



Programación en Java I: conceptos básicos

1. Introducción: que es Java

1.1. Historia de Java

El lenguaje Java surge como respuesta a la necesidad de Sun Microsystems de crear un lenguaje que sirviera para programar dispositivos electrónicos tales como televisores, vídeos, etc...

Las primeras versiones de este proyecto, llamado originalmente Green –y luego Oak- no fueron particularmente exitosas.

Sin embargo la aparición de la World Wide Web permitió reimplementar Java como un lenguaje orientado a las aplicaciones de Internet, donde si tuvo un gran éxito al aparecer programas Java como código insertado en los navegadores web para ejecutar aplicaciones dinámicas.

Sin embargo Java no solo es un lenguaje para Internet sino que también es un lenguaje de propósito general de gran aceptación, gracias a su sintaxis, heredada de C++, de quien ha depurado sus características más complejas, como la gestión de memoria dinámica –punteros- que en Java se hace de manera automática.

Java dispone de versiones de lenguaje aptas para la programación de propósito general, J2SE, para móviles, J2ME y para entornos empresariales, J2EE.

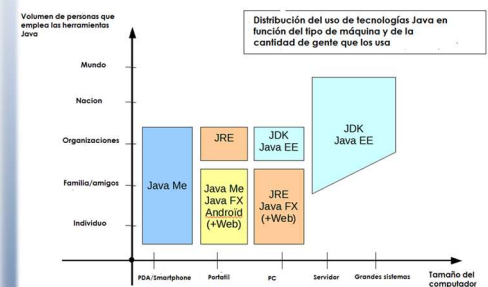
En este tema y el siguiente estudiaremos J2SE y nos iniciaremos en J2ME.

1.2. Características de Java

Java se define como un lenguaje interpretado. Esto quiere decir que cuando se compila un programa Java, se genera un archivo llamado *bytecode*, que no se puede ejecutar directamente por la CPU de los equipos. Es necesario un intérprete, llamado *Máquina Virtual Java*, que se instala en cualquier equipo e interpreta los bytecodes, ejecutándolos.



James Gosling, creador de Java.



La gráfica de la imagen anterior muestra la distribución del uso de las diferentes herramientas Java en función del tipo de dispositivo y la entidad de la agrupación humana.

De acuerdo con sus creadores, Java posee las siguientes características:

Sencillez: Java fue diseñado usando la sintaxis de C/C++ con lo que la curva de aprendizaje para el programador es menor si este viene de trabajar en ese lenguaje. Por otro lado ha eliminado los aspectos más complejos de C/C++ en el manejo de la memoria dinámica.

Orientación a objetos: Java ha sido diseñado según el paradigma POO desde un principio, a diferencia de C++, que incluyó la orientación a objetos como una extensión del C original, basado en el paradigma de la programación estructurada.

Distribuido: Java permite generar aplicaciones capaces de trabajar de manera coordinada desde varias máquinas en la Red, articulando lo que se llama computación distribuida.

Robustez: Para evitar que los programas fallen en tiempo de ejecución, Java incluye las excepciones, que son instrucciones que se ejecutan en áreas críticas del programa, en caso de producirse alguna interrupción que pueda poner en riesgo el correcto funcionamiento del programa.

Seguridad: Las aplicaciones Java han sido diseñadas para evitar la acción de virus y de esta manera no generar trastornos al usuario.

Neutralidad respecto a la plataforma: La utilización de la Máquina Virtual de Java permite que los programas implementados en este lenguaje puedan adaptarse a cualquier arquitectura de computador, puesto que es la máquina virtual la que evoluciona con las arquitecturas a través de diferentes versiones.

Portabilidad: Java no solo es portable gracias a la máquina virtual que se instala en cualquier equipo, sino que fue diseñado de tal manera que los tipos de datos –enteros, reales, etc...- fueron diseñados con unos tamaños fijos independientes del tipo de máquina en el que se ejecuten.

Esto no ocurría en C/C++, donde el tamaño de los tipos de datos dependía de la máquina sobre la que se ejecutaba el programa.

Alto rendimiento: Las nuevas máquinas virtuales son tan rápidas interpretando que casi no hay diferencia con los programas ejecutados directamente sobre la CPU

Multihilo: Java está diseñado para que cada programa sea capaz de ejecutar varias tareas simultáneamente. Para ello tiene integrada la programación multihilo. Esta es especialmente útil en programación distribuida y aplicaciones de Internet, donde un servidor puede tener que realizar varias tareas a la vez.

Dinámico: Los programas Java pueden enlazar nuevas librerías y clases directamente en tiempo de ejecución, lo que lo hace más adaptable que C/C++.

2. El IDE BlueJ

Existen numerosos entornos de desarrollo integrado –IDE– para programar en Java. El más conocido es JDK, que ofrece gratuitamente Sun Microsystems. Otro entorno ampliamente usado por los desarrolladores es Eclipse, una herramienta de software libre que no solo ofrece programar en Java, sino que se extiende a otros lenguajes como C/C++ o Python.

Sin embargo aquí emplearemos BlueJ. Este es un IDE que se ha desarrollado para apoyar la enseñanza-aprendizaje de la POO, y por ello es usado en entornos docentes.

Tiene como ventaja la posibilidad de ejecutar los diferentes objetos que se diseñan a medida que están listos y ofrece un módulo de ingeniería del software muy parecido a UML donde se pueden ver las relaciones entre las distintas clases que conforman el programa –en ingeniería del software se le llama a esