------	-----------------

/=	op1 /= op2	op1 = op1 / op2
----	------------	-----------------

%=	op1 %= op2	op1 = op1 % op2
----	------------	-----------------

● **Práctica 11:** Introducir en un código java de NetBeans las siguientes líneas:

```
int x,y;
```

```
x= 3;
```

```
y= 4;
```

```
y *= ++x;
```

```
System.out.println("x=" + x + " y=" + y);
```

¿ qué resultado sale para los valores x e y ? Escribe las líneas de código equivalentes si no usáramos el operador incremento: ++ ni el operador de asignación combinado: *=

Nota: Observa que en el ejercicio anterior hemos usado el operador + para mostrar textos y variables numéricas combinadas: "x=" + x + " y=" + y

Al solicitarle que haga esa operación Java transforma el número que tienen almacenado las variables x e y en su equivalente en texto y une las cadenas de texto:

Ej.

x=2; y=3;

Entonces

"x=" + x + " y=" + y

es igual a:

"x=" + "2" + " y=" + "3"

que queda finalmente:

"x=2 y=3"

- **Práctica 12:** Se tienen tres variables, a, b y c. Escribe las asignaciones necesarias para intercambiar sus valores, sean cuales sean, de manera que:
- b tome el valor de a
 - c tome el valor de b
 - a tome el valor de c

Operaciones lógicas

Estas operaciones sólo pueden dar como resultado verdadero o falso, es decir, su resultado debe ser un valor lógico.

Hay dos tipos de operadores que se utilizan en estas operaciones: los operadores de relación y los operadores lógicos.

1) Operadores de relación. **Son los siguientes:**

Operación	Operador
menor que	<
mayor que	>
igual que	==
menor o igual que	<=
mayor o igual que	>=
distinto de	!=

Los operadores de relación se pueden usar con todos los tipos de datos simples: entero, real, carácter o lógico. El resultado será verdadero si la relación es cierta, o falso en caso contrario.

Aquí tienes algunos ejemplos:

Operandos	Operador	Operación	Resultado
35, 9 (enteros)	>	35 > 9	verdadero
35, 9 (enteros)	<	35 < 9	falso
35, 9 (enteros)	==	35 == 9	falso
35, 9 (enteros)	!=	35 != 9	verdadero
5, 5 (enteros)	<	5 < 5	falso
5, 5 (enteros)	<=	5 <= 5	verdadero
5, 5 (enteros)	!=	5 != 5	falso
"a", "c" (caracteres)	==	'a' == 'c'	falso
"a", "c" (caracteres)	>=	'a' > 'c'	falso
"a", "c" (caracteres)	<=	'a' <= 'c'	verdadero



Práctica 13: Sea el siguiente programa:

```
public static void main(String[] args) {
    Scanner cin = new Scanner(System.in);
    System.out.println("Introduzca un número mayor que 5 ");
    int numero = cin.nextInt();
    boolean comparaCon5;
    comparaCon5 = numero > 5;
    System.out.println("Es " + comparaCon5 + " que el número sea mayor que 5");
}
```

Ejecutarlo e introducir un número mayor que 5 ¿ qué mensaje muestra ? Ahora introduce un número menor que 5 ¿ qué mensaje muestra ahora ?.

2) Operadores lógicos. Los operadores lógicos son and (y), or (o) y not (no). Sólo se pueden emplear con tipos de datos lógicos.

El operador and, que también podemos llamar y, se escribe en Java como &&, y da como resultado verdadero sólo si los dos operandos son verdaderos:

Operandos	Operador	Operación	Resultado
verdadero, falso	&&	verdadero && falso	falso
falso, verdadero	&&	falso && verdadero	falso
verdadero, verdadero	&&	verdadero && verdadero	verdadero
falso, falso	&&	falso && falso	falso

El operador or (también nos vale o) da como resultado verdadero cuando al menos uno de los dos operandos es verdadero. En Java se escribe ||

Operandos	Operador	Operación	Resultado
verdadero, falso		verdadero falso	verdadero
falso, verdadero		falso verdadero	verdadero
verdadero, verdadero		verdadero verdadero	verdadero
falso, falso		falso falso	falso

El operador not (o no), que se escribe !, es uno de los escasos operadores que sólo afectan a un operando (operador monario), no a dos (operador binario). El resultado es la negación del valor del operando, es decir, que le cambia el valor de verdadero a falso y viceversa

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:

```
(x>y)?x:y;
```

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**.

Por ejemplo, con la siguiente expresión obtenemos el valor más grande entre **x** e **y**:

```
( x > y ) ? x : y
```

Vamos a ver en un código cómo nos saldría en pantalla:

```
int x=3, y=4;  
System.out.println( (x>y)?x:y );
```

Después de asignar el 3 a la **x**, así como 4 a la **y** va a resolver la expresión:

(x>y)?x:y que en este caso nos va a dar el valor de **y**: 4 que será lo que nos aparecerá en pantalla.

● **Práctica 14:** Si tenemos las líneas de código:

```
int x=3, y=4;
```

```
int z = !(x<=y)?x:y ;
```

```
System.out.println(z);
```

¿qué valor nos va a mostrar en pantalla ?

Escribe ahora

```
System.out.println( 5 > 3 );
```

¿ qué muestra en pantalla ?

y ahora escribe:

```
System.out.println( 5 < 3 );
```

¿ qué muestra la pantalla ?

En la última actividad hemos visto que el operador negación (!) hizo su operación antes que el operador ternario: (?) Esto es así por la prioridad de operadores:

Prioridad de los operadores

Es habitual encontrar varias operaciones juntas en una misma línea. En estos casos es imprescindible conocer la prioridad de los operadores, porque las operaciones se calcularán en el orden de prioridad y el resultado puede ser muy distinto del esperado. Por ejemplo, en la operación $6 + 4 / 2$, no es lo mismo calcular primero la operación $6 + 4$ que calcular primero la operación $4 / 2$.

La prioridad de cálculo respeta las reglas generales del álgebra. Así, por ejemplo, la división y la multiplicación tienen más prioridad que la suma o la resta. La prioridad del cálculo se puede alterar usando paréntesis, como en el álgebra convencional. Los paréntesis se pueden anidar tantos niveles como sean necesarios. Por supuesto, a igualdad de prioridad entre dos operadores, la operación se calcula de izquierda a derecha, en el sentido de la lectura de los operandos.

En general, es una excelente idea hacer explícita la prioridad mediante paréntesis y no dejarla a merced de los deseos del lenguaje.

Además de todas estas operaciones aritméticas, lógicas y relacionales, los lenguajes de programación disponen de mecanismos para realizar operaciones más complejas con los datos, como, por ejemplo, calcular raíces cuadradas, logaritmos, senos, cosenos, redondeo de números reales, etc.

