**Operadores**

Un operador lleva a cabo operaciones sobre uno (operador unario), dos (operador binario) o tres (operador ternario) datos u operandos de tipo primitvo devolviendo un valor determinado también de un tipo primitivo. El tipo de valor devuelto tras la evaluación depende del operador y del tipo de los operandos. Por ejemplo, los operadores aritméticos trabajan con operandos numéricos, llevan a cabo operaciones aritméticas básicas y devuelven el valor numérico correspondiente. Los operadores se pueden clasificar en distintos grupos según se muestra en los siguientes apartados.

**Operador asignación:**

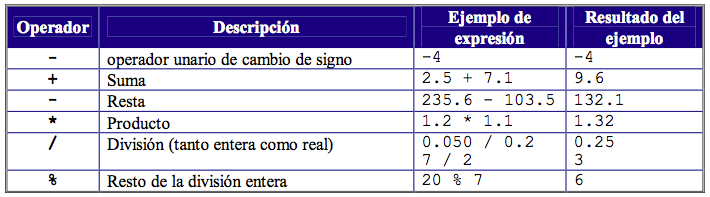


El operador asignación =, es un operador binario que asigna el valor del término de la derecha al operando de la izquierda. El operando de la izquierda suele ser el identificador de una variable. El término de la derecha es, en general, una expresión de un tipo de dato compatible; en particular puede ser una constante u otra variable. Como caso particular, y a diferencia de los demás operadores, este operador no se evalúa devolviendo un determinado valor.

No debe confundirse el operador asignación (=) con el operador relacional de igualdad (==) que se verá más adelante. Además Java dispone de otros operadores que combinan la asignación con otras operaciones (operadores aritméticos combinados).

**Operadores aritméticos:**

El lenguaje de programación Java tiene varios operadores aritméticos para los datos numéricos enteros y reales. En la siguiente tabla se resumen los diferentes operadores de esta categoría.



El resultado exacto depende de los tipos de operando involucrados. Es conveniente tener en cuenta las siguientes peculiaridades:

\* El resultado es de tipo long si, al menos, uno de los operandos es de tipo long y ninguno es real (float o double).

\* El resultado es de tipo int si ninguno de los operandos es de tipo long y tampoco es real (float o double).

\* El resultado es de tipo double si, al menos, uno de los operandos es de tipo double.

\* El resultado es de tipo float si, al menos, uno de los operandos es de tipo float y  ninguno es double.

\* El formato empleado para la representación de datos enteros es el complemento a dos. En la aritmética entera no se producen nunca desbordamientos (overflow) aunque el resultado sobrepase el intervalo de representación (int o long).

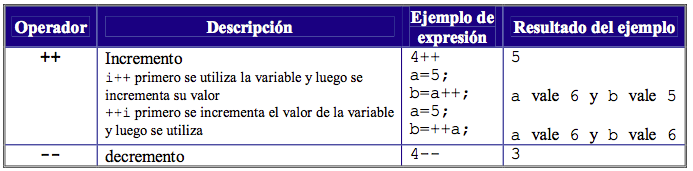
\* La división entera se trunca hacia 0. La división o el resto de dividir por cero es una operación válida que genera una excepción ArithmeticException que puede dar lugar a un error de ejecución y la consiguiente interrupción de la ejecución del programa.

\* La aritmética real (en coma flotante) puede desbordar al infinito (demasiado grande, overflow) o hacia cero (demasiado pequeño, underflow).

\* El resultado de una expresión inválida, por ejemplo, dividir infinito por infinito, no genera una excepción ni un error de ejecución: es un valor NaN (Not a Number).

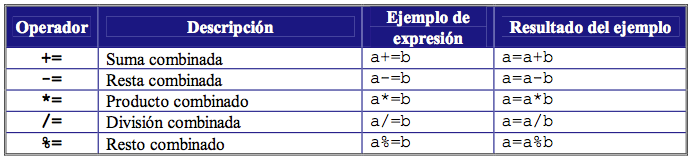
**Operadores aritméticos incrementales**

Los operadores aritméticos incrementales son operadores unarios (un único operando). El operando puede ser numérico o de tipo char y el resultado es del mismo tipo que el operando. Estos operadores pueden emplearse de dos formas dependiendo de su posición con respecto al operando.



