SIMULACIONES

IV + IoT + DL

Industria 5.0

Diego L. Aristizábal Ramírez
Profesor, Facultad de Ciencias, Departamento de Física
Universidad Nacional de Colombia
Medellín

ANEXO 0

ABCS DE JAVA

Temas
✓ INTRODUCCIÓN
✓ ¿QUÉ ES JAVA?
✓ ELEMENTOS DEL LENGUAJE JAVA
✓ OPERADORES ARITMÉTICOS
✓ OPERADORES RELACIONALES
✓ ARREGLOS
✓ VECTORES
✓ EL FLUJO DE UN PROGRAMA
✓ MANEJO DE EXCEPCIONES
✓ REFERENCIAS

INTRODUCCIÓN

Se debe advertir que este no es un curso de programación, aunque ésta obviamente es usada para lograr los objetivos de este (simulaciones e instrumentación virtual). En principio se asume que los estudiantes ya saben programar en algún lenguaje (Basic, Pascal, C, C++, Java, Phyton…) y con cualquier paradigma de programación (programación estructurada, programación orientada a objetos,…).

Como se afirmó en el módulo # 1, el curso usa como paradigma de programación la OOP, como lenguaje de programación Java, como sistema operativo ANDROID y las aplicaciones se orientan para dispositivos móviles (tabletas y celulares). Es por esto que antes de comenzar a desarrollar aplicaciones android en plena forma, se

abordará hasta el módulo 7 lo más básico del lenguaje de programación Java y de la respectiva OOP.

Un estudiante que realice este curso de forma responsable, además de quedar capacitado para realizar buenas simulaciones en ciencias y tecnología, adquirirá también las competencias necesarias para hacer software de muy buen nivel que involucre instrumentación electrónica, por ejemplo, en áreas como: domótica, comunicaciones, robótica, etc.

¿QUÉ ES JAVA?

El lenguaje Java se parece al lenguaje C++ de modo que un programador que conozca este lenguaje ha dado un gran paso adelante.

Sin embargo, existen también grandes diferencias entre ambos lenguajes. Un programador puede haber usado el lenguaje C++ como un lenguaje C mejorado sin haber usado para nada la Programación Orientada a Objetos. Sin embargo, Java es un lenguaje plenamente orientado a objetos, y para escribir el programa más simple debe definirse una clase. Los tipos básicos de datos son similares, pero los arrays (arreglos) son distintos, y las cadenas de caracteres en Java son objetos de la clase String.

El lenguaje Java es a la vez compilado e interpretado, Figura 1. Con el compilador se convierte el código fuente que reside en archivos cuya extensión es **.java**, a un conjunto de instrucciones que recibe el nombre de *bytecodes* que se guardan en un archivo cuya extensión es **.class**. Estas instrucciones son independientes del tipo de computador. El intérprete ejecuta cada una de estas instrucciones en un computador específico (Windows, Macintosh, etc.). Solamente es necesario, por tanto,

compilar una vez el programa, pero se interpreta cada vez que se ejecuta en un computador.

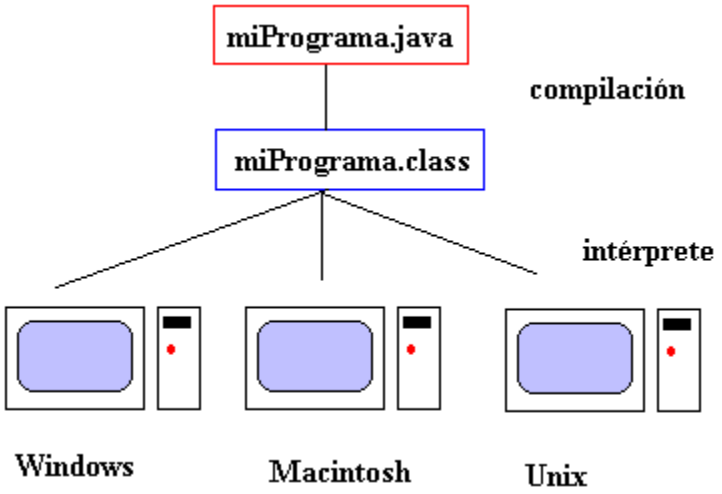

Figura 1

Cada intérprete Java es una implementación de la Máquina Virtual Java (JVM). Los *bytecodes* posibilitan el objetivo de "write once, run anywhere" (escribir el programa una vez y que se pueda correr en cualquier plataforma que disponga de una implementación de la JVM). Por ejemplo, el mismo programa Java puede correr en Windows, Solaris, Macintosh, etc. Java es un lenguaje multiplataforma.

Java es, por tanto, algo más que un lenguaje, ya que la palabra Java se refiere a dos cosas inseparables: el lenguaje que nos sirve para crear programas y la Máquina Virtual Java que sirve para ejecutarlos. Como se observa en la Figura 2, el API de Java y la Máquina Virtual Java forman una capa intermedia (Java platform) que aisla el programa Java de las especificidades del hardware (hardware-based platform).

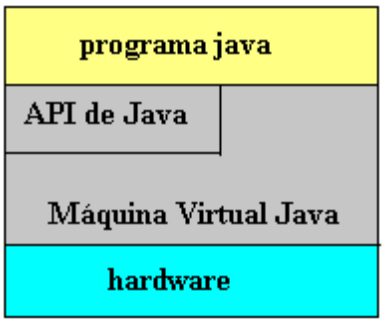

Figura 2

La Máquina Virtual Java (JVM) es el entorno en el que se ejecutan los programas Java, su misión principal es la de garantizar la portabilidad de las aplicaciones Java. Define esencialmente un computador abstracto y especifica las instrucciones (*bytecodes*) que este computador puede ejecutar. El intérprete Java específico ejecuta las instrucciones que se guardan en los archivos cuya extensión es **.class**. Las tareas principales de la JVM son las siguientes:

- ✓ Reservar espacio en memoria para los objetos creados.
- ✓ Liberar la memoria no usada (garbage collection).
- ✓ Asignar variables a registros y pilas.
- ✓ Llamar al sistema huésped para ciertas funciones, como los accesos a los dispositivos.
- ✓ Vigilar el cumplimiento de las normas de seguridad de las aplicaciones Java.

Los tipos de programas más comunes que se pueden hacer con Java son los *applets* (se ejecutan en el navegador de la máquina cliente) y las *aplicaciones* (programas que se ejecutan directamente en la JVM). Otro tipo especial de programa se denomina *servlet* que es similar a los *applets* pero se ejecutan en los servidores Java.

La API (*Aplication Programing Interface*) de Java es muy rica, está formada por un conjunto de paquetes de clases que le proporcionan una gran funcionalidad. Para acceder a la **documentación de la API** en su versión 19.0 de java visitar,

https://docs.oracle.com/en/java/javase/19/docs/api/index.html

Java proporciona también extensiones, por ejemplo, define un API para 3D, para los servidores, telefonía, reconocimiento de voz, etc.

