# General Java

Java se creó como una herramienta de programación para ser usada en un proyecto de set-top-box en una pequeña operación denominada the Green Project en Sun Microsystems en 1991 (**fecha oficial de creación/lanzamiento en 1995**) con el nombre "OAK", posteriormente cambiado por Green por problemas legales, y finalmente con la denominación actual JAVA. El objetivo de java era crear un lenguaje de programación parecido a C++ en estructura y sintaxis, fuertemente orientado a objetos, pero con una máquina virtual propia.

El lenguaje de programación Java fue desarrollado originalmente por **James Gosling**.

# Características de Java

* Año de creación (1995).
* Es un lenguaje multiplataforma, no está diseñado para un equipo en concreto.
* Es un lenguaje de programación orientado a objetos.
* Se ha creado para usarse en internet.
* Cualquier equipo/dispositivo con una JMV (Máquina Virtual Java) puede leer el código y ejecutarlo.

# Lema de Java

Write Once, Run Everywhere -- Escribir una vez, ejecutar en todas partes

# Conceptos Básicos

* 1. Algoritmo

Un **algoritmo** es un conjunto finito de instrucciones bien definidas que nos ayudan a resolver un problema.

* 1. Lenguajes de programación

Un **lenguaje de programación** puede definirse como un idioma artificial diseñado para que sea fácilmente entendible por un humano e interpretable por una máquina. Un algoritmo escrito utilizando las instrucciones de un lenguaje de programación se le denomina **programa**.

Consta de una serie de reglas y de un conjunto de órdenes o instrucciones. Cada una de estas instrucciones realiza una tarea determinada. A través de una secuencia de instrucciones podemos indicar a una computadora el algoritmo que debe seguir para solucionar un problema dado.

* + 1. Lenguajes compilados e interpretados

Un lenguaje de programación esta diseñado para que una persona escriba fácilmente algoritmos, para que un ordenador comprenda lo que se ha escrito se utiliza una herramienta llamada **compilador**, que transforma el conjunto de ordenes en los ceros y unos comprensibles por la máquina (denominado como **código máquina**).

Existen dos enfoques para realizar el proceso de traducción del lenguaje de programación (código fuente) al código máquina.

* **Compilación:** Traduce todas las instrucciones del código fuente y almacena el código máquina generado. Permite ejecutar el programa, sin volver a compilarlo, tantas veces como se necesite y sin disponer del código fuente.
* **Interpretación:** Consiste en traducir el código fuente instrucción a instrucción e ir ejecutando. Solo se traduce y ejecuta la siguiente instrucción que necesitamos. El proceso continuará sucesivamente hasta que la ejecución termine.

A partir del proceso de compilación e interpretación introducimos dos nuevos conceptos:

* **Tiempo de compilación:** Es el intervalo de tiempo durante el cual se compila un programa.
* **Tiempo de ejecución:** Es el intervalo de tiempo durante el cual un programa se ejecuta.
  + 1. Lenguajes Multiplataforma

El lenguaje C necesita que un mismo programa se compile para cada combinación de tipo de máquina y sistema operativo donde se va a ejecutar. Aunque se pueda ejecutar en distintas plataformas, requiere del compilado específico para cada una de ellas. Por este motivo, **no se considera un lenguaje multiplataforma**.

Java se considera un **lenguaje multiplataforma**. Un mismo programa, una vez compilado, se puede ejecutar en cualquier ordenador que tenga instalada la Máquina Virtual Java, que no es más que un intérprete de bytecode.

Para el caso concreto de Java, vamos a afinar los conceptos de:

* **Tiempo de compilación:** Es el espacio de tiempo en el que se traduce el código fuente al bytecode.
* **Tiempo de ejecución:** Es el tiempo durante el cual el bytecode se interpreta (por la JVM) y se ejecuta por la plataforma correspondiente.

1.5. El programa principal

Para escribir los primeros programas usaremos:

import java.util.Scanner;

import java.lang.Math;

public class Nombre del archivo{

    public static void main(String []args){

Al escribir un programa en Java usaremos literalmente la fórmula anterior.

1.6. Palabras reservadas

En java existe una serie de palabras con un significado especial, como **package**, **class** o **public**. Estas se denominan palabras reservadas y definan la gramática del lenguaje.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Abstract | Continue | Float | Native | Strictfp | Void |
| Assert | Default | For | New | Super | volatile |
| Boolean | Do | If | Package | Switch | While |
| Break | Double | Implements | Private | Synchronized | yield |
| Byte | Else | Import | Protected | This |  |
| Case | Enum | Instanceof | Public | Throw |  |
| Catch | Extends | Int | Return | Throws |  |
| Char | Final | Interface | Short | Transient |  |
| Class | Finally | Long | Static | try |  |

Al conjunto anterior hay que sumar otras palabras reservadas: **const**, **goto**, **true**, **false** y **null**. Las palabras reservadas solo pueden escribirse en determinado lugar de un programa y no pueden ser utilizadas como identificadores.

1.7. Concepto de variable

Una **variable** es una representación, mediante un identificador, de un valor, que puede cambiar durante la ejecución de un programa. A las variables se les asignan valores concretos por medio del operador de asignación (**=**).

1.7.1. Identificadores

El nombre con el que se identifica cada variable se denomina **identificador**. Hay que tener en cuenta que Java distingue entre mayúscula y minúscula (edad es distinto a eDaD) Los identificadores deben seguir las siguientes reglas:

* Comienzan siempre por una letra, un subrayado (\_) o un dólar ($).
* Los siguientes caracteres pueden ser letras, dígitos, subrayado (\_) o dólar ($).
* Se hace distinción entre mayúsculas y minúsculas.
* No hay una longitud máxima para el identificador.

1.8. Tipos primitivos

En un programa de ejecución, las variables se almacenan en la memoria del ordenador. Cada una de ellas necesita un tamaño para guardar sus valores. Un tamaño demasiado pequeño no permite guardar valores grandes o muy precisos, y se corre el riesgo de que el valor que se va a guardar no quepa en el espacio reservado. Utilizar un tamaño excesivamente grande desaprovecha la memoria, haciendo un uso ineficiente de ella.

La solución a este problema no es definir un tamaño para cada variable, sino definir unos tipos de variables, con unos tamaños y rangos de valores conocidos, y que las variables utilizadas en nuestros programas se ciñan a estos tipos.

1.8.1 Variables de tipo primitivo

Al escribir un programa, hemos de indicar a qué tipo pertenece cada variable. Este proceso recibe el nombre de declaración de variables y se hará forzosamente antes de su primer uso.

Todas las declaraciones de variables terminan en punto y coma (;), aunque es posible declarar a la vez varias del mismo tipo, separándolas por comas (,).

1.8.2. Rangos

|  |  |  |  |
| --- | --- | --- | --- |
| Tipo | Uso | Tamaño | Rango |
| Byte | Entero corto | 8 bits | De -128 a 127 |
| Short | Entero | 16 bits | De -32768 a 32767 |
| Int | Entero | 32 bits | De -2147483648 a 2147483647 |
| Long | Entero largo | 64 bits | ± 9 223 372 036 854 775 808 |
| Float | Real precisión sencilla | 32 bits | De -1032 a 1032 |
| Double | Real precisión doble | 64 bits | De -10300 a 10300 |
| Boolean | Lógico | 1 bit | **True** 0 **false** |
| Char | Texto | 16 bits | Cualquier carácter |

1.9. Variables de objeto

Es posible declarar variables cuyo tipo no sea un tipo primitivo, sino una clase. A estas variables se les denomina **variables de objeto** y las utilizaremos para poder aprovechar las herramientas que Java proporciona. Se inicializan por defecto, si es su declaración no se les asigna un valor.

1.10. Constantes

Las constantes son un caso especial de variables, donde, una vez se les asigna su primer valor, este permanece inmutable durante el resto del programa. Cualquier dato que no cambie es candidato a guardarse en una constante.

La declaración de constantes es similar a la de variables, pero añadiendo la palabra reservada **final**.

        final tipo nombreConstante; // Constante "final"

        final byte mayoriaEdad = 18;

        final int numeroAlumnos;

        numeroAlumnos = aulas \* 30;

1.11. Comentarios

- **Comentario multilínea:** Cualquier texto incluido entre los caracteres /\* (apertura de comentario) y \*/ (cierre de comentario) será interpretado como un comentario y puede extenderse a través de varias líneas.

- **Comentario hasta final de la línea:** Todo lo que sigue a los caracteres // hasta el final de la línea se considera un comentario.

- **Comentario de documentación:** Similar al comentario multilínea, con la diferencia de que, para iniciarlo, se utilizar los caracteres /\*\*. Existen herramientas que generan documentación automática a partir de ese tipo de comentarios.

1.12. API de Java

Una de las grandes ventajas de los lenguajes de programación modernos es que disponen de una amplia biblioteca de herramientas que realizan tareas complejas de forma transparente al programador que las utiliza, facilitando su tarea.

A estas herramientas, en Java, se les denomina clases y facilitan multitud de tareas. Algunos ejemplos de las funcionalidades que nos brindan son:

* **Lectura de datos:** Leen información desde el teclado, desde un fichero o desde otros dispositivos.
* **Cálculos complejos:** Realizan operaciones matemáticas como raíces cuadradas, logaritmos, cálculos trigonométricos, etcétera.
* **Manejo de errores:** Controlan la situación cuando se produce un error de algún tipo.
* **Escritura de datos:** Escriben información relevante en dispositivos de almacenamiento, impresoras, monitores, etcétera.

1.12.1. Paquetes

A una agrupación de clases se le denomina **paquete**. Los paquetes pueden agruparse, a su vez, en otros paquetes (Por ejemplo, **Math**).

import java.util.Scanner;

import java.lang.Math;

import java.time.LocalTime;

Cada clase que compone la API puede utilizarse de dos modos:

* **De forma estática:** Se usa directamente el método. Por ejemplo, la case **Math** se utiliza de forma estática.
* **De forma no estática:** Esta manera de utilizar las clases requiere del operador **new**, que se verá en profundidad en unidades posteriores. Un ejemplo de clase que se utiliza de esta forma es **Scanner**, que permite que el usuario introduzca datos en una aplicación.

1.12.2. Salida por consola

Una de las operaciones más básicas que proporciona la API es aquella que permite mostrar un mensaje en el monitos, con idea de aportar información al usuario.

        System.out.println("Hola");

        System.out.print("Hola holita");

        System.out.printf("Que pasó");

**Print** que muestra literalmente el mensaje en el monitor.

**Println** igual que el anterior, pero tras el mensaje, inserta un retorno de carro (nueva línea).

1.12.3. Entrada de datos

Otra operación muy utilizada, consiste en recabar información del usuario a través del teclado. Cuando se hace de forma simple, en modo texto, sin ratón ni interfaz gráfica, se dice que obtenemos datos por consola.

**Scanner** es una clase de la API que se utiliza de forma no estática, es decir, necesita del operador **new**. Y la forma de trabajar con ella es siempre la misma: en primer lugar tendremos que crear un nuevo escáner.

        Scanner teclado = new Scanner(System.in);

        variable = teclado.next"TipoVariable"();

Este fragmento de código se utilizará como una fórmula literal cada vez que necesitemos introducir información por teclado. Según deseemos leer un entero, un real o una cadena de caracteres, solo será necesario modificar el nombre y tipo de la variable que leer y el método.

1.13. Operaciones Básicas

Java dispone de multitud de operadores con los que se pueden crear expresiones utilizando como operandos variables, constantes, números y otras expresiones.

1.13.1. Operador de Asignación

El operador = se usa para modificar el valor de una variable. A la variable se le asigna como valor el resultado de la expresión, si en el momento de la asignación la variable tuviera un valor anterior, este se pierde.

1.13.2. Operadores Aritméticos

El operador – (menos unario) sirve para cambiar el signo de la expresión que le sigue, que estará formada por cualquier secuencia de operaciones aritméticas.

        a = 1;

        b = -a // b vale -1

El operador % devuelve el resto de dividir el primer operando entre el segundo. Por ejemplo **7%3** (se lee 7 módulo de 3) vale 1, ya que al dividir 7 entre 3 el resto (el módulo) es 1.

|  |  |
| --- | --- |
| Símbolo | Descripción |
| + | Suma |
| + | Más unario: positivo |
| - | Resta |
| - | Menor unario: negativo |
| \* | Multiplicación |
| / | División |
| % | Módulo |
| ++ | Incremento en 1 |
| -- | Decremento en 1 |

Los operadores **++** y **--** se utilizan para incrementar o decrementar una variable en 1.

        a ++; // a = a + 1;

        b --; // b = b - 1;

1.13.3 Operadores Relacionales

Son aquellos que producen un resultado lógico o booleano a partir de las comparaciones de expresiones numéricas. El resultado solo permite dos posibles valores: **true** o  **false.**

|  |  |
| --- | --- |
| Símbolo | Descripción |
| == | Igual que |
| != | Distinto que |
| < | Menor que |
| <= | Menor o igual que |
| > | Mayor que |
| >= | Mayor o igual que |

1.13.4 Operadores Lógicos

Permiten operar a partir de expresiones lógicas, formando expresiones más complejas, que devuelven, a su vez, un valor lógico. Existen los operadores:

|  |  |
| --- | --- |
| Símbolo | Descripción |
| && | Operador **and**: Y |
| || | Operador **or**: O |
| ! | Operador **not**: negación |

La expresión formada a partir de otras dos unidas por el operador será **true** cuando ambas expresiones utilizadas se evalúen como ciertas. Y en caso contrario, si alguna o las dos expresiones se evalúan como falsas, la expresión resultante también será falsa.

1.13.5. Operadores Opera y Asigna

Por simplicidad existen otros operadores de asignación llamados opera y asigna, que realizan la operación indicada tomando como operandos el valor de la variable a la izquierda y el valor a la derecha del =. El resultado se asigna a la misma variable utilizada como primer operando.

|  |  |
| --- | --- |
| Símbolo | Descripción |
| += | Igual que |
| -= | Distinto que |
| \*= | Menor que |
| /= | Menos o igual que |
| %= | Mayor que |

Todos tienen el mismo funcionamiento. Utilizan la misma variable para operar con su valor y asignarle el resultado.

        variable += 3; // variable = variable + 3;

        x \*= 2; //X = x \* 2;

1.13.6 Operador Ternario

Este operador devuelve un valor que se selecciona de entre dos posible. La selección dependerá de la evaluación de una expresión relacional o lógica que, como hemos visto, piuede tomar dos valores: **verdadero** o **falso**.

//Ejemplo ternario

Scanner teclado = new Scanner(System.in);

int a = teclado.nextInt();

int b = teclado.nextInt();

int maximo = a > b ? a : b; // A mayor que B o contrario

System.out.println("El máximo es: " + maximo);

1.14. Conversión de tipos

Todas las variables en Java tienen asociado un tipo. Cuando asignamos un valor a una variable, ambos deben ser del mismo tipo. A una variable tipo **int** se le puede asignar un valor **int** y a una variable **double** se le puede asignar un valor **double**.

        int a = 2;

        double x = 2.3;

Java permite la asignación de variables de un tipo a otra. Para pasar de **int** a **double** Java convierte de forma automática el valor entero (3, por ejemplo) a valor **double** (3.0) antes de asignarlo a la variable X. Esto es posible porque la variable double es de mayor tamaño que el valor int.

Muy distinto es que intentemos asignar un valor double a una variable int, que no tiene sitio suficiente para guardarlo. En este caso, java no hará ninguna conversión automática. Se limitará a darnos un error de compilación, aunque a veces es interesante guardar la parte entera de un número con decimales en una variable entera. Para ellos deberemos colocar un cast o molde delante del valor que queremos asignar.

    int a = (int) 2.6; //(int) indica el tipo al que se convertirá el valor

# Condicionales

Un programa no tiene por qué ejecutar siempre la misma secuencia de instrucciones. Puede darse el caso de que, dependiendo del valor de alguna expresión o de alguna condición, interese ejecutar o evitar un conjunto de sentencias (**if**, **if-else** y **switch**).

2.1. Expresiones Lógicas

Una condición no es más que una expresión relacional o lógica. El valor de una condición siempre es de tipo booleano: verdadero o falso.

2.1.1. Operadores Relacionales

Los operadores relacionales son aquellos que comparan expresiones numéricas para generar valores booleanos. Solo pueden tener dos valores: verdadero o falso.

|  |  |
| --- | --- |
| Símbolo | Descripción |
| == | Igual que |
| != | Distinto que |
| < | Menor que |
| <= | Menor o igual que |
| > | Mayor que |
| >= | Mayor o igual que |

2.1.2. Operadores Lógicos

Los operadores lógicos permiten construir condiciones más complejas, ya que estos operan y generan valores booleanos.

|  |  |
| --- | --- |
| Símbolo | Descripción |
| && | Operador **and**: Y |
| || | Operador **or**: O |
| ! | Operador **not**: negación |

* **Operador &&:** Será cierto si ambos operandos son ciertos.
* **Operador ||:** Es cierto si cualquiera (uno o ambos) de los operandos es cierto.
* **Operador !:** Niega ---cambia--- el valor al que se aplica, convirtiendo **true** en **false** y viceversa.

2.2. Condicional Simple: if

La sentencia **if** proporciona un control sobre un conjunto de instrucciones que pueden ejecutarse o no, dependiendo de la evaluación de una condición. Los dos posibles valores (true o false) de esta determinan si el bloque de instrucciones (**{}**) de if se ejecuta o no.

        a = 3;

        if (a + 1 < 10) {

            a = 0;

            System.out.println("Hola");

        }

        System.out.println("El valor de a es " + a);

2.3 Condicional doble: if-else

Existe otra versión de la sentencia **if**, denominada **if-else**, donde se especifican dos bloques de instrucciones. El primero (true) se ejecutará cuando la condición resulte verdadera y el segundo (false), cuando la condición resulte falsa. Ambos bloques son mutuamente excluyentes, en cada ejecución de la instrucción **if-else** solo se ejecutará uno de ellos.

        if (a > 0) {

            System.out.println("Valor positivo");

        } else {

            System.out.println("Valor negativo o cero");

        }

El operador ternario permite seleccionar un valor de entre dos posible, dependiendo de la evaluación de una condición.

        variable = condición ? valor1 : valor2

        //Es lo mismo que usar

        if (condición) {

            variable = valor1;

        } else {

            variable = valor2;

        }

Un ejemplo sería calcular el máximo de dos números introducidos:

        Scanner teclado = new Scanner(System.in);

        int a = teclado.nextInt();

        int b = teclado.nextInt();

        int maximo = a > b ? a : b; // A mayor que B o contrario

        System.out.println("El máximo es: " + maximo);

2.3.2. Anidación de condicionales

Cuando debamos realizar múltiples comprobaciones, podemos anidar tantos **if** o **if-else** como necesitemos, unos dentro de otros. La anidación de condicionales hace que las comprobaciones sean excluyentes, y resulta más eficiente.

        if (a - 2 == 1) {

            System.out.println("Hola ");

        } else {

            if (a - 2 == 5) {

                System.out.println ("Me ");

            } else {

                if (a - 2 == 8) {

                    System.out.println("Alegro ");

                } else {

                    if (a - 2 == 9) {

                        System.out.println("De ");

                    } else {

                        if (a - 2 == 11) {

                            System.out.println("Conocerte.");

                        } else {

                            System.out.println("Sin coincidencia");

                        }

                    }

                }

            }

        }

Cuando el bloque de instrucciones del condicional está formado por una única sentencia, no es necesario utilizar llaves (**{}**), aunque son recomendables. Cuando hay muchos casos alternativos es habitual eliminar las llaves de los bloques **else**, consiguiendo un código más compacto.

        if (a - 2 == 1) {

            System.out.println("Hola ");

        } else if (a - 2 == 5) {

            System.out.println("Me ");

        } else if (a - 2 == 8) {

            System.out.println("Alegro ");

        } else if (a - 2 == 9) {

            System.out.println("De ");

        } else if (a - 2 == 11) {

            System.out.println("Conocerte.");

        } else {

            System.out.println("Sin coincidencia.");

        }

2.4. Condicional Múltiple: Switch

El hecho de utilizar muchos **if** o **if-else** anidados suele producir un código poco legible y difícil de mantener. Para estos casos Java dispone de la sentencia **switch**. La evaluación de expresión debe dar un resultado entero, convertible en entero o un valor de tipo **String**. La cláusula **default** es opcional.