Todas estas operaciones (y muchas más) se realizan a través de objetos predefinidos de la biblioteca de clases de Java. Cuando llegue el momento, ya veremos en detalle muchas clases de esa biblioteca y cómo se usan, y aprenderemos a hacer las nuestras. Por ahora nos basta saber que algunas de ellas sirven para hacer cálculos más complejos que una simple suma o una división.

La ejecución de estos métodos predefinidos requiere el uso de una clase de la biblioteca estándar llamada `Math`, porque en ella se agrupan todos los métodos de índole matemática. Al método se le pasarán los parámetros necesarios para que haga sus cálculos, y nos devolverá el resultado.

Ej.

```
z = Math.abs(-7);  
System.out.println( z );
```

En ese código se obtiene el valor absoluto de un número

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. Ej.

```
// comentario de una sola 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.

```
/* Esto es un comentario
```

```
de varias líneas */
```

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.

```
/** Comentario de documentación.
```

```
Javadoc extrae los comentarios del código y
```

```
genera un archivo html a partir de este tipo de comentarios.
```

```
Para generar javadoc en Netbeans: Run →Generate javadoc
```

```
*/
```

Vida y ámbito de las variables

Las variables en Java son, por definición, todas locales al bloque en el que se han declarado, entendiendo por bloque a cualquier conjunto de instrucciones enmarcado entre una llave { y otra llave }

Esto quiere decir que la variable sólo existirá dentro del bloque y se destruirá cuando el bloque finalice, resultando inaccesible al resto del código.

En cambio, las variables miembros de una clase son accesibles en todo el código de la clase. Veremos más adelante este tipo de variables.

Veámoslo con un ejemplo. Pongamos este código en el IDE

```
class Prueba {  
  
    public static void main(String[] args) {  
        int m = 5, r = 0;    // Variables locales.  
        r = m + 7;  
        System.out.println("El resultado es: " + r);  
    }  
  
    public static void otroMetodo() {  
        int x = 2, r = 0;    // Más variables locales  
        r = x + m;           // El IDE nos mostrara error por variable no definida  
        System.out.println("El resultado es: " + r);  
    }  
}
```

En el código anterior está el método main() que como hemos dicho debe de estar en todo programa y es el punto de inicio de nuestro programa. Observamos que también hay otro método llamado: otroMetodo() Lo hemos creado para que veamos dos conjuntos de instrucciones que están delimitadas por llaves (es el motivo por el que hemos puesto las llaves más grandes)

El IDE mostrará error en el método: otroMetodo() por no estar definida la variable: m

Esto es por lo que habíamos comentado: la variable m que hemos definido mediante la instrucción: int m=5 tiene por ámbito el método main() El código ya no mostraría error si definiéramos dentro de otroMetodo() la variable m

Introducir datos por teclado

Hay varias formas para introducir datos por teclado. Vamos a usar de momento la clase Scanner.

```
public static void main(String[] args) {  
    Scanner cin = new Scanner(System.in);  
    System.out.print("Introducir un número: ");  
    int numero = cin.nextInt();  
    System.out.println("Has escrito el número: " + numero);  
}
```

En java tenemos accesible el teclado desde **System.in**, que es un InputStream del que podemos leer bytes. El problema es que después los bytes leídos se deben transformar en lo que necesitamos. Por suerte, si usamos Scanner nos simplifica el trabajo. En el ejemplo anterior se ha usado: `nextInt()` para tomar el siguiente entero que se haya introducido por teclado.

También tenemos disponible: `nextLong()`, `nextDouble()`, `nextLine()`, ...

● **Práctica 15:** Hacer un programa que lea una línea de texto entrada por teclado y la muestre en pantalla

● **Práctica 16:** Cálculo del área de una Finca. La finca es rectangular así que el usuario introducirá el número de metros de cada lado y se le mostrará el área calculada tanto en metros cuadrados como en centímetros cuadrados

● **Práctica 17:** Crea un programa que calcule el IGIC (7%) de un producto dado su precio de venta sin IGIC que introduzca el usuario por teclado.

● **Práctica 18:** Escribe un programa que calcule el salario semanal de un empleado en base a las horas trabajadas, a razón de 18 euros la hora

● **Práctica 19:** Realizar un programa que sirva para convertir grados Fahrenheit en centígrados. El usuario introducirá los grados Fahrenheit y se le mostrará la equivalencia en centígrados. La fórmula es:
$$C = (F - 32) * 5/9$$

donde C es grados centígrados y F es Fahrenheit

Estructuras Selectivas (Condicionales)

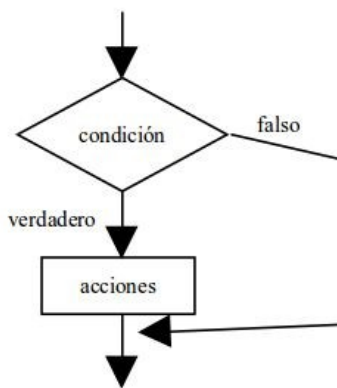
Los algoritmos que usan únicamente estructuras secuenciales están muy limitados y no tienen ninguna utilidad real. Esa utilidad aparece cuando existe la posibilidad de ejecutar una de entre varias secuencias de instrucciones dependiendo de alguna condición asociada a los datos del programa.

Las estructuras selectivas pueden ser de tres tipos:

- simples
- dobles
- múltiples

Condicional simple (IF)

La estructura condicional simple se escribe así:



```
if (condicion) {  
    acciones  
}
```

```
if( numero == 5 ){  
    int numero2 = numero;  
    System.out.println("numero vale cinco");  
    System.out.println("numero2 sólo se crea si numero vale 5");  
}
```

La condición que aparece detrás de "if" es siempre una expresión lógica, es decir, una expresión cuyo resultado es "verdadero" o "falso". Si el resultado es verdadero, entonces se ejecutan las acciones situadas entre { y }. Si es falso, se saltan las acciones y se prosigue por la siguiente instrucción (lo que haya debajo de la llave de cierre)

Aprovecharemos para entender un poco mejor el ámbito de las variables

En el ejemplo anterior hemos creado la variable numero2 dentro del bloque de llaves de la condición. Eso significa que esa llave no funciona fuera de esa condición. Probemos a poner este código en un proyectos Netbeans