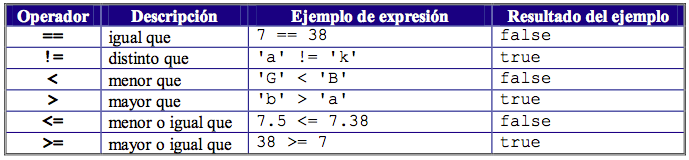
**Operadores aritméticos combinados**

Combinan un operador aritmético con el operador asignación. Como en el caso de los operadores aritméticos pueden tener operandos numéricos enteros o reales y el tipo específico de resultado numérico dependerá del tipo de éstos. En la siguiente tabla se resumen los diferentes operadores de esta categoría.



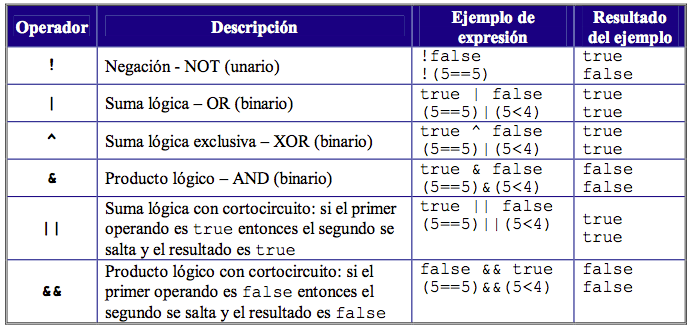
**Operadores de relación**

Realizan comparaciones entre datos compatibles de tipos primitivos (numéricos, carácter y booleanos) teniendo siempre un resultado booleano. Los operandos booleanos sólo pueden emplear los operadores de igualdad y desigualdad.



**Operadores lógicos o booleanos**

Realizan operaciones sobre datos booleanos y tienen como resultado un valor booleano. En la siguiente tabla se resumen los diferentes operadores de esta categoría.



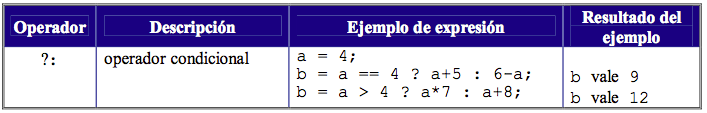
Para mejorar el rendimiento de ejecución del código es recomendable emplear en las expresiones booleanas el operador && en lugar del operador &. En este caso es conveniente situar la condición más propensa a ser falsa en el término de la izquierda. Esta técnica puede reducir el tiempo de ejecución del programa. De forma equivalente es preferible emplear el operador || al operador |. En este caso es conveniente colocar la condición más propensa a ser verdadera en el término de la izquierda.

**El operador condicional**

Este operador ternario tomado de C/C++ permite devolver valores en función de una expresión lógica. Sintaxis:

expresionLogica ? expresion\_1 : expresion\_2

Si el resultado de evaluar la expresión lógica es verdadero, devuelve el valor de la primera expresión, y en caso contrario, devuelve el valor de la segunda expresión.



La sentencia de asignación:

valor = (expresionLogica ? expresion\_1 : expresion\_2);`

como se verá más adelante es equivalente a:

if (expresionLogica)[{

valor = expresion\_1;

}

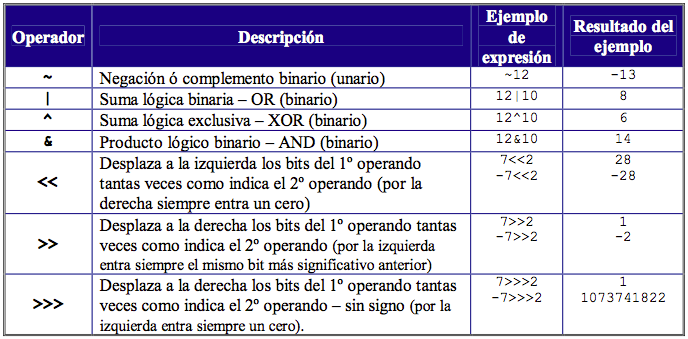
else{

valor = expresion\_2;

}

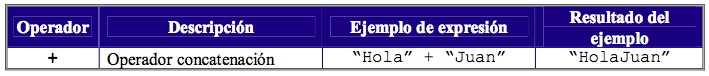
**Operadores de bit**

Tienen operandos de tipo entero (o char) y un resultado de tipo entero. Realizan operaciones con dígitos (ceros y unos) de la representación binaria de los operandos. Exceptuando al operador negación, los demás operadores son binarios. En la siguiente tabla se resumen los diferentes operadores de esta categoría.



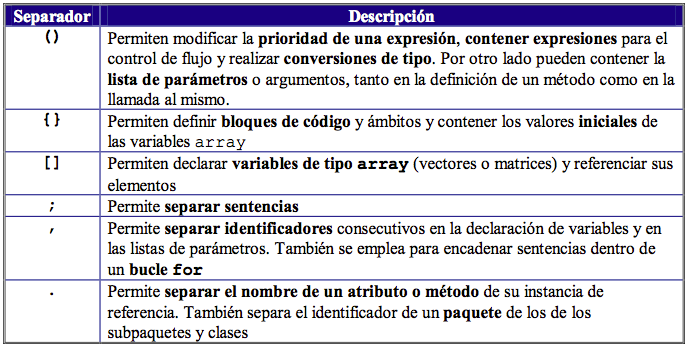
**Operador concatenación de cadenas**

El operador concatenación +, es un operador binario que devuelve una cadena resultado de concatenar las dos cadenas que actúan como operandos. Si sólo uno de los operandos es de tipo cadena, el otro operando se convierte implícitamente en tipo cadena.



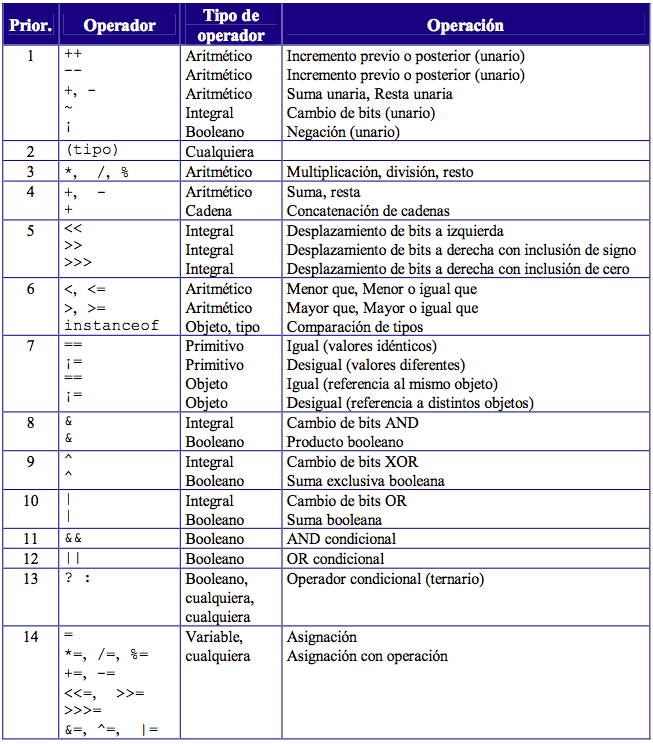
**Separadores**

Existen algunos caracteres que tienen un significado especial en el lenguaje Java. En la siguiente tabla se resumen los diferentes separadores que pueden encontrarse en el código fuente de un programa.