* Evalúa la expresión y obtiene su valor.
* Compara el valor obtenido con cada valor de las cláusulas **case**. Se puede utilizar en un mismo **case** varios valores separados por coma.
* En el momento en que coincide con alguno de ellos, ejecuta el conjunto de instrucciones de esa cláusula **case** y de todas las siguientes.
* Si no existe coincidencia alguna, se ejecuta el conjunto de instrucciones de la cláusula **default**, siempre y cuando esté presente.
* Las ejecuciones continuaran hasta encontrarse con un **break**.

        switch (nota) {

            case 0,1,2,3,4: {//Bloque formado por dos instrucciones: entre llaves

                System.out.println("Suspenso.");

                System.out.println("Ánimo...");

            }

            break;

            case 5: //Bloque de una única instrucción: podemos obviar las llaves

                System.out.println("Suficiente.");

            break;

            case 6:

                System.out.println("Bien.");

            break;

            case 7,8:

                System.out.println("Notable.");

            break;

            case 9,10: {

                System.out.println("Sobresaliente.");

                System.out.println("Enhorabuena.");

            }

            break;

            default:

                System.out.println("Nota incorrecta.");

        }

# Bucles

Un bucle es un tipo de estructura que contiene un bloque de instrucciones que se ejecuta repetidas veces; cada ejecución o repetición del bucle se llama iteración.

El uso de bucles simplifica la escritura de programas, minimizando el código duplicado.

* 1. Bucles controlados por condición

El control del número de iteraciones se lleva a cabo mediante una condición.

* + 1. While

Al igual que **if**, **while** depende de la evaluación de una condición. Decide si realizar una nueva iteración basándose en el valor de la condición.

1. Se evalúa **condición**.
2. Si la evaluación resulta **true**, se ejecuta el bloque de instrucciones.
3. Tras ejecutarse el bloque de instrucciones, se vuelve al primer punto.
4. Si la condición es **false**, terminamos la ejecución del bucle.
5. int i = 1; *//Valor inicial*
7. while (i <=3) { */\*El bucle iterará mientras i sea menos*
8. *o igual que 3\*/*
9. System.out.println(i); *//Mostramos i*
10. i++; */\*Incrementamos i para la siguiente vuelta*
11. *del bucle\*/*
12. }

Un bucle **while** puede realizar cualquier número de iteraciones, desde cero, cuando la primera evaluación de la condición resuelta falsa, hasta infinitas, en el caso de que la condición sea siempre cierta. Esto es lo que se conoce como **bucle infinito**.

* + 1. Do-while

El bucle **do-while** es similar al **while**, con la diferencia de que primero se ejecuta el bloque de instrucciones y después se evalúa la condición para decidir si se realiza una nueva iteración.

        int i = 1;

        do {

            System.out.println(i);

            i++;

        } while (i <=10);

Es el único bucle que termina en punto y coma (;). Mientras **while** se puede ejecutar de 0 a infinitas veces, el **do-while** lo hace de 1 a infinitas veces.

* 1. Bucles controlados por: for

El bucle **for** permite controlar el número de iteraciones mediante una variable (que suele recibir el nombre de contador).

Consta de 3 partes:

* **Inicialización:** Es una lista de instrucciones, separadas por comas, donde generalmente se inicializan las variables que van a controlar el bucle. Se ejecutan una sola vez antes de la primera iteración.
* **Condición:** Es una expresión booleana que controla las iteraciones del bucle. Se evalúa antes de cada iteración; el bloque de instrucciones se ejecutará solo cuando el resultado sea **true**.
* **Incremento:** Es una lista de instrucciones, separadas por comas, donde se suelen modificar las variables que controlan la condición. Se ejecuta al final de la iteración.