En este anexo se abordará los siguientes temas del lenguaje Java:

- ✓ ELEMENTOS DEL LENGUAJE JAVA
- ✓ OPERADORES ARITMÉTICOS
- ✓ OPERADORES RELACIONALES
- ✓ ARREGLOS
- ✓ VECTORES
- ✓ EL FLUJO DE UN PROGRAMA
- ✓ MANEJO DE EXCEPCIONES

ELEMENTOS DEL LENGUAJE JAVA

La sintaxis de un lenguaje define los elementos de dicho lenguaje y cómo se combinan para formar un programa. Los elementos típicos de cualquier lenguaje son los siguientes:

- ✓ Identificadores: los nombres que se dan a las variables.
- ✓ Tipos de datos.
- ✓ Palabras reservadas: las palabras que utiliza el propio lenguaje.

- ✓ Sentencias.
- ✓ Bloques de código.
- ✓ Comentarios.
- ✓ Expresiones.
- ✓ Operadores

Identificadores

Un identificador es un nombre que identifica a una variable, a una constante, a un método, a una clase a un paquete de un programa. Todos los lenguajes tienen ciertas reglas para componer los identificadores:

- ✓ Todos los identificadores han de comenzar con una letra, el carácter subrayado (_) o el carácter dólar ($).
- ✓ Puede incluir, pero no comenzar por un número.
- ✓ No puede incluir el carácter espacio en blanco.
- ✓ Distingue entre letras mayúsculas y minúsculas.
- ✓ No se pueden utilizar las palabras reservadas como identificadores.

Además de estas restricciones, hay ciertas convenciones que hacen que el programa sea más legible, pero que no afectan a la ejecución del programa. La primera y fundamental es la de encontrar un nombre que sea significativo, de modo que el programa sea lo más legible posible. El tiempo que se pretende ahorrar eligiendo nombres cortos y poco significativos se pierde con creces cuando se revisa el programa después de cierto tiempo.

Tipo de identificador	Convención	Ejemplo
Nombre de una clase	Comienza con letra mayúscula	String, Rectangulo, Persona, Polea.
Nombre de método	Comienza con letra minúscula	calcularArea, setColor, getDireccion, mover, rotar.
Nombre de variable	Comienza con letra minúscula	area, color, angulo, x, ancho.
Nombre de constante	En letras mayúsculas	PI, MAX_ANCHO.

Comentarios

Un comentario es un texto adicional que se añade al código para explicar su funcionalidad, bien a otras personas que lean el programa, o al propio autor como recordatorio. Los comentarios son una parte importante de la documentación de un programa. Los comentarios son ignorados por el compilador, por lo que no incrementan el tamaño del archivo ejecutable; se pueden, por tanto, añadir libremente al código para que pueda entenderse mejor.

La programación orientada a objetos facilita mucho la lectura del código, por lo que no se precisa hacer tanto uso de los comentarios como en los lenguajes estructurados. En Java existen tres tipos de comentarios:

- ✓ Comentarios en una sola línea.
- ✓ Comentarios de varias líneas.
- ✓ Comentarios de documentación

Como se observa un comentario en varias líneas es un bloque de texto situado entre el símbolo de comienzo del bloque **/*,** y otro de terminación del mismo ***/.** Teniendo en cuenta este hecho, los programadores diseñan comentarios como el siguiente en el cual se suministra información básica, por ejemplo, de una clase de una aplicación (autor y otros datos),


```
/*--------------------------------
(C) Diego Aristizábal Ramírez
fecha: Enero 2017
Clase: Resorte.java
-------------------------------- */
```

Este tipo de comentario también es usado para anular un bloque código con el fin, por ejemplo, de hacer pruebas u algún otro objetivo que considere el programador,

```
/*
Eje eje_x= new Eje2D();
eje_x.setPosicion(20f, 10f);
eje_x.setLongitud(20f);
eje_x.setColor(Color.RED);
eje_x.setEtiqueta("Eje x");
eje_x.setTamanoLetra(2f);
*/
```

Los comentarios de documentación es un bloque de texto situado entre el símbolo de comienzo del bloque /**, y otro de terminación de este */. El programa ***javadoc*** utiliza estos comentarios para generar la documentación del código (documentación de una API),

```
/**
* Este método modifica el radio del objeto de laboratorio
Polea2D
* @param radio
*/

public void setRadio(float radio){

        this.radio=radio;

    }
```


Habitualmente, se usarán comentarios en una sola línea //, también muy usados para asuntos de prueba de código donde se deben anular una o pocas líneas de éste, o para dar información que no necesite documentarse en la API,

```
public class Polea {

private double radio;
private Color color;

//constructor por defecto de la polea
public Polea() {


}
}
```

Un procedimiento elemental de depuración de un programa consiste en anular ciertas sentencias de un programa mediante los delimitadores de comentarios. Por ejemplo,

se puede modificar el programa y anular la sentencia que imprime el mensaje, poniendo delante de ella el delimitador de comentarios en una sola línea.

Sentencias

Una sentencia es una orden que se le da al programa para realizar una tarea específica, esta puede ser: mostrar un mensaje en la pantalla, declarar una, inicializarla, llamar a un método, etc. Las sentencias acaban con un "punto y coma (;)". Este carácter separa una sentencia de la siguiente. Normalmente, las sentencias se ponen unas debajo de otras, aunque sentencias cortas pueden colocarse en una misma línea pero pierde legibilidad el programa (se dificulta su lectura pero no hay error en el mismo). He aquí algunos ejemplos de sentencias:

```
int i=1;
import java.awt.*;
rect.mover(10, 20);
```

En el lenguaje Java, los caracteres espacio en blanco se pueden emplear libremente. Como podremos ver en los sucesivos ejemplos, es muy importante para la legibilidad de un programa la colocación de unas líneas debajo de otras empleando tabuladores. El editor del entorno de desarrollo integrado (IDE, por sus siglas en inglés que en nuestro caso será el Android Studio) ayudará plenamente en esto de forma automática.

Bloques de código

Un bloque de código es un grupo de sentencias que se comportan como una unidad. Un bloque de código está limitado por las llaves de apertura {**y** cierre }. Son ejemplos, el código de una clase, el código de un método, el código de una sentencia iterativa **for**, el código de los bloques **try** ... **catch** para el tratamiento de las excepciones, etc. Todos estos códigos se encierran entre llaves. Ejemplo,


```
public Resorte2D(){

        this.longitud=10f;
        this.ancho=2f;
    }
```

Expresiones

Una expresión es todo aquello que se puede poner a la derecha del operador asignación =. Por ejemplo:

```
s=235;
p=(3*x+10.0)/2;
area=rectangulo.calcularArea(2.5,3.8);
Resorte r=new Resorte(Color rojo,10);
```

La primera expresión asigna un valor a la variable s La segunda, realiza una operación. La tercera, es una llamada a un método llamado calcularArea desde un objeto rectangulo de una clase determinada. La cuarta, crea un objeto de tipo Resorte usando un método constructor que asigna dos parámetros, uno de ellos es el color del resorte y el otro

podría ser el número de espiras (esto dependerá de lo que diga la documentación de la clase Resorte).

Variables

Todas las variables han de declararse antes de usarlas, la declaración consiste en una sentencia en la que figura el tipo de dato y el nombre que se asigna a la variable. Una vez declarada se le podrá asignar valores.

Java tiene tres tipos de variables:

- ✓ de instancia,
- ✓ de clase,
- ✓ locales.

Las variables de instancia se usan para guardar los atributos de un objeto particular.

Las variables de clase (son estáticas) son similares a las variables de instancia, con la excepción de que los valores que guardan son los mismos para todos los objetos de una determinada clase. En el siguiente ejemplo, PI es una variable de clase y radio es una variable de instancia. PI guarda el mismo valor para todos los objetos de la clase Circulo, pero el radio de cada círculo puede ser diferente

```
class Circulo {
static final double PI=3.1416;
double radio;
//...
}
```

Las variables locales se utilizan dentro de los métodos. En el siguiente ejemplo area es una variable local al método calcularArea en la que se guarda el valor del área de un objeto de la clase Circulo. Una variable local existe desde el momento de su definición hasta el final del bloque en el que se encuentra.


```
class Circulo{
//...
double calcularArea(){
double area=PI*radio*radio;
return area;
}
}
```

En el lenguaje Java, las variables locales se declaran en el momento en el que son necesarias. Es una buena costumbre inicializar las variables en el momento en el que son declaradas. Ejemplos:

```
float radio=20;
String nombre="Patricia";
double a=3.5, b=0.0, c=-2.4;
boolean mover=true;
double[] datos;
```

Delante del nombre de cada variable se ha de especificar el tipo de variable que hemos destacado en letra negrita. Las variables pueden ser:

- ✓ Un tipo de dato primitivo.
- ✓ El nombre de una clase.
- ✓ Un array.

El lenguaje Java utiliza el conjunto de caracteres Unicode, que incluye no solamente el conjunto ASCII sino también caracteres específicos de la mayoría de los alfabetos. Así, se puede declarar una variable que contenga la letra ñ

```
int año=2018;
```


Se ha de poner nombres significativos a las variables, generalmente formados por varias palabras combinadas, la primera empieza por minúscula, pero las que le siguen llevan la letra inicial en mayúsculas.

```
double radioCirculo=8.05;
```

Las variables son uno de los elementos básicos de un programa, y se deben

- ✓ Declarar.
- ✓ Inicializar.
- ✓ Usar.

Tipos de datos primitivos

Tipo	Descripción
boolean	Tiene dos valores **true** o **false.**
char	Caracteres Unicode de 16 bits. Los caracteres alfa-numéricos son los mismos que los ASCII con el bit alto puesto a 0. El intervalo de valores va desde 0 hasta 65535 (valores de 16-bits sin signo).
byte	Tamaño 8 bits. El intervalo de valores va desde -2^7 hasta 2^7 -1 (-128 a 127)
short	Tamaño 16 bits. El intervalo de valores va desde -2^{15} hasta 2^{15}-1 (-

	32768 a 32767)
int	Tamaño 32 bits. El intervalo de valores va desde -2^{31} hasta $2^{31}-1$ (-2147483648 a 2147483647)
long	Tamaño 64 bits. El intervalo de valores va desde -2^{63} hasta $2^{63}-1$ (- 9223372036854775808 a 9223372036854775807)
float	Tamaño 32 bits. Números en coma flotante de simple precisión. Estándar IEEE 754-1985 (de 1.40239846e-45f a 3.40282347e+38f)
double	Tamaño 64 bits. Números en coma flotante de doble precisión. Estándar IEEE 754-1985. (de 4.94065645841246544e-324d a 1.7976931348623157e+308d.)

Caracteres

En Java los caracteres no están restringidos a los ASCII sino son Unicode. Un carácter está siempre rodeado de comillas simples como 'A', '9', 'ñ', etc. El tipo de dato **char** sirve para guardar estos caracteres.

Un tipo especial de carácter es la secuencia de escape, que se utilizan para representar caracteres de control o caracteres que no se imprimen. Una secuencia de escape está formada por la barra invertida (\) y un carácter. En la siguiente tabla se dan las secuencias de escape más utilizadas.

Carácter	Secuencia de escape
retorno de carro	\r
tabulador horizontal	\t
nueva línea	\n
Barra invertida	\\

Variables booleanas

En Java existe el tipo de dato **boolean**. Una variable booleana solamente puede guardar uno de los dos posibles valores: true (verdadero) y false (falso).

```
boolean seMueve=false;
```

Variables enteras

Una variable entera consiste en cualquier combinación de cifras precedidos por el signo más (opcional), para los positivos, o el signo menos, para los negativos. Son ejemplos de números enteros,

```
18, -26, 0, 4327, -36945
```

Son ejemplos de declaración de variables enteras,

```
int numero=3245;
int x,y;
long m=30L;
```

int es la palabra reservada para declarar una variable entera. En el primer caso, el compilador reserva una porción de 32 bits de memoria en el que guarda el número

3245. En el segundo caso, las porciones de memoria cuyos nombres son *x* e ***y***, guardan cualquier valor entero si la variable es local o cero si la variable es de instancia o de clase. El uso de una variable local antes de ser convenientemente inicializada puede conducir a consecuencias desastrosas. Por tanto, declarar e inicializar una variable es una práctica aconsejable.

En la tercera línea 30 es un número de tipo **int** por defecto, le ponemos el sufijo **L** en mayúsculas o minúsculas para indicar que es de tipo **long**.

Existen como se ve en la tabla varios tipos de números enteros (**byte**, **short**, **int**, **long**), y también existe una clase denominada BigInteger cuyos objetos pueden guardar un número entero arbitrariamente grande.

Variables en punto flotante

Las variables del tipo **float** o **double** (punto flotante) se usan para guardar números en memoria que tienen parte entera y parte decimal.

```
double PI=3.14159;
double g=9.7805;
double c=3.0e8;
```

El primero es una aproximación del número real π, el segundo es la aceleración de la gravedad a nivel del mar en $m.s^{-2}$, el tercero es la velocidad de la luz en $m.s^{-1}$ en el vacío, que es la forma de escribir $3.0x10^{8}$. El carácter punto '.', separa la parte entera de la parte decimal, en vez del carácter coma ',' que usamos habitualmente en nuestro idioma.

Otros ejemplos son los siguientes

```
float a=12.5f;
float b=7f;
double c=7.0;
double d=7d;
```


En la primera línea 12.5 lleva el sufijo **f**, ya que por defecto 12.5 es **double**. En la segunda línea 7 es un entero y por tanto 7f es un número de tipo **float**. Y así el resto de los ejemplos.

Conceptualmente, hay infinitos números de valores entre dos números reales. Ya que los valores de las variables se guardan en un número prefijado de bits, algunos valores no se pueden representar de forma precisa en memoria. Por tanto, los valores de las variables en punto flotante en un computador solamente se aproximan a los verdaderos números reales en matemáticas. La aproximación es tanto mejor, cuanto mayor sea el tamaño de la memoria que se reserva para guardarlo. De este hecho, surgen las variables del tipo **float** y **double**. Para números de precisión arbitraria se emplea la clase BigDecimal.

Valores constantes

Cuando se declara una variable de tipo **final**, se ha de inicializar y cualquier intento de modificarla en el curso de la ejecución del programa da lugar a un error en tiempo de compilación.

Normalmente, las constantes de un programa se suelen poner en letras mayúsculas, para distinguirlas de las que no son constantes. He aquí ejemplos de declaración de constantes.

```
final double PI=3.141592653589793;
final int MAX_DATOS=150;
```

Las cadenas de caracteres o strings

Además de los ocho tipos de datos primitivos, las variables en Java pueden ser declaradas para guardar una instancia de una clase, como se verá en el siguiente módulo (Clases y Objetos).

En Java las cadenas de caracteres o strings son objetos de la clase String.

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

En una cadena se pueden insertar caracteres especiales como el carácter tabulador '\t' o el de nueva línea '\n'

```
String texto="Un string con \t un carácter tabulador y \n un salto de
              línea";
```

Palabras reservadas

En el siguiente muestra las palabras reservadas por el lenguaje Java (Java tiene definidas 50 palabras y se reserva 8 más), y que el programador no puede utilizar como identificadores.

Palabras claves

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**

Otras

cast	**future**	**generic**	**inner**
operator	**outer**	**rest**	**var**

Las palabras reservadas se pueden clasificar en las siguientes categorías:

- ✓ Tipos de datos: **boolean, float, double, int, char.**
- ✓ Sentencias condicionales: **if, else, switch.**
- ✓ Sentencias iterativas: **for, do, while, continue.**

- ✓ Tratamiento de las excepciones: **try, catch, finally, throw.**
- ✓ Estructura de datos: **class, interface, implements, extends.**
- ✓ Modificadores y control de acceso: **public, private, protected, transient.**
- ✓ Otras: **super, null, this.**

OPERADORES ARITMÉTICOS

Java tiene cinco operadores aritméticos cuyo significado se muestra en la tabla adjunta.

Operador	Nombre	Ejemplo
+	Suma	10+8
-	Diferencia	10-8
*	Producto	10*8
/	Cociente	20/3
%	Módulo	20%3

El cociente entre dos enteros da como resultado un entero. Por ejemplo, al dividir 20 entre 3 da como resultado 6.

El operador módulo da como resultado el resto de la división entera. Por ejemplo 20%3 da como resultado 2 que es el resto de la división entre 20 y 3.

El operador asignación

El operador más importante y más frecuentemente usado es el operador asignación =, que se ha empleado para la inicialización de las variables. Así,

```
int x;
x=15;
```

la primera sentencia declara una variable entera de tipo **int** y le da un nombre (numero). La segunda sentencia usa el operador asignación para inicializar la variable con el número 15.

Considerar ahora, la siguiente sentencia.

```
a=b;
```

que asigna a a el valor de b. A la izquierda siempre se tendrá una variable tal como a, que recibe valores, a la derecha otra variable b, o expresión que tiene un valor. Por tanto, tienen sentido las expresiones

```
a=1234;
double area=calculaArea(radio);
perimetro=2*ancho+2*alto;
```

Sin embargo, no tienen sentido las expresiones

```
1234=a;
calculaArea(radio)=area;
```

Las asignaciones múltiples son también posibles. Por ejemplo, es válida la sentencia

```
c=a=b; //equivalente a c=(a=b);
```

la cual puede ser empleada para inicializar en la misma línea varias variables

```
c=a=b=321; //asigna 321 a a, b y c
```

El operador asignación se puede combinar con los operadores aritméticos

Expresión	Significado
x+=y	x=x+y
x-=y	x=x-y
x*=y	x=x*y
x/=y	X=x/y

Así, la sentencia

```
x=x+10;
```

evalúa la expresión x+10, que es asignada de nuevo a x. El compilador lee primero el contenido de la porción de memoria nombrada x, realiza la suma, y guarda el resultado en la misma porción de memoria. Se puede escribir la sentencia anterior de una forma equivalente más simple

```
x+=10;
```

Concatenación de strings

En Java se usa el operador + para concatenar cadenas de caracteres o strings,

```
double d= 20.0;
String distancia= "la distancia es en m "+d;
```

El operador + cuando se utiliza con strings y otros objetos, crea un solo string que contiene la concatenación de todos sus operandos. Si alguno de los operandos no es una cadena, se convierte automáticamente en una cadena. Por ejemplo, en la sentencia anterior el número del tipo **double** que guarda la variable v se convierte en un string que se añade al string "la distancia es en m ".

La precedencia de operadores

Los operadores aritméticos tienen distinta precedencia, así la expresión

```
a+b*c;
```

es equivalente a

```
a+(b*c);
```

ya que el producto y el cociente tienen mayor precedencia que la suma o la resta. Por tanto, en la segunda expresión el paréntesis no es necesario. Sin embargo, si se quiere efectuar antes la suma que la multiplicación se tiene que emplear los paréntesis

```
(a+b)*c;
```

Para realizar la operación

$$\frac{a}{b\,c}$$

se escribe,

```
a/(b*c);
```

o bien,

```
a/b/c;
```

En la mayoría de los casos, la precedencia de las operaciones es evidente, sin embargo, en otros que no lo son tanto, se aconseja emplear paréntesis.

La conversión automática y promoción (casting)

Cuando se realiza una operación, si un operando es entero (**int**) y el otro es de punto flotante (**double**) el resultado es en punto flotante (**double**).

```
int a=10;
double b=263.2;
double suma=a+b;
```

Cuando se declaran dos variables una de tipo **int** y otra de tipo **double**.

```
int x;
double y=23.205;
```

¿qué ocurrirá cuando si se asigna a la variable x el número guardado en la variable y?. Recordar que se trata de dos tipos de variables distintos cuyo tamaño en memoria es de 32 y 64 bits respectivamente. Por tanto, la sentencia

```
x=y;
```

convierte el número real en un número entero eliminando los decimales. La variable x guardará el número 23.

Se ha de tener cuidado, ya que la conversión de un tipo de dato en otro es una fuente frecuente de error entre los programadores principiantes. Ahora bien, suponer que se desea calcular la división 7/3, como se ha visto, el resultado de la división entera es 2, aún en el caso de que se trate de guardar el resultado en una variable del tipo **double**, como lo prueba la siguiente porción de código.


```
int x=7;
int y=3;
double z=x/y;
```

Si se quiere obtener una aproximación decimal del número 7/3, se debe promocionar el entero x a un número en coma flotante, mediante un procedimiento denominado promoción o *casting*.

```
int x=7;
int y=3;
double z=(double)x/y;
```

Los operadores unarios

Los operadores unarios son:

- ✓ ++ Incremento.
- ✓ -- Decremento

actúan sobre un único operando. Se trata de uno de los aspecto más confusos para el programador, ya que el resultado de la operación depende de que el operador esté a la derecha i++ o a la izquierda ++i.

Así,

```
i=i+1; //añadir 1 a i
i++;
```

Del mismo modo, lo son

```
i=i-1; //restar 1 a i
i--;
```


Examinar ahora, la posición del operador respecto del operando. Considerar en primer lugar que el operador unario ++ está a la derecha del operando. La sentencia

```
j=i++;
```

asigna a j, el valor que tenía i. Por ejemplo, si i valía 3, después de ejecutar la sentencia, j toma el valor de 3 e i el valor de 4. Lo que es equivalente a las dos sentencias

```
j=i;
i++;
```

Un resultado distinto se obtiene si el operador ++ está a la izquierda del operando

```
j=++i;
```

asigna a j el valor incrementado de i. Por ejemplo, si i valía 3, después de ejecutar la sentencia j e i toman el valor de 4. Lo que es equivalente a las dos sentencias

```
++i;
j=i;
```

OPERADORES RELACIONALES

Los operadores relacionales son símbolos que se usan para comparar dos valores. Si el resultado de la comparación es correcto la expresión considerada es verdadera, en caso contrario es falsa. Por ejemplo, 5>3 es verdadera, se representa por el valor **true** del tipo básico **boolean**, en cambio, 5<3 es falsa, **false**. En la primera columna de la tabla, se dan los símbolos de los operadores relacionales, la segunda, el nombre de dichos operadores, y a continuación su significado mediante un ejemplo.

Operador	nombre	ejemplo	significado
<	menor que	a<b	a es menor que b
>	mayor que	a>b	a es mayor que b
==	igual a	a==b	a es igual a b
!=	no igual a	a!=b	a no es igual a b
<=	menor que o igual a	a<=5	a es menor que o igual a b
>=	mayor que o igual a	a>=b	a es menor que o igual a b

Se debe tener especial cuidado en no confundir el operador asignación con el operador relacional igual a. Las asignaciones se realizan con el símbolo **=**, las comparaciones con **==**.

Los operadores lógicos

Los operadores lógicos son:

- ✓ && AND (el resultado es verdadero si ambas expresiones son verdaderas).
- ✓ || OR (el resultado es verdadero si alguna expresión es verdadera).
- ✓ ! NOT (el resultado invierte la condición de la expresión).

AND y OR trabajan con dos operandos y retornan un valor lógico basadas en las denominadas tablas de verdad. El operador NOT actúa sobre un operando. Las tablas de verdad de los operadores AND, OR y NOT se muestran en las tablas siguientes

El operador lógico AND

x	y	resultado
true	true	true
true	false	false
false	true	false
false	false	false

El operador lógico OR

x	y	resultado
true	true	true
true	false	true
false	true	true

false	false	false

El operador lógico NOT

x	resultado
true	false
false	true

Los operadores AND y OR combinan expresiones relacionales cuyo resultado viene dado por la última columna de sus tablas de verdad. Por ejemplo:

```
(a<b) && (b<c);
```

es verdadero (**true**), si ambas son verdaderas. Si alguna o ambas son falsas el resultado es falso (**false**). En cambio, la expresión

```
(a<b) ||(b<c);
```

es verdadera si una de las dos comparaciones lo es. Si ambas, son falsas, el resultado es falso.

La expresión " NO a es menor que b"

```
!(a<b) ;
```

es falsa si *(a<b)* es verdadero, y es verdadera si la comparación es falsa. Por tanto, el operador NOT actuando sobre *(a<b)* es equivalente a

(a>=b);

La expresión "NO a es igual a b"
!(a==b);

es verdadera si a es distinto de b, y es falsa si a es igual a b. Esta expresión es equivalente a

(a!=b);

ARREGLOS

Un array (arreglo) es un medio de guardar un conjunto de objetos de la misma clase. Se accede a cada elemento individual del array mediante un número entero denominado índice. 0 es el índice del primer elemento y *n-1* es el índice del último elemento, siendo *n*, la dimensión del array. Los arrays son objetos en Java y como tales vamos a ver los pasos que hemos de seguir para usarlos convenientemente

- ✓ Declarar el array
- ✓ Crear el array
- ✓ Inicializar los elementos del array
- ✓ Usar el array

Declarar y crear un array

Para declarar un array se escribe

tipo_de_dato[] nombre_del_array;

Para declarar un array de enteros escribimos

int[] x;

Para crear un array de 4 números enteros se escribe

```
x=new int[4];
```

La declaración y la creación del array se pueden hacer en una misma línea.

```
int[] x =new int[4];
```

Inicializar y usar los elementos del array

Para inicializar el array de 4 enteros escribimos

```
x[0]=2;
x[1]=-4;
x[2]=15;
x[3]=-25;
```

Se pueden inicializar en un bucle **for** como resultado de alguna operación

```
for(int i=0; i<4; i++){
      x[i]=i*i+4;
}
```

No se necesita recordar el número de elementos del array, su miembro dato length nos proporciona la dimensión del array. Se escribe de forma equivalente

```
for(int i=0; i<x.length; i++){
      x[i]=i*i+4;
}
```

Los arrays se pueden declarar, crear e inicializar en una misma línea, del siguiente modo

```
int[] x={2, -4, 15, -25};
String[] nombres={"Patricia", "Camila", "Diego", "Rocko"};
```

Java verifica que el índice no sea mayor o igual que la dimensión del array, lo que facilita mucho el trabajo al programador.

Para crear un array de tres objetos de la clase Polea se escribe

- Declarar

```
Polea[] poleas;
```

- Crear el array

```
poleas=new Polea[3];
```

- Inicializar los elementos del array

```
poleas[0]=new Polea(Color.RED, 20);
poleas[1]=new Polea(Color.BLACK, 40);
poleas[2]=new Polea(Color.BLUE, 80);
```

O bien, en una sola línea

```
Polea[] poleas={ Polea(Color.RED, 20),
      new Polea(Color.BLACK, 40), Polea(Color.BLUE, 80)};
```

Arrays multidimensionales

Un arreglo bidimensional puede tener varias filas, y en cada fila no tiene por qué haber el mismo número de elementos o columnas. Por ejemplo, se puede declarar e inicializar la siguiente matriz bidimensional

```
double[][] x={{1,2,3,4},{5,6},{7,8,9,10,11,12},{13}};
```

- La primer fila tiene cuatro elementos {1,2,3,4}
- La segunda fila tiene dos elementos {5,6}

- La tercera fila tiene seis elementos {7,8,9,10,11,12}
- La cuarta fila tiene un elemento {13}

VECTORES

Los arrays en Java son suficientes para guardar tipos básicos de datos, y objetos de una determinada clase cuyo número se conoce de antemano. Algunas veces se desea guardar objetos en un array pero no se sabe cuántos objetos se van a guardar. Una solución es la de crear un array cuya dimensión sea más grande que el número de elementos que se necesite guardar. La clase Vector proporciona una solución alternativa a este problema. Un vector es similar a un array, la diferencia estriba en que un vector crece automáticamente cuando alcanza la dimensión inicial máxima. Además, proporciona métodos adicionales para añadir, eliminar e insertar elementos entre otros.

Crear un vector

Para usar la clase Vector se pone al principio del archivo del código fuente la sentencia **import**

```
import java.util.*;
```

Cuando se crea un *vector* u objeto de la clase Vector, se puede especificar su dimensión inicial, y cuánto crecerá si se rebasa dicha dimensión.

```
Vector v=new Vector(20, 5);
```

Se tiene un vector con una dimensión inicial de 20 elementos. Si se rebasa dicha dimensión y se guardan 21 elementos la dimensión del vector crece a 25.

Al segundo constructor, solamente se le pasa la dimensión inicial.

```
Vector v=new Vector(20);
```

Si se rebasa la dimensión inicial guardando 21 elementos, la dimensión del vector se duplica. El programador ha de tener cuidado con este constructor, ya que si se pretende guardar un número grande de elementos se tiene que especificar el incremento de la capacidad del vector, si no se quiere desperdiciar inútilmente la memoria el computador.

Con el tercer constructor, se crea un vector cuya dimensión inicial es 10.

```
Vector v=new Vector();
```

La dimensión del vector se duplica si se rebasa la dimensión inicial, por ejemplo, cuando se pretende guardar once elementos.

Añadir elementos al vector

Hay dos formas de añadir elementos a un vector. Se puede añadir un elemento a continuación del último elemento del vector, mediante la función miembro addElement.

```
v.addElement("uno");
```

Se puede también insertar un elemento en una determinada posición, mediante insertElementAt. El segundo parámetro o índice, indica el lugar que ocupará el nuevo objeto. Si se trata de insertar un elemento en una posición que no existe todavía se obtiene una **excepción** del tipo ArrayIndexOutOfBounds. Por ejemplo, si se trata de insertar un elemento en la posición 9 cuando el vector solamente tiene cinco elementos.

Para insertar el string "tres" en la tercera posición del vector *v*, escribimos

```
v.insertElementAt("tres", 2);
```

En la siguiente porción de código, se crea un vector con una capacidad inicial de 10 elementos, valor por defecto, y se le añaden o insertan objetos de la clase String.


```
Vector v=new Vector();
v.addElement("uno");
v.addElement("dos");
v.addElement("cuatro");
v.addElement("cinco");
v.addElement("seis");
v.addElement("siete");
v.addElement("ocho");
v.addElement("nueve");
v.addElement("diez");
v.addElement("once");
v.addElement("doce");

v.insertElementAt("tres", 2);
```

Para saber cuántos elementos guarda un vector, se llama al método size. Para saber la dimensión actual de un vector se llama al método capacity. Por ejemplo, en la porción de código se han guardado 12 elementos en el vector v. La dimensión de v es 20, ya que se ha superado la dimensión inicial de 10 establecida en la llamada al tercer constructor cuando se ha creado el vector v.

```
String numeroElementos= v.size();
String dimensión= v.capacity();
```

Se pueden eliminar todos los elementos de un vector, llamando a la función miembro removeAllElements. O bien, se puede eliminar un elemento concreto, por ejemplo el que guarda el string "tres".

```
v.removeElement("tres");
```

Se puede eliminar dicho elemento, si se especifica su índice.


```
v.removeElementAt(2);
```

Acceso a los elementos de un vector

El acceso a los elementos de un vector no es tan sencillo como el acceso a los elementos de un array. En vez de dar un índice, se usa el método elementAt. Por ejemplo, v.elementAt(4) sería equivalente a v[4], si v fuese un array.

EL FLUJO DE UN PROGRAMA

La sentencia if

La sentencia **if**, actúa como cabría esperar, Figura 3. Si la condición es verdadera, la sentencia se ejecuta, de otro modo, se salta dicha sentencia, continuando la ejecución del programa con otras sentencias a continuación de ésta. La forma general de la sentencia **if** es:

```
if (condición)
      sentencia;
```

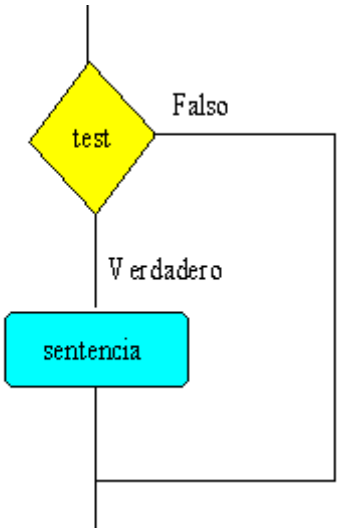

Figura 3

Si el resultado del test es verdadero (**true**) se ejecuta la sentencia que sigue a continuación de **if**, en caso contrario, falso (**false**), se salta dicha sentencia, tal como se indica en la figura. La sentencia puede consistir a su vez, en un conjunto de sentencias agrupadas en un bloque.

```
if (condición){
      sentencia1;
      sentencia2;
}
```

La sentencia if...else

La sentencia **if...else** completa la sentencia **if**, Figura 4, para realizar una acción alternativa

```
if (condición)
      sentencia1;
else
      sentencia2
```

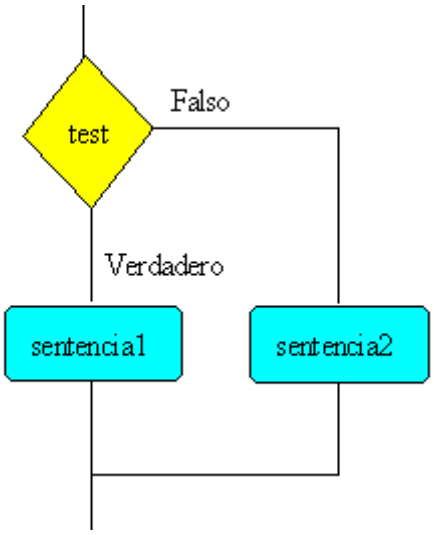

Figura 4

Las dos primeras líneas indican que si la condición es verdadera se ejecuta la sentencia 1. La palabra clave **else**, significa que si la condición no es verdadera se ejecuta la sentencia 2, tal como se ve en la figura..