**Prioridad entre operadores**

Si dos operadores se encuentran en la misma expresión, el orden en el que se evalúan puede determinar el valor de la expresión. En la siguiente tabla se muestra el orden o prioridad en el que se ejecutan los operadores que se encuentren en la misma sentencia. Los operadores de la misma prioridad se evalúan de izquierda a derecha dentro de la expresión.



**Sentencias condicionales de control del flujo de un programa**

Cuando se escribe un programa, se introduce la secuencia de sentencias dentro de un archivo. Sin sentencias de control del flujo, el intérprete ejecuta las sentencias conforme aparecen en el programa de principio a fin. Las sentencias de control de flujo se emplean en los programas para ejecutar sentencias condicionalmente, repetir un conjunto de sentencias o, en general, cambiar el flujo secuencial de ejecución. Las sentencias selectivas o condicionales se verán en este capítulo y las sentencias repetitivas en el siguiente.

**Sentencia if-else**

Es una bifurcación o sentencia condicional de una o dos ramas. La sentencia de control evalúa la condición lógica o booleana. Si esta condición es cierta entonces se ejecuta la sentencia o sentencias (1) que se encuentra a continuacion. En caso contrario, se ejecuta la sentencia (2) que sigue a else (si ésta existe). La sentencia puede constar opcionalmente de una o dos ramas con sus correspondientes sentencias.

if (expresionLogica) {

sentencia\_1;

}

o bien (con dos ramas):

if (expresionLogica) {

sentencia\_1;

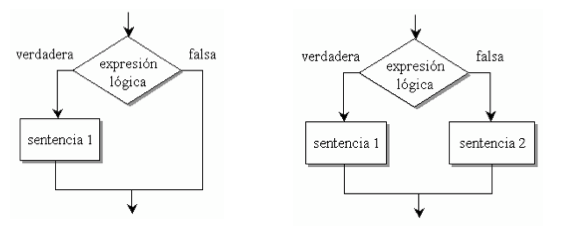
} else {

sentencia\_2;

}

La expresion Logica debe ir entre paréntesis. Las llaves sólo son obligatorias si las sentencias (1) ó (2) son compuestas (las llaves sirven para agrupar varias sentencias simples).

La parte else y la sentencia posterior entre llaves no son obligatorias. En este caso quedaría una sentencia selectiva con una rama.



**Sentencias repetitivas o bucles**

Los bucles, iteraciones o sentencias repetitivas modifican el flujo secuencial de un programa permitiendo la ejecución reiterada de una sentencia o sentencias. En Java hay tres tipos diferentes de bucles: for, while y do-while.

**Sentencia for**

Es un bucle o sentencia repetitiva que

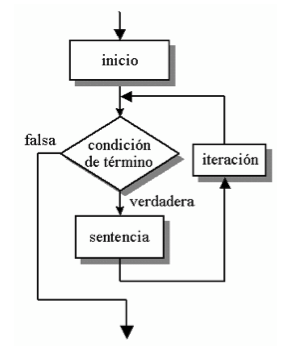
\* ejecuta la sentencia de inicio.

\* verifica la expresión booleana de término.

Sintaxis:

\* si es cierta, ejecuta la sentencia entre llaves y la sentencia de iteración para volver a verificar la expresión booleana de término.

\* si es falsa, sale del bucle.



Sintaxis:

for (inicio; termino; iteracion)

sentencia;

O si se desean repetir varias sentencias:

for (inicio; termino; iteracion) {

sentencia\_1;

sentencia\_2;

sentencia\_n;

}

Las llaves sólo son necesarias si se quieren repetir varias sentencias, aunque se recomienda su porque facilita la lectura del código fuente y ayuda a evitar errores al modificarlo. Habitualmente, en la expresión lógica de término se verifica que la variable de control alcance un uso determinado valor. Por ejemplo:

for (i = valor\_inicial; i <= valor\_final; i++) {

sentencia;

}

Es completamente legal en Java declarar una variable dentro de la cabecera de un bucle for. De esta forma la variable (local) sólo tiene ámbito dentro del bucle. Ejemplo sencillo:

System.out.println("Tabla de multiplicar del 5");

for (int i =0 ; i <= 10; i++) {

System.out.println(5 + " \* " + i + " = " + 5 \* i);

}

Salida por pantalla al ejecutar el código anterior:

5 \* 0 = 0

5 \* 1 = 5

5 \* 2 = 10

5 \* 3 = 15

5 \* 4 = 20

5 \* 5 = 25

5 \* 6 = 30

5 \* 7 = 35

5 \* 8 = 40

5 \* 9 = 45

5 \* 10 = 50

**Sentencia while**

Es un bucle o sentencia repetitiva con una condicion al principio. Se ejecuta una sentencia mientras sea cierta una condición. La sentencia puede que no se ejecute ni una sola vez.

Sintaxis:

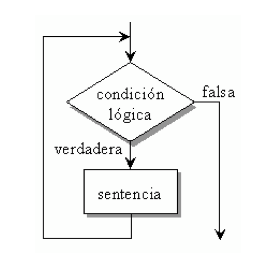
[inicializacion;]

while (expresionLogica) {

sentencias;

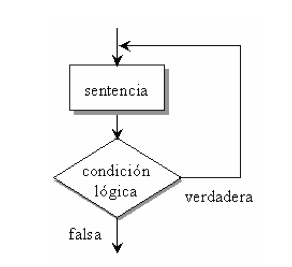
[iteracion;]

}



**Sentencia do-while**

Es un bucle o sentencia repetitiva con una condicion al final. Se ejecuta una sentencia mientras sea cierta una condición. En este caso, la sentencia se ejecuta al menos una vez.



Sintaxis:

do {

sentencias;

[iteracion;]

} while (expresionLogica);

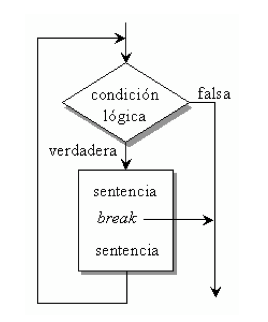
**Sentencia break**

La sentencia break puede encontrarse en sentencias switch o en bucles. Al ejecutarse, deja el ámbito de la sentencia en la que se encuentra y pasa a la siguiente sentencia. Puede emplearse con etiquetas, especificando sobre qué sentencia se aplica si hay varias anidadas.

etiqueta: sentencia;

break [etiqueta];

En la siguiente figura se muestra el diagrama de flujo de un bucle while que contiene una sentencia break:



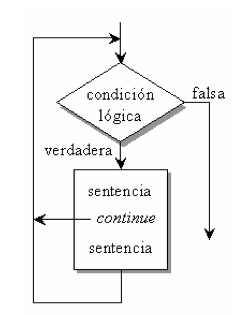
**Sentencia continue**

La sentencia continue se emplea sólo en bucles. Al ejecutarse la iteración en la que se encuentra, el bucle finaliza y se inicia la siguiente. También puede emplearse con etiquetas, especificando sobre que sentencia se aplica si hay varias anidadas.

etiqueta: sentencia;

continue [etiqueta];

En la siguiente figura se muestra el diagrama de flujo de un bucle while que contiene una sentencia continue:



Ejemplo de utilización combinada de break y continue con y sin etiquetas:

uno: for( ... ; ... ; ... ) {

dos: for( ... ; ... ; ... ) {

// Grupo de sentencias A

sentenciasA;

continue; // sigue en el bucle interno

// Grupo de sentencias B

sentenciasB;

continue uno;

// sigue en el principal

// Grupo de sentencias C

sentenciasC;

break uno;

// sale del principal

// Grupo de sentencias D

sentenciasD;

}

}

**Tratamiento de excepciones**

Una excepción es una situación anómala a la que llega la ejecución de un programa. Estas situaciones anómalas pueden ser el intento de abrir un fichero que no existe, la división por cero o el acceso a un objeto no inicializado.