```
public static void main(String[] args) {
    int numero=5;
    if( numero == 5 ){
        int numero2 = numero;
        System.out.println("numero vale cinco");
        System.out.println("numero2 sólo se crea si numero vale 5");
        System.out.println("numero vale: "+numero + "numero2 vale: "+numero2);
    }
    System.out.println("numero vale: "+numero + "numero2 vale: "+numero2);
}
```

El System.out.println() que está fuera de las llaves dará error por variable no definida. Esto es porque la variable numero2 fue creada dentro del bloque de llaves del IF

Sin embargo observar que la variable numero fue creada dentro del bloque de llaves del main() así que se puede usar tanto dentro como fuera del bloque IF. Lo que es importante es que esté dentro de su propio bloque de llaves (en este caso dentro de las llaves del método main())

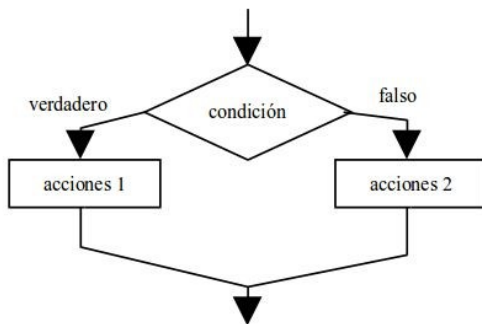
Ejemplo: El siguiente código calcula el área y el perímetro de un rectángulo usando un condicional simple

```
{
    Scanner cin;
    cin = new Scanner();
    double base, altura, area, perimetro;
    base = cin.nextInt();
    altura = cin.nextInt();
    if ((area > 0) && (altura > 0)) {
        area = base * altura;
        perimetro = 2 * base + 2 * altura;
        System.out.println("Area = " + area);
        System.out.println("Perimetro = " + perimetro);
    }
    if ((area <= 0) || (altura <= 0)) {
        System.out.println("Los datos son incorrectos");
    }
}
```


Observa que, en la primera instrucción condicional `if ((área > 0) && (altura > 0))` se comprueba que los dos datos sean positivos; en caso de serlo, se procede al cálculo del área y el perímetro mediante las acciones situadas entre `{` y `}`. Más abajo hay otra condicional `if ((área <= 0) || (altura <= 0))` para el caso de que alguno de los datos sea negativo o cero: en esta ocasión, se imprime en la pantalla un mensaje de error.

Condicional doble (if else)

La forma doble de la instrucción condicional es:



```
if (condicion) {  
    acciones si verdadero  
}  
else {  
    acciones si falso  
}
```

En esta forma, la instrucción funciona del siguiente modo: si el resultado de la condición es verdadero, entonces se ejecutan las acciones de la primera parte. Si es falso, se ejecutan las acciones de la parte "else".

Ejemplo: Podemos reescribir nuestro algoritmo del rectángulo usando una alternativa doble:

```
{
    Scanner cin;
    cin = new Scanner;
    double base, altura, area, perimetro;
    base = cin.nextInt();
    altura = cin.nextInt();
    if ((area > 0) && (altura > 0)) {
        area = base * altura;
        perimetro = 2 * base + 2 * altura;
        System.out.println("Area = " + area);
        System.out.println("Perimetro = " + perimetro);
    }
    else{
        System.out.println("Los datos son incorrectos");
    }
}
```

Lo más interesante de este algoritmo es compararlo con el anterior, ya que hace exactamente lo mismo. ¡Siempre hay varias maneras de resolver el mismo problema! Pero esta solución es un poco más sencilla, al ahorrarse la segunda condición, que va implícita en el else.

● **Práctica 20:** Hacer un programa que se le pase un número entero y que devuelva los siguientes mensajes:

- Si es impar: “El número: x es impar”
- Si es par: “El número x es par”

Donde x es el número introducido por el usuario

● **Práctica 21:** Hacer un programa que se le pase un número entero y determine si es múltiplo de 3 y de 5. Un modelo de mensajes a mostrar sería:

- x es múltiplo de 3 pero no es múltiplo de 5
- x no es múltiplo de 3 pero sí es múltiplo de 5
- x no es múltiplo de 3 ni de 5
- x es múltiplo de 3 y de 5

IF ELSE IF

Nota: Si sólo tenemos una única instrucción dentro de un bloque de llaves: “{“ “}” podemos eliminar las llaves

Ej.

```
If( num >= 0 ){  
    System.out.println(“num es un número positivo”);  
}  
else{  
    System.out.println(“num es un número negativo”);  
}
```

Equivale a:

```
If( num >= 0 )  
    System.out.println(“num es un número positivo”);  
else  
    System.out.println(“num es un número negativo”);
```

Utilizaremos para los ejemplos de if else if esa posibilidad de evitar las llaves para que nos quede más compacto el código.

Como hemos visto con una estructura if else cubrimos todo el espectro de posibilidades:

```
if (juanHablaCastellano == true)  
    System.out.println(“Juan sabe castellano”);  
else  
    System.out.println(“Juan no sabe castellano”);
```

Es fácil ver que para cualquier situación se mostraría un mensaje en pantalla. Si Juan sabe hablar castellano mostraría el primer mensaje, en otro caso el segundo

Ahora bien el apartado del else puede ser muchas veces una región muy amplia que puede ser importante hacer más preguntas en ese momento para hacer un mejor programa y más detallado. Pudiera

ser que Juan hubiera intentado alguna vez hablar castellano pero no continuó o que simplemente nunca tuvo interés en aprender. Esto nos genera estructuras anidadas en el else:

```
if (juanHablaCastellano == true)
    System.out.println("Juan sabe castellano");
else
    if( juanIntentoAprenderCastellano == false )
        System.out.println("Juan nunca intentó saber castellano");
    else
        System.out.println("Juan no sabe castellano pero lo intentó");
```

Como se ve es un if anidado en el else y como a su vez esta condición anidada tiene la parte del if y la parte del else SIEMPRE se le mostrará un mensaje en pantalla al usuario.

Y así podríamos seguir. Por ejemplo Juan intentó hablar castellano pero no lo consiguió porque su profesor era malo sería una condición que podríamos poner en el último else e ir así poniendo más detalladamente el histórico del aprendizaje de Juan

```
if (juanHablaCastellano == true)
    System.out.println("Juan sabe castellano");
else
    if( juanIntentoAprenderCastellano == false )
        System.out.println("Juan nunca intentó saber castellano");
    else
        if( juanNoAprendioPorUnMalProfesor == true )
            System.out.println("Juan no sabe castellano por mal profesor");
        else
            System.out.println("Juan intentó castellano con un profesor\n"
                                +"que no era malo, pero por algún motivo no lo logró");
```

Observar que el problema es que cada vez tenemos que indentar el código más adentro y puede dificultar la lectura. **SI PONEMOS SIEMPRE IF ELSE SIN SALTARNOS NINGÚN ELSE** está bien admitido escribir lo anterior de la siguiente forma:

```
if (juanHablaCastellano == true)
    System.out.println("Juan sabe castellano");
else if( juanIntentoAprenderCastellano == false )
    System.out.println("Juan nunca intentó saber castellano");
else if( juanNoAprendioPorUnMalProfesor == true )
    System.out.println("Juan no sabe castellano por mal profesor");
else
    System.out.println("Juan intentó castellano con un profesor\n"
        + "que no era malo, pero por algún motivo no lo logró");
    );
```

queda mucho más compacto y legible y sabemos que en todos los casos se le termina mostrando UN ÚNICO MENSAJE al usuario.

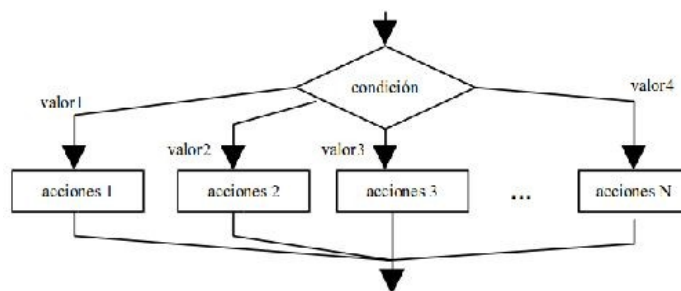
- **Práctica 22:** Hacer un programa que haga uso de estructuras if else if para que dado un número entero x muestre en pantalla:
- x es menor de 0
 - x es igual a 0
 - x es mayor que 0 y menor que 10
 - x es igual a 10
 - x es mayor que 10 y menor que 20
 - x es mayor o igual a 20

- **Práctica 23:** Hacer un programa que resuelva ecuaciones de segundo grado. El usuario introduce los valores de a,b,c de la fórmula: $aX^2+bX+c = 0$
- Se debe mostrar las dos posibles soluciones, salvo que haya una raíz negativa en cuyo caso se mostrará el mensaje: "no hay solución real"
- Nota. Para resolver podemos hacer uso de: `Math.sqrt()` para obtener la raíz cuadrada

Condicional múltiple

En algunas ocasiones nos encontraremos con selecciones en las que hay más de dos alternativas (es decir, en las que no basta con los valores "verdadero" y "falso"). Siempre es posible plasmar estas selecciones complejas usando varias estructuras if-else if-else if... anidadas, es decir, unas dentro de otras, justo como hemos descrito antes. Pero, tenemos otra opción para escribirlas sin tantas líneas de código

La estructura condicional múltiple sirve, por tanto, para simplificar estos casos de condiciones con muchas alternativas. Su sintaxis general es:



```
switch (expresión) {  
  case valor1: acciones-1;  
    break;  
  case valor2: acciones-2;  
    break;  
  case valor3: acciones-3;  
    break;  
  ...  
  default: acciones-N;  
}
```

Su funcionamiento es el siguiente:

1- se evalúa expresión, que en esta ocasión no tiene que ser de tipo lógico, sino **char, byte, short o int** (Lo más habitual es que sea de tipo entero).

ej.

```
switch(dia){ //dia en este caso es un número entero de 1 a 7 por día de la semana
```

2- El resultado de expresión se compara con cada uno de los valores valor1, valor2... valorN:

ej.

```
case 1: //si accedemos a este caso es que la variable dia es 1
```

2.1- Si coincide con alguno de ellas, se ejecutan únicamente las acciones situadas a la derecha del valor coincidente (acciones-1, acciones-2... acciones-N).

ej.

```
case 1: System.out.println("Lunes"); //Si dia es 1 se mostrará en pantalla Lunes
```

2.2- Si se diera el caso de que ningún valor fuera coincidente, entonces se ejecutan las acciones-default ubicadas al final de la estructura. Esta última parte de la estructura no es obligatorio que aparezca.

ej.

```
default: System.out.println("Número incorrecto");
```

Ejemplo:

Construyamos un algoritmo que escriba los nombres de los días de la semana en función del valor de una variable entera llamada "día". Su valor se introducirá por teclado. Los valores posibles de la variable "día" serán del 1 al 7: cualquier otro valor debe producir un error.

```
{
    Scanner cin = new Scanner();
    int dia;
    dia = cin.nextInt();
    switch(dia) {
        case 1: System.out.println("Lunes");
                break;
        case 2: System.out.println("Martes");
                break;
        case 3: System.out.println("Miércoles");
                break;
        case 4: System.out.println("Jueves");
                break;
        case 5: System.out.println("Viernes");
                break;
        case 6: System.out.println("Sábado");
                break;
        case 7: System.out.println("Domingo");
                break;
        default: System.out.println("Número incorrecto");
    }
}
```

En este programa, la variable día, una vez leída, se compara con los siete valores posibles. Si vale 1, se realizará la acción `System.out.println("lunes");`; si vale 2, se realiza `System.out.println("martes");`; y así sucesivamente. Por último, si no coincide con ninguno de los siete valores, se ejecuta la parte default.

● **Práctica 24:** Hacer un programa que le pregunte al usuario su idioma preferido. Donde la letra 'c' será castellano, 'i' inglés, 'f' francés. Según la opción que haya elegido se le mostrará respectivamente: "Buenos días", "Good morning", "Bonjour" y termina. Si el usuario escribe cualquier otra cosa el programa mostrara: "no entiendo tu idioma" y termina. Tener en cuenta que Scanner no tiene un método `nextChar()` en su defecto usar: `next().charAt(0)` que va a tomar el primer carácter de la línea de texto introducida. Ej.

```
Scanner cin = new Scanner(System.in);
char c = cin.next().charAt(0);
```

Observemos ahora con detenimiento la orden **break** que hemos introducido. **break es una opción para "romper" el flujo del programa que estamos teniendo. Si está dentro de un bucle finaliza el bucle. Si está dentro de un switch no continúa ejecutando las instrucciones del switch**

Observemos el caso de las siguientes calificaciones posibles:

insuficiente (de 0 a 4 ptos), suficiente (5ptos), bien (6ptos), notable (7,8ptos), sobresaliente(9,10ptos) Ejemplo de ejecución:

```
{
    Scanner cin = new Scanner(System.in);
    System.out.println("Introducir nota numérica");
    int nota = cin.nextInt();
    switch(nota){
        case 0: System.out.println("Insuficiente");
                break;
        case 1: System.out.println("Insuficiente");
                break;
        case 2: System.out.println("Insuficiente");
                break;
        case 3: System.out.println("Insuficiente");
                break;
        case 4: System.out.println("Insuficiente");
                break;
        case 5: System.out.println("Suficiente");
                break;
        case 6: System.out.println("Bien");
                break;
        case 7: System.out.println("Notable");
                break;
        case 8: System.out.println("Notable");
                break;
        case 9: System.out.println("Sobresaliente");
                break;
        case 10: System.out.println("Sobresaliente");
                break;
        default: System.out.println("no entiendo esa nota");
    }
}
```

Podemos observar que los casos: 0,1,2,3,4 van a realizar exactamente lo mismo, los casos 7,8 tienen también un mismo comportamiento así como 9,10.

Podemos aprovechar esa circunstancia para ahorrarnos varias líneas de código:

```
{
    Scanner cin = new Scanner(System.in);
    System.out.println("Introducir nota numérica");
    int nota = cin.nextInt();
    switch(nota){
        case 0:
        case 1:
        case 2:
        case 3:
        case 4: System.out.println("Insuficiente");
                break;
        case 5: System.out.println("Suficiente");
                break;
        case 6: System.out.println("Bien");
                break;
        case 7:
        case 8: System.out.println("Notable");
                break;
        case 9:
        case 10: System.out.println("Sobresaliente");
                break;
        default: System.out.println("no entiendo esa nota");
    }
}
```

Al quitar las líneas break que teníamos antes el programa sigue ejecutando las sentencias que tiene debajo hasta que se encuentra un System.out.println() y finalmente el break que detiene la ejecución del switch.

Nota: El uso de break es necesario dentro de la estructura switch() pero suele estar bastante mal visto en cualquier otra estructura (if, for,...) ya que rompe la estructura del programa. Debe intentarse restringir su uso a los switch()

● **Práctica 25:** Hacer un programa que le pida al usuario que introduzca una letra y por medio de una estructura switch, evitando el mayor número de sentencias break posibles , muestre en pantalla el mensaje: “es una vocal” cuando el usuario haya introducido una vocal (ya sea minúscula o mayúscula) y el mensaje: “no es una vocal” si no lo fuera.

● **Práctica 26:** Escribe un programa que pida por teclado un día de la semana (ún número entero del 1 al 5 que representa de lunes a viernes) y que diga qué asignatura toca a primera hora ese día.

Estructuras repetitivas (bucles)

Los ordenadores se diseñaron inicialmente para realizar tareas sencillas y repetitivas. La estructura repetitiva, por tanto, reside en la naturaleza misma de los ordenadores y consiste, simplemente, en repetir varias veces un conjunto de instrucciones. Las estructuras repetitivas también se llaman bucles, lazos, iteraciones o *loops*. Nosotros preferiremos la denominación "bucle".

Los bucles tienen que repetir un conjunto de instrucciones un número finito de veces. Si no, nos encontraremos con un bucle infinito y el algoritmo no funcionará. En rigor, ni siquiera será un algoritmo, ya que no cumplirá la condición de finitud.

El bucle infinito es un peligro que acecha constantemente a los programadores y nos toparemos con él muchas veces en el ejercicio de nuestra actividad. Para conseguir que el bucle se repita sólo un número finito de veces, tiene que existir una condición de salida del mismo, es decir, una situación en la que ya no sea necesario seguir repitiendo las instrucciones.

Por tanto, los bucles se componen, básicamente, de dos elementos:

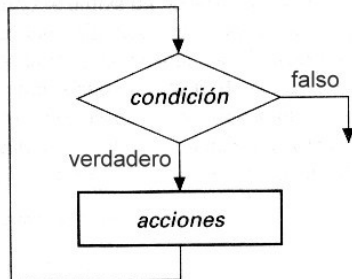
- 1) un cuerpo del bucle o conjunto de instrucciones que se ejecutan repetidamente
- 2) una condición de salida para dejar de repetir las instrucciones y continuar con el resto del algoritmo

Dependiendo de dónde se coloque la condición de salida (al principio o al final del conjunto de instrucciones repetidas), y de la forma de realizarla, existen tres tipos de bucles. Los tres tipos de bucle se denominan:

- 1) Bucle "**mientras... hacer**": la condición de salida está al principio del bucle (bucle while).
- 2) Bucle "**hacer... mientras**": la condición de salida está al final del bucle. (bucle do while)
- 3) Bucle "**para**": la condición de salida está al principio y se realiza con un contador automático. (bucle for)

Bucle Mientras... hacer (bucle while)

El bucle "mientras... hacer" o, simplemente, "mientras", es una estructura que se repite mientras una condición sea verdadera. La condición, en forma de expresión lógica, se escribe en la cabecera del bucle, y a continuación aparecen las acciones que se repiten (cuerpo del bucle):



```
while (condición) {  
  acciones (cuerpo del bucle)  
}
```

Cuando se llega a una instrucción mientras, se evalúa la condición. Si es verdadera, se realizan las acciones y, al terminar el bloque de acciones, se regresa a la instrucción mientras (he aquí el bucle o lazo). Se vuelve a evaluar la condición y, si sigue siendo verdadera, vuelve a repetirse el bloque de acciones. Y así, sin parar, hasta que la condición se haga falsa.

Ejemplo: Escribir un algoritmo que muestre en la pantalla todos los números enteros entre 1 y 100.

```
{  
  int i = 0;  
  while (i < 100) {  
    i = i + 1;  
    System.out.println(i);  
  }  
}
```

Aquí observamos el uso de un contador en la condición de salida de un bucle, un elemento muy común en estas estructuras. Observa la evolución del algoritmo:

- `int i = 0;` Se declara y se le asigna el valor 0 a la variable `i` (contador)
- `while (i <= 100).` Condición de salida del bucle. Es verdadera porque `cont` vale 0, y por lo tanto es menor o igual que 100.
- `i = i + 1;` Se incrementa el valor de `i` en una unidad. Como valía 0, ahora vale 1.
- `println(i);` Se escribe por pantalla el valor de `i`, que será 1.

Después, el flujo del programa regresa a la instrucción mientras, ya que estamos en un bucle, y se vuelve a evaluar la condición. Ahora `i` vale 1, luego sigue siendo verdadera. Se repiten las instrucciones del bucle, y `cont` se incrementa de nuevo, pasando a valer 2. Luego valdrá 3, luego 4, y así sucesivamente.

La condición de salida del bucle hace que éste se repita mientras cont valga menos de 100. De este modo nos aseguramos de escribir todos los números hasta el 100.

Lo más problemático a la hora de diseñar un bucle es, por lo tanto, pensar bien su condición de salida, porque si la condición de salida nunca se hiciera falsa, caeríamos en un bucle infinito. Por lo tanto, la variable implicada en la condición de salida debe sufrir alguna modificación en el interior del bucle; si no, la condición siempre sería verdadera. En nuestro ejemplo, la variable cont se modifica en el interior del bucle: por eso llega un momento, después de 100 repeticiones, en el que la condición se hace falsa y el bucle termina.

Cuando tenemos una única instrucción en el cuerpo del bucle while podemos evitar escribir las llaves: “{ “}” para delimitar el cuerpo del while:

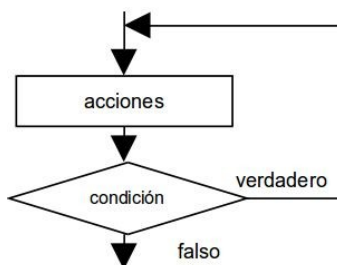
● **Práctica 27:** Hacer un programa que muestre la tabla de multiplicar del número 5 usando un bucle while

● **Práctica 28** Hacer un programa que el usuario vaya introduciendo números enteros. El programa finaliza cuando el usuario introduce el número 0. En ese momento se le muestra la suma total de los números positivos y la suma total de los números negativos

Bucle Hacer... mientras (Bucle do while)

El bucle de tipo "Hacer... mientras" es muy similar al bucle "mientras", con la salvedad de que la condición de salida se evalúa al final del bucle, y no al principio, como a continuación veremos. Todo bucle "Hacer... mientras" puede escribirse como un bucle "mientras", pero al revés no siempre sucede.

La forma de la estructura "hacer... mientras" es la que sigue:



```
do {  
    acciones (cuerpo del bucle)  
}while (condicion);
```

Cuando el ordenador encuentra un bucle de este tipo, ejecuta las acciones escritas entre { y } y, después, evalúa la condición, que debe ser de tipo lógico. Si el resultado es falso, se vuelven a repetir las

acciones. Si el resultado es verdadero, el bucle se repite. Si es falso, se sale del bucle y se continúa ejecutando la siguiente instrucción.

Existe, pues, una diferencia fundamental con respecto al bucle "mientras": la condición se evalúa al final. Por lo tanto, **las acciones del cuerpo de un bucle "hacer... mientras" se ejecutan al menos una vez**, cuando en un bucle "mientras" es posible que no se ejecuten ninguna (si la condición de salida es falsa desde el principio)

Ejemplo: Diseñar un algoritmo que escriba todos los números enteros entre 1 y 100, pero esta vez utilizando un bucle "hacer... mientras" en lugar de un bucle "mientras"

```
{  
    int i = 0;  
    do {  
        i = i + 1;  
        System.out.println(i);  
    }  
    while (cont < 100);  
}
```

Observa que el algoritmo es básicamente el mismo que en el ejemplo anterior, pero hemos cambiado el lugar de la condición de salida.

● **Práctica 29:** Hacer un programa que muestre la tabla de multiplicar del número 5 usando un bucle do while

● **Práctica 30:** Hacer un programa que vaya mostrando un número aleatorio mayor o igual a 10 y menor a 20. Una vez mostrado el número se le pide al usuario si quiere o no continuar. Si el programa introduce una "c" o una "C" vuelve a mostrar un nuevo número aleatorio. El programa finaliza cuando usuario introduzca algo diferente. El bucle se deberá realizar mediante do while

Para el número aleatorio hay varias opciones:

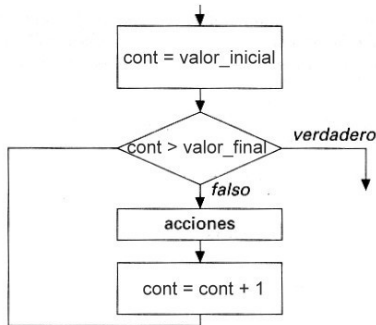
(int)(Math.random()*10);

Math.random() devuelve un double mayor o igual a 0 y menor que 1 si multiplicas por 10 tienes entonces mayor igual a 0 y menor que 10. Si hacemos cast a int: (int) truncamos el número y tenemos un número entero de 0 a 9 (ambos inclusive)

Bucle "para" (Bucle for)

En muchas ocasiones se conoce de antemano el número de veces que se desean ejecutar las acciones del cuerpo del bucle. Cuando el número de repeticiones es fijo, lo más cómodo es usar un bucle "para", aunque sería perfectamente posible sustituirlo por uno "mientras".

La estructura "para" repite las acciones del bucle un número prefijado de veces e incrementa automáticamente una variable contador en cada repetición. Su forma general es:



```
for (inicialización; condición; incremento) {  
  acciones  
}
```

La inicialización consiste en la asignación del valor inicial a una variable contador (por ejemplo, `i`). La primera vez que se ejecutan las acciones del cuerpo del bucle, la variable `cont` tendrá el valor especificado en la inicialización. En la siguiente repetición, la variable contador se incrementará según lo expresado en la sección incremento (por ejemplo, `i = i + 1`, o bien `i++`), y así sucesivamente. El bucle se repetirá mientras que se cumpla la condición.

Ej. Diseñar un algoritmo que escriba todos los números enteros entre 1 y 100, utilizando un bucle "para"

```
{  
  int i;  
  for (i = 1; i <= 100; i = i + 1) {  
    System.out.println(i);  
  }  
}
```

De nuevo, lo más interesante es observar las diferencias de este algoritmo con los dos ejemplos anteriores. Advierte que ahora no es necesario asignar un valor inicial a la variable `i` antes de entrar al bucle, ya que se hace en la misma declaración del bucle; y tampoco es necesario incrementar el valor de `i` en el cuerpo del bucle (`i = i + 1`), ya que de eso se encarga el propio bucle "para". Por último, la condición de repetición (`i <= 100`) se expresa también en la declaración del bucle.

Ej. Escribir todos los números enteros impares entre 1 y 100, utilizando un bucle "para"

```
{  
  int i;  
  for (i = 1; i <= 100; i = i + 2) {
```

```
System.out.println(i);  
}
```

```
}
```

Este ejemplo, similar al anterior, sirve para ilustrar la gran flexibilidad del bucle "para" cuando se conocen bien los límites iniciales y finales del bucle. La variable cont se incrementará en 2 unidades en cada repetición del bucle, por lo que tomará los valores 1, 3, 5, 7, y así sucesivamente hasta 99.

- **Práctica 31:** Hacer un programa que muestre la tabla de multiplicar del número 5 usando un bucle for

- **Práctica 32:** Hacer un programa que reciba dos números enteros positivos del usuario y muestre la suma de todos los números que hay entre esos dos números.
Ej. Usuario introduce: 10, 5 entonces la secuencia de números es:
5,6,7,8,9,10
y la suma de esos números es:
45
Hacer uso de un bucle for para este programa

- **Práctica 33:** Se pretende imitar el comportamiento de un sistema de control por pin. El usuario tiene 3 intentos para acertar con el pin, cada vez de esos tres intentos que falle se le informa y se le dice el número de intentos que le queda. Si acierta se le muestra un mensaje que diga: "El código es correcto. Bienvenido" Si no acierta en los tres intentos el programa termina
Hacer 3 versiones de este código una con while, otra con do while y finalmente for

Formas peculiares de bucle

- Bucles de una instrucción

Cuando el cuerpo del bucle es de una única línea podemos evitar utilizar las llaves: “{“ “}” para delimitar el cuerpo del bucle

Ej