Dado que las sentencias pueden ser simples o compuestas la forma general de **if...else** es

```
if (condición){
      sentencia1;
      sentencia2;
}else{
      sentencia3
      sentencia4;
      sentencia5;
      }
```

La sentencia switch

Como podemos ver en la figura del apartado anterior, la sentencia **if...else** tiene dos ramas, el programa va por una u otra rama dependiendo del valor verdadero o falso de la

expresión evaluada. A veces, es necesario, elegir entre varias alternativas, Figura 5

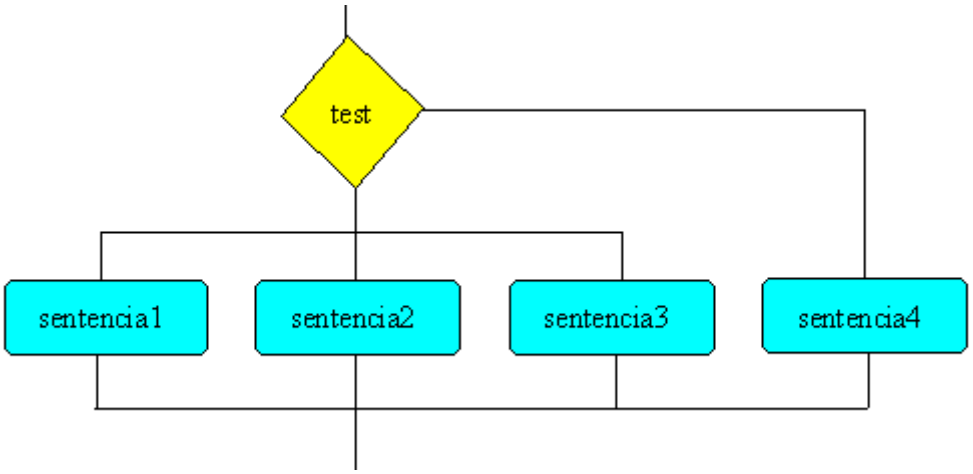

Figura 5

Por ejemplo, considérese las siguientes series de sentencias **if...else**

```
if(expresion==valor1)
      sentencia1;
else if(expresion==valor2)
      sentencia2;
else if(expresion==valor3)
      sentencia3;
else
      sentencia4;
```

El código resultante puede ser difícil de seguir y confuso incluso para el programador avanzado. El lenguaje Java proporciona una solución elegante a este problema mediante la sentencia condicional **switch** para agrupar a un conjunto de sentencias **if...else**.

```
switch(expresion){
      case valor1:
            sentencia1;
```

```
            break;              //sale de switch
      case valor2:
            sentencia2;
            break;              //sale switch
      case valor3:
            sentencia3;
            break;              //sale de switch
      default:
            sentencia4;
      }
```


En la sentencia **switch**, se compara el valor de una variable o el resultado de evaluar una expresión, con un conjunto de números enteros valor1, valor2, valor3, .. o con un conjunto de caracteres, cuando coinciden se ejecuta el bloque de sentencias que están asociadas con dicho número o carácter constante. Dicho bloque de sentencias no está entre llaves sino que empieza en la palabra reservada **case** y termina en su asociado **break**. Si el compilador no encuentra coincidencia, se ejecuta la sentencia **default**, si es que está presente en el código.

La sentencia for

Esta sentencia se encuentra en la mayoría de los lenguajes de programación, Figura 6. El bucle **for** se empleará cuando conocemos el número de veces que se ejecutará una sentencia o un bloque de sentencias, tal como se indica en la figura. La forma general que adopta la sentencia **for** es

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

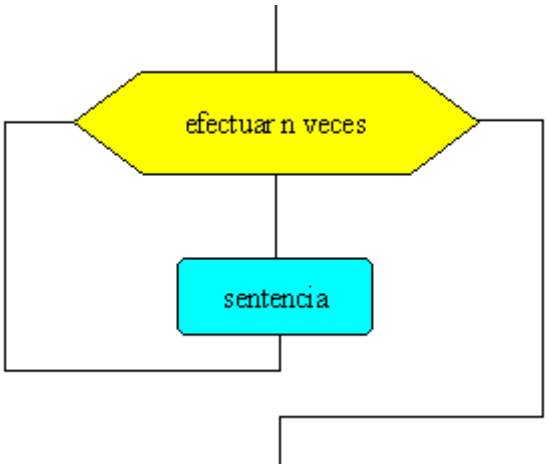

Figura 6

El primer término *inicialización*, se usa para inicializar una variable índice, que controla el número de veces que se ejecutará el bucle. La *condición* representa la condición que ha de ser satisfecha para que el bucle continúe su ejecución. El *incremento* representa la cantidad que se incrementa la variable índice en cada repetición.

Ejemplo: Escribir un programa que imprima los primeros 10 primeros números enteros empezando por el cero

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

El resultado será: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

La sentencia while

A la palabra reservada **while** le sigue una condición encerrada entre paréntesis, Figura 7. El bloque de sentencias que le siguen se ejecuta siempre que la condición sea verdadera tal como se ve en la figura. La forma general que adopta la sentencia **while** es:

```
while (condición)
```

```
sentencia;
```

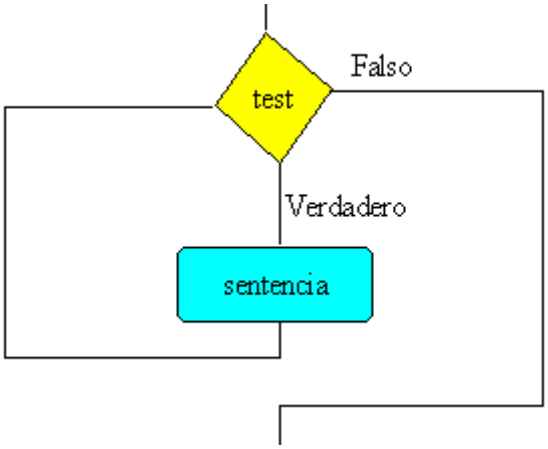

Figura 7

Ejemplo: Escribir un programa que imprima los primeros 10 primeros números enteros empezando por el cero, empleando la sentencia iterativa while.

```
int i=0;

while (i<10) {
      System.out.println(i);
      i++;
}
```

El valor inicial de i es cero, se comprueba la condición (i<10), la cual resulta verdadera. Dentro del bucle, se imprime i, y se incrementa la variable contador i, en una unidad. Cuando i vale 10, la condición (i<10) resulta falsa y el bucle ya no se ejecuta. Si el valor inicial de i fuese 10, no se ejecutaría el bucle. Por tanto, el bucle **while** no se ejecuta si la condición es falsa.