Aunque **for** está controlado por una condición que, en principio, puede ser cualquier expresión booleana, la posibilidad de configurar la inicialización y el incremento de las variables que controlan el bucle permite determinar de antemano el número de iteraciones.

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

            System.out.println("La i vale " + i);

        }

* 1. Salidas anticipadas

Dependiendo de la lógica que implementemos en el programa, puede ser interesante terminar un bucle antes de tiempo y no esperar a que termine por su condición (realizando las iteraciones). Para poder hacer esto disponemos de:

* **Break:** Interrumpe completamente la ejecución del bucle.
* **Continue:** Detiene la iteración actual y continúa con la siguiente.

Cualquier programa puede escribirse sin **break** ni **continue**; se recomienda evitarlos, ya que rompen la secuencia natural de las instrucciones.

        i = 1;

        while (i <= 10) {

            System.out.println("La i vale" + );

            if (i == 2) {

                break; */\*El bucle esta para valor 10*

*pero con break lo cortamos en 2 \*/*

            }

            i++;

        }

        i = 0;

        while (i < 10) {

            i++;

            if (i % 2 == 0) { *//Si i es par*

                continue;

            }

            System.out.println("La i vale " + i);

        }

* 1. Bucles anidados

En el uso de los bucles es muy frecuente la anidación, que consiste en incluir un bucle dentro de otro.

Al hacer esto se multiplica el número de veces que se ejecuta el bloque de instrucciones de los bucles internos. Los bucles anidados pueden encontrarse relacionados cuando las variables de los bucles más externos intervienen en el control de la iteración de un bucle interno; o independientes, cuando no existe relación alguna entre ellos.

* + 1. Bucles independientes

Cuando los bucles anidados no dependen, en absoluto, unos de otros para determinar el número de iteraciones, se denominan **bucles anidados independientes**.

        for (i = 1; i <= 4; i++) {

            for (j = 1; j <= 3; j++) { *//j va independiente de i*

                System.out.println("Ejecutando...");

            }

        }

3.4.2 Bucles dependientes

Puede darse el caso de que el número de iteraciones de un bucle interno no sea independiente de la ejecución de los bucles exteriores, y dependa de sus variables de control. Decimos entonces que son **bucles anidados dependientes.**

        for (i = 1; i <=3; i++) {

            System.out.println("Bucle externo, i=" + i);

            j = 1;

            while (j <= i) { *//j depende del valor de i*

                System.out.println("...Bucle interno, j=" + j);

                j++;

            }

        }

# Funciones / Métodos

4.1 Conceptos Básicos

La solución para cuando necesitamos la misma funcionalidad en distintos lugares de nuestro código no es más que etiquetar con un nombre de fragmento de código y sustituir en el programa dicho fragmento, en todos los lugares donde aparezca, por el nombre que le hemos asignado.

En general, la sintaxis para definir una función es:

    static tipo nombreFuncion() {

        cuerpo de la función

    }

Definimos algunos conceptos necesarios para seguir trabajando con funciones:

* **Llamada a la función:** Es el nombre de la función, seguido de (). Se convierte en una nueva instrucción que podemos utilizar para invocarla.
* **Prototipo de la función:** Es la declaración de la función, donde se especifica su nombre, el tipo que devuelve y, entre paréntesis, los parámetros de entrada que utiliza “static tipo nombreFuncion()”.
* **Cuerpo de la función:** Es el bloque de código que ejecuta la función cada vez que se invoca y que aparece entre llaves después del prototipo (contenido de la función).
* **Definición de una función:** Está formada por le prototipo más el cuerpo de la función (función al completo).

Con esto evitamos:

* La duplicidad del código: ya que el código se escribe una única vez.
* La dificultad en el mantenimiento, en el caso de realizar modificaciones, solo se realizarán en la función, no en donde se aplica.

El comportamiento de una llamada a una función consiste en:

1. Las instrucciones del programa principal se ejecutan hasta que encuentran la llamada a la función.
2. La ejecución salta a la definición de la función.
3. Se ejecuta el cuerpo de la función.
4. Cuando la ejecución del cuerpo termina, retornamos al punto del programa desde donde se invocó la función.
5. El programa continúa su ejecución.