```
{  
  int i;  
  for (i = 1; i <= 100; i = i + 1) {  
    System.out.println(i);  
  }  
  System.out.println("Esta línea está fuera del bucle");  
}
```

equivale a:

```
{  
  int i;  
  for (i = 1; i <= 100; i = i + 1)  
    System.out.println(i);  
  System.out.println("Esta línea está fuera del bucle");  
}
```

- Bucle for infinito:

```
for(;;){  
    //acciones que se repiten para siempre  
}
```

Para entender el anterior bucle infinito veamos,

El bucle for en profundidad

El bucle for se compone de tres partes separadas por punto y coma “;”

Veámoslo:

for(declaración-inicialización; condicion de repetición; otras instrucciones)

y ninguna de ellas es obligatoria.

Así que la siguiente expresión provoca un bucle infinito:

```
Scanner cin = new Scanner(System.in);  
for(;;){  
    System.out.println(cin.nextInt());  
}
```

la parte de declaración inicialización puede admitir varias variables:

for(int i=0, j=0, k=0; condición de repetición; otras instrucciones)

y la parte de otras instrucciones admite varias instrucciones separadas por comas:

for(inicialización; condición de repetición; a=a+1, b=a+2)

Así pues se pueden conseguir cosas como:

- Bucle for con dos contadores:

```
for(int i=0, j=0; i < 10; i++, j += 2){  
    System.out.println("i= " + i + " j= "+j);  
}
```

- Bucle for sin contadores

Este ejemplo acepta entradas por teclado y se detiene cuando se introduce 100

```
Scanner cin = new Scanner(System.in);  
for(int teclado=0;teclado != 100;teclado=cin.nextInt()) {  
    System.out.print("has introducido: " + teclado);  
    System.out.println("Intro un número: ");  
}
```

Contadores, acumuladores

Asociadas a los bucles se encuentran a menudo algunas variables auxiliares. Como siempre se utilizan de la misma manera, las llamamos con un nombre propio (contador, acumulador, etc.), pero hay que dejar claro que no son más que variables comunes, aunque se usan de un modo especial.

Un contador es una variable cuyo valor se incrementa o decrementa en cada repetición de un bucle. Es habitual llamar a esta variable "i" (de índice). En caso de tener que usar varios contadores los siguientes nombres serían "j", "k",...

Si observamos como usamos arriba la variable i vemos que hemos declarado la variable antes de entrar en el bucle. Eso significa que el ámbito de esta variable es mayor que el bucle para el que lo estamos usando. Observar Este ejemplo:

```
{
    int i; //Estamos definiendo i antes del bucle
    for (i = 1; i <= 100; i = i + 1) {
        System.out.println(i);
    }
    System.out.println("El valor de i después del bucle es: " + i);
}
```

La variable i existe y mantiene valor después de la ejecución del bucle. Eso en alguna ocasión puede ser interesante, pero la mayoría de las veces sólo necesitamos esa variable durante la ejecución del bucle. Ej.

```
{
    for (int i = 1; i <= 100; i = i + 1){//la variable i tiene ahora por ámbito el bucle
        System.out.println(i);
    }
    System.out.println("El valor de i después del bucle es: " + i);
}
```

El ejemplo que acabamos de poner da error porque estamos usando la variable i fuera del ámbito en el que ha sido creado. Para que el código compile tendremos que eliminar el último System.out.println:

```
{
    for (int i = 1; i <= 100; i = i + 1) { //la variable i tiene ahora por ámbito el
bucle
        System.out.println(i);
    }
}
```

En general, veremos casi siempre códigos que declaran la variable `i` en el bucle `for`. También la forma habitual de escribir la orden de incremento es mediante `++`

Ej

```
for (int i = 1; i <= 100; i++ ) {  
    System.out.println(i);  
}
```

Otra forma típica del contador es:

```
i--;
```

Por lo que ya sabemos esa instrucción equivale a:

```
i = i - 1;
```

Ej. Mostrar los números del 1 hasta el 10 de forma invertida:

```
for(int i=10; i > 0; i--){  
    System.out.println(i);  
}
```

En este caso, la variable se decrementa en una unidad; si `i` valiera 5 antes de la instrucción, tendremos que `i` valdrá 4 después de su ejecución.

El incremento o decremento no tiene por qué ser de una unidad. La cantidad que haya que incrementar o decrementar vendrá dada por la naturaleza del problema.

Ej.

```
i += 3;
```

Como ya sabemos la anterior línea equivale a:

```
i = i + 3;
```

Así que de esta forma el contador irá dando pasos de 3 en 3. Si `i` valiera 5 después de la ejecución valdría 8;

Ej. Mostrar la siguiente secuencia de números: 6, 9, 12, 15,..., 27, 30

```
for( int i=6; i<=30; i +=3 )  
    System.out.println(i);
```

● **Práctica 34:** Hacer un programa que muestre la secuencia de números: 71, 65, 59,..., y que pare cuando ya sean negativos

● **Práctica 35:** Sabiendo que:

```
char letra = 'a';  
letra++;  
System.out.println(letra);
```

nos muestra la letra 'b' tenemos una forma para ir mostrando las diferentes letras incrementando una variable. Usar un bucle for y mostrar en pantalla la secuencia de las 10 primeras letras del abecedario (todas en minúscula)

Acumuladores

Las variables acumuladoras tienen la misión de almacenar resultados sucesivos, es decir, de acumular resultados, de ahí su nombre.

Imaginemos que el usuario va introduciendo dinero. Es interesante mostrarle la cantidad acumulada que ha ingresado.

Para eso usaremos el acumulador.

Ej
{

```
Scanner cin = new Scanner(System.in);  
total = 0;  
for(int i= 0; i<4; i++){  
    System.out.println("Introduzca dinero: ");  
    int dinero = cin.nextInt();  
    total = total + dinero;    //aquí tenemos nuestro acumulador  
}
```

}

En la variable total vamos “acumulando” el dinero que introduce el usuario en cada una de las cuatro iteraciones

● **Práctica 36:** Hacer un programa que se emule un juego de lanzar 3 dados. En cada iteración se muestra el resultado sacado en cada uno de los dados y cuánto es el acumulado de esa tirada de 3 dados. Cuando el usuario pulse en “f” o “F” el programa finaliza y

muestra el acumulado de puntos de sumar todas las tiradas y el número de tiradas
Nota: se puede utilizar Random para el número aleatorio:

```
Random rnd = new Random();  
rnd.nextInt(6); //genera un aleatorio desde 0 hasta 5
```

- **Práctica 37:** Crear un programa que le pregunte al usuario la cantidad de billetes que tiene de 500, luego le pregunte por la cantidad de billetes que tiene de 200, después pregunte por los de 100, 50, 20, 10, 5. Finalmente se le dirá al usuario la cantidad de dinero acumulado que tiene en billetes.

Conmutador

Un conmutador (o interruptor) es una variable que sólo puede tomar dos valores. Pueden ser, por tanto, de tipo booleano, aunque también pueden usarse variables enteras o de tipo carácter.

La variable conmutador recibirá uno de los dos valores posibles antes de entrar en el bucle. Dentro del cuerpo del bucle, debe cambiarse ese valor bajo ciertas condiciones. Utilizando el conmutador en la condición de salida del bucle, puede controlarse el número de repeticiones.

Ej Escribir un programa que sume todos los números positivos introducidos por el usuario a través del teclado. Para terminar de introducir números, el usuario tecleará un número negativo.

```
{
    int num;
    int suma=0;
    boolean terminar=false;
    Scanner cin = new Scanner(System.in);

    while (terminar == false) {
        System.out.println("Introduce un número (negativo para terminar)");
        num = cin.nextInt();
        if (num >= 0) {
            suma = suma + num;
        }
        else {
            terminar = true;
        }
    }
    System.out.println(suma);
}
```

Este código es una variación del ejemplo con acumuladores que vimos más atrás. Entonces el usuario introducía 10 números, y ahora puede ir introduciendo números indefinidamente, hasta que se canse. ¿Cómo indica al ordenador que ha terminado de introducir números? Simplemente, tecleando un número negativo.

El bucle se controla por medio de la variable "terminar": es el conmutador. Sólo puede tomar dos valores: "verdadero", cuando el bucle debe terminar, y "falso", cuando el bucle debe repetirse una vez más. Por lo tanto, "terminar" valdrá "falso" al principio, y sólo cambiará a "verdadero" cuando el usuario introduzca un número negativo.

● **Práctica 38:** Hacer un programa que recibe un número de horas, por ejemplo 135.25 y lo convierte a un formato de: días, horas, minutos, segundos. Siguiendo el ejemplo anterior: 5días 15horas 15minutos 0segundos

● **Práctica 39:** Hacer un programa que recibe un número entero mayor o igual a 10 y menor o igual a 99 (si el usuario introduce un número no válido se le pedirá que repita hasta que lo haga bien) , y se le muestre el número con las cifras invertidas. Por ej. si 34 se mostraría en pantalla 43

● **Práctica 40:** Escribir un programa que reciba un número entero mayor o igual a 0 y menor o igual a 999 y muestre cuántas cifras tiene

● **Práctica 41:** Escribir un programa que el usuario escriba una cadena de texto y se le muestre el texto al revés. Ej “cabello” mostraría: “ollebac”
Nota: “cabello”.substring(2,3) devuelve: “b” que es la tercera letra de ese texto.

● **Práctica 42:** Hay una cuadrilla de trabajadores que aceptar encargos. Pongamos por ejemplo que tienen un encargo por 10000€. Al finalizar el trabajo observan que han participado 5 componentes de la cuadrilla dedicando cada uno de ellos respectivamente el siguiente número de horas:
10, 100, 100, 20, 20
Así que en total han invertido 250horas para un trabajo de 10000€ saliendo la hora trabajada a: 40€
Hacer un programa que les permita calcular lo que sale cada hora trabajada. Como datos de entrada recibirá el dinero total que se pagará por el encargo. El número de participantes de la cuadrilla y el número de horas que haya realizado cada uno de los participantes

● **Práctica 43:** Escribir un programa donde se genera un número secreto mayor o igual a 1 y menor o igual a 99 El jugador intentará acertar el número si se equivoca el programa le informa de si el número secreto es mayor o menor que el que él ha introducido. El programa finaliza cuando el jugador acierte en cuyo caso se le mostrará el número de intentos que ha necesitado para acertar.
Nota: se puede utilizar Random para el número aleatorio:
Random rnd = new Random();
rnd.nextInt(30); //genera un aleatorio desde 0 hasta 29

● **Práctica 44:** Programa generador de potencias. El usuario introduce un número entero de 1 a 20. Se le mostrará en pantalla las 5 primeras potencias de ese número.
Ej. número introducido 2
 $2^1 = 2$
 $2^2 = 4$
 $2^3 = 8$
 $2^4 = 16$
 $2^5 = 32$

● **Práctica 45:** Crear un programa que reciba un número entero y que muestre su descomposición en números primos:
 $300 = 2 * 2 * 3 * 5 * 5$

● **Práctica 46:** Convertir un número a binario. El usuario introduce un número entero decimal y el programa muestra en pantalla como es su forma en binario

Instrucciones de salto

Las instrucciones de salto las consideraremos en general prohibidas.

¿Que por qué existen entonces? Porque a veces, MUY POCAS VECES, son necesarias.

break

La sentencia break sale abruptamente del bloque actual. No importa si estás en un if, en un switch o en un for. Sales afuera, rompiendo con ello la estructura del programa.

El único uso racional de esta instrucción es dentro de un switch. Cada case del switch debe terminar con un break o, de lo contrario, el switch se ejecutará desde el case actual hasta el final.

continue

La sentencia continue fuerza la finalización prematura de la iteración de un bucle. Si te ves en la obligación de usarla, es que el bucle está mal planteado.

Ej

```
int i=0;
while( i < 5){
    i++;
    if( i < 0 || i > 4)
        continue; //no sigue mirando en el cuerpo del bucle.Regresa al while
    else
        System.out.println("i está entre 0 y 4: " + i);
}
```

return

Teóricamente, return debería ser la última instrucción de un método que devuelve algún valor. LA ÚLTIMA. Aunque funcione no es una buena idea ponerlo en mitad del código, ni poner múltiples returns en cada una de las ramas de un if. Cada algoritmo debería tener un único punto de entrada y un único punto de salida. Esto mantiene la estructura del código, permite detectar más fácilmente los errores con el debugger y hace más fácil realizar las pruebas de software.

Sin embargo si dentro del algoritmo ocurre algo tan problemático, que resulte imposible que continúe su ejecución con normalidad, sería procedente `return` y terminar prematuramente.

En tal caso, se aceptará un `return` a destiempo para devolver el control al objeto que llamó al método que ha fallado, siempre que el diseño normal (es decir, con un único `return` al final del método) sea mucho más complejo que la solución del `return` excepcional.

Vamos a ver con más detalle el uso de `return` con los métodos.