Java proporciona un mecanismo para detectar y solucionar las excepciones que se puede llegar a producir durante la ejecución de un programa. En Java estamos obligados a tratar las excepciones cuando se producen, bien gestionándolas directamente o bien desentendiéndonos de ellas, pero hasta esto último debemos hacerlo explicitamente.

En Java existen dos grandes tipos de excepciones: los Errores y las Excepciones propiamente dichas:

\* Los Errores son situaciones irrecuperables, por ejemplo fallos de la máquina vitual. Ante ellos no hay más alternativa que cerrar la aplicación, y no estamos obligados a gestionarlas.

\* Las excepciones son situaciones anómales ante las cuales bien debemos reaccionar o bien nos desentendemos explítamente. Cuando una excepción se produce se acompaña de toda la información relevante para que podamos gestionarla.

Un caso particular son las excepciones que derivan de RuntimeException, como por ejemplo NullPointerException.

Para gestionar una excepción debe emplearse una sentencia try. La sintaxis de la sentencia se muestra a continuación:

try {

sentencia\_1;

sentencia\_2;

...

}  catch (ClaseExcepcion objetoExcepcion) {

sentencia\_a;

sentencia\_b;

...

} catch(IOException e) {

//Aquí tratamos otro tipo de expeción

} finally {

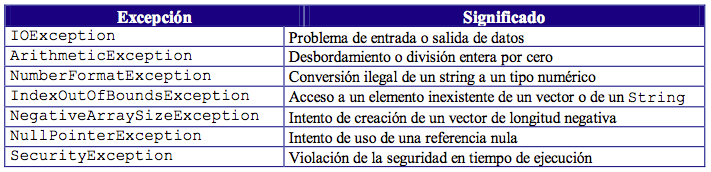
//Aquí realizamos las tareas comunes.

}

La sentencia comienza por la palabra reservada try seguida de una o más sentencias agrupadas entre paréntesis. Dichas sentencias son las que, en principio, pueden dar lugar a una excepción durante su ejecución. La clase Exception, que incluye todos los tipos de excepciones de interés, recoge cualquier excepción generada por el bloque try. Por ejemplo, determinados métodos como readLine generan una excepción si se produce un problema con la entrada o salida de datos. En este caso, se necesita recoger las excepciones de tipo IOException. La ejecución de otros metódos, como parseInt, genera un error si la cadena dada como parámetro no puede convertirse en un entero. En este caso, la excepción es de tipo NumberFormatException.

A continuación aparece una o más clausulas catch que son las manejadoras de las excepciones. En cada una de ellas, se necesita especificar lo que hacer cuando ocurre una excepción en particular. Esta parte del código sólo se ejecuta si se ha producido una excepción.

En Java existen muchos tipos de excepciones estándar. Algunas de las excepciones más comunes se muestran en la siguiente tabla.



**Excepciones y significados**

Como hemos comentado antes, no es necesario que tratemos las excepciones, pero si no lo vamos a hacer, debemos indicarlo explicitamente. El modo de hacerlo es indicando que el método dentro del cual se puede lanzar una excepción a su vez la relanza, como en el siguiente ejemplo:

void metodoLanzador(int a) throws IOException, ClassNotFoundException {...}

**Operaciones de entrada y salida de datos**

Como se ha mostrado en el ejemplo anterior las operaciones de entrada y salida de datos en un programa se llevan a cabo utilizando el paquete java.io. La sentencia import java.io.\*; da acceso a la librería de Java necesaria para cualquier operación de este tipo. Los canales o dispositivos predefinidos para realizar entradas o salidas de datos son los siguientes:

\* System.in:entrada estándar

\* System.out:salida estándar

\* System.err:salida de errores

Los métodos print y println se emplean para la salida de datos en formato de concatenación de Strings. El método readLine facilita una forma sencilla para realizar la entrada de datos mediante un objeto String. Este objeto toma el valor de la cadena de caracteres que acaben en un final de línea o en un final de archivo. Para poder emplear el método readLine es necesario construir un objeto BufferedReader sobre un objeto InputStreamReader, que a su vez se crea a partir de System.in.