En la programación estructurada se llaman **funciones**, y en la programación orientada a objetos, se denominan **métodos**.

4.2 Ámbito de las variables

En el cuerpo de la función podemos declarar variables, que se conocen como **variables locales**. Su ámbito es la propia función donde se declaran, no pudiéndose utilizar fuera de ella. Se puede declarar variables con el mismo nombre en **diferentes funciones**.

4.3 Paso de información a una función

Cuando una función necesita conocer información externa para poder llevar a cabo su tarea. Se transmite esa información mediante una llamada a la función, indicando las variables que tienen esa información, por ejemplo “nombreFunción(2)”. Es lo que denominamos un **parámetro de entrada**, no es más que una variable local a la que se le asignan valores en cada llamada.

4.3.1 Valores en la llamada

En la llamada a una función se pueden pasar valores que provienen de literales, expresiones o variables.

    variosSaludos(2); *//llamada con un literal*

    int n = 3;

    variosSaludos(2+n); *//llamada con una expresión*

4.3.2 Parámetros de entrada

Una función puede definirse para recibir tantos datos como necesita. Cada dato utilizado en la llamada a una función será asignado a un parámetro de entrada, especificado en la definición de la función con de la siguiente manera:

    tipo nombreFuncion(tipo1 parametro1, tipo2 parametro2...) {

        cuerpo de la función

    }

El primer parámetro de entrada lo hemos llamado parametro1 y se le puede asignar un valor del tipo tipo1, lo mismo ocurre con el resto de parámetros. El número de parámetros definidos en la función determina el número de valores que hay que utilizar en cada llamada.

En Java, los parámetros toman su valor como una copia del valor de la expresión o variable utilizada en la llamada; este mecanismo de paso de parámetros se denomina **paso de parámetros por valor** o **por copia**.

4.4 Valor devuelto por una función

Es posible realizar paso de información de la función a nuestro programa principal, realizando una llamada desde el cuerpo de la función hacia el código don de se hace la llamada principal. Con esto conseguimos que la llamada a una función se convierta en un valor cualquiera. Este puede ser utilizado desde el lugar donde se invoca.

    int a = suma(2, 3);

    int b = suma(7, 1) \* 5;

Para que la función devuelva el la información, disponemos de la instrucción **return**, que finaliza la ejecución de la función y devuelve el valor indicado.

    tipo nombreFunción(parámetros) {

        ...

        return (valor);

    }

Debe existir una concordancia entre el tipo declarado en la función y el tipo del valor devuelto con **return**. Nada impide utilizar varios *return* en una misma función, pero es una práctica desaconsejable, ya que una función debe tener un único punto de entrada y de salida; el uso de varios *return* rompe esta norma.

4.5 Sobrecarga de funciones

Java permite que dos o más funciones compartan el mismo identificador en un mismo programa. Esto es lo que se conoce como **sobrecarga de funciones**. La forma de distinguir entre las distintas funciones sobrecargadas es mediante sus listas de parámetros, que deben ser distintas, ya sean en número o en tipo.

Las funciones sobrecargadas pueden devolver tipos distintos, aunque estos no sirven para distinguir una función sobrecarga de otra.

*// Función sobrecargada*

    static int suma(int a, int b) {

        int suma;

        suma = a + b;

        return (suma);

    }

*// Función sobrecargada*

    static double suma(int a, double pesoA, int b, double pesoB) {

        double suma;

        suma = a \* pesoA / (pesoA + pesoB) + b;

        return (suma);

    }

4.6 Recursividad

Una función puede ser invocada desde cualquier lugar: desde el programa principal, desde otra función e incluso desde dentro de su propio cuerpo de instrucciones. En este último caso, cuando una función se invoca a sí misma, diremos que es una **función recursiva**.

    static int funcionRecursiva() {

        if (caso base) {

            resultado = valorBase;

        } else {

            resultado = funcionRecursiva(nuevosDatos); *//Llamada recursiva*

            ...

        }

        return (resultado);

    }