La sentencia do...while

Como se ha podido apreciar las sentencias **for** y **while** la condición está al principio del bucle, sin embargo, **do...while** la condición está al final del bucle, por lo que el bucle se ejecuta por lo menos una vez tal como se ve en la Figura 8. **do** marca el comienzo del bucle y **while** el final del mismo. La forma general es:


```
do{
      sentencia;
}while(condición);
```

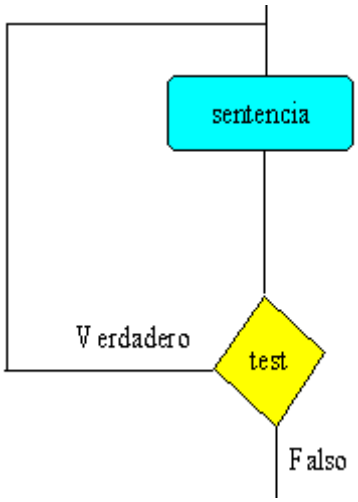

Figura 8

Ejemplo: Escribir un programa que imprima los primeros 10 primeros números enteros empezando por el cero, empleando la sentencia iterativa do..while.

```
int i=0;

do{
System.out.println(i);
i++;
  }while(i < 10);
```

El bucle **do...while**, se usa menos que el bucle **while**, ya que habitualmente evaluamos la expresión que controla el bucle al comienzo, no al final.

La sentencia break

A veces es necesario interrumpir la ejecución de un bucle **for**, ***while***, o **do...while**.


```
for(int i = 0; i < 10; i++){
	if (i == 8)  break;
	System.out.println(i);
}
```

Considerar de nuevo el ejemplo del bucle **for**, que imprime los 10 primeros números enteros, se interrumpe la ejecución del bucle cuando se cumple la condición de que la variable contador *i* valga 8. El código se leerá: "salir del bucle cuando la variable contador i, sea igual a 8".

Como podemos apreciar, la ejecución del bucle finaliza prematuramente. Quizás el lector pueda pensar que esto no es de gran utilidad pues, el código anterior es equivalente a

```
for(int i = 0; i <=8; i++)
	System.out.println(i);
```

Sin embargo, se puede salir fuera del bucle prematuramente si se cumple alguna condición de finalización.

```
while(true){
	if (condicionFinal)break;
	//...otras sentencias
}
```

Como se aprecia en esta porción de código, la expresión en el bucle **while** es siempre verdadera, por tanto, tiene que haber algún mecanismo que permita salir del bucle. Si la condición de finalización es verdadera, es decir la variable *condicionFinal* del tipo **boolean** toma el valor **true**, se sale del bucle, en caso contrario se continua el procesamiento de los datos.

La sentencia continue

La sentencia **continue**, exige al bucle a comenzar la siguiente iteración desde el principio. En la siguiente porción de código, se imprimen todos los números del 0 al 9 excepto el número 8.

```
for(int i = 0; i < 10; i++){
      if (i == 8)  continue;
      System.out.println(i);
}
```

Etiquetas

Tanto **break** como **continue** pueden tener una etiqueta opcional que indica a Java hacia dónde dirigirse cuando se cumple una determinada condición.

```
salida:
      for(int i=0; i<20; i++){
            while(j<70){
                  if(i*j==500)
                  break salida;
                  //...
            }
            //...
      }
```

La etiqueta en este ejemplo se denomina salida, y se añade antes de la parte inicial del ciclo. La etiqueta debe terminar con el carácter dos puntos :. Si no disponemos de etiqueta, al cumplirse la condición i*j==500, se saldría del bucle interno **while**, pero el proceso de cálculo continuaría en el bucle externo **for**.

MANEJO DE EXCEPCIONES

Los programadores de cualquier lenguaje se esfuerzan por escribir programas libres de errores, sin embargo, es muy difícil que los programas reales se vean libres de ellos. En Java las situaciones que pueden provocar un fallo en el programa se denominan excepciones.

Java lanza una excepción en respuesta a una situación poco usual. El programador también puede lanzar sus propias excepciones. Las excepciones en Java son objetos de clases derivadas de la clase base Exception. Existen también los errores internos que son objetos de la clase Error que no estudiaremos. Ambas clases Error y Exception son clases derivadas de la clase base Throwable.

Existe toda una jerarquía de clases derivada de la clase base Exception. Estas clases derivadas se ubican en dos grupos principales:

Las excepciones en tiempo de ejecución ocurren cuando el programador no ha tenido cuidado al escribir su código. Por ejemplo, cuando se sobrepasa la dimensión de un array se lanza una excepción ArrayIndexOutOfBounds. Cuando se hace uso de una referencia a un objeto que no ha sido creado se lanza la excepción NullPointerException. Estas excepciones le indican al programador que tipos de fallos tiene el programa y que debe arreglarlo antes de proseguir.

El segundo grupo de excepciones, es el más interesante, ya que indican que ha sucedido algo inesperado o fuera de control.

Instrucción Try - Catch

Cuando sabemos que un código podría lanzar un error, como por ejemplo una división entre cero, debemos encerrarla entre un bloque try-catch. Veamos un ejemplo:


```
int a = 5 / 0;
```

esta línea nos lanzaría la siguiente excepción:

```
Exception in thread "main" java.lang.ArithmeticException: /
by zero
```

Si en cambio atrapamos esta excepción podremos controlarla:

```
try{
    int a = 5 / 0;
}catch(ArithmeticException err){
    int a = 0;
}
```

El Bloque Finally

Finally se utiliza cuando el programador solicita ciertos recursos al sistema que se deben liberar, y se coloca después del último bloque catch. Veamos un ejemplo en el que intentamos leer un archivo:

```
FileReader lector = null;
try {
    lector = new FileReader("archivo.txt");
```

```
    int i=0;
    while(i != -1){
        i = lector.read();
        System.out.println((char) i );
    }
} catch (IOException e) {
    System.out.println("Error");
} finally {
    if(lector != null){
            lector.close();
    }
}
```


el código contenido en finally se ejecutará tras terminar el bloque try, haya habido o no excepción, lo que permite liberar los recursos reservados para abrir el archivo.

La Cláusula Throws

Esta cláusula advierte de las excepciones que podría lanzar un método, van entre la declaración del método y su cuerpo (pueden ser varias), así:

```
public static void metodo() throws ArithmeticException{
    try{
        int a = 5 / 0;
    }catch(ArithmeticException err){
        int a = 0;
    }
}
```

La palabra clave throw

throw permite lanzar una excepción propia, esto se verá en el siguiente ejemplo en el que se aprovecha para englobar todo lo visto aquí:

```
public static void main(String[] args) {
    int a;
    try{
        a = dividir(5,0);
    }catch(MalNumeroADividir err){
        System.out.println(err);
    }finally{
        a = 0;
    }
    System.out.println("Valor de a = "+a);
}

public static int dividir(int a, int b)throws
MalNumeroADividir{
    if(b == 0){
        throw new MalNumeroADividir();
    }
    return a / b;
}

public static class MalNumeroADividir extends Exception {
    MalNumeroADividir() {
       super("No es posible dividir entre cero");
   }
}
```


Como se vio el manejo de las excepciones en java no es algo difícil de implementar.

REFERENCIAS

- ✓ Franco G. A (2000)., **Curso de Lenguaje Java, Universidad** del País Vasco. http://www.sc.ehu.es/sbweb/fisica/curso.htm. Consultada [Septiembre 2025].

- ✓ Android Studio. https://developer.android.com/studio. Consultada [Septiembre 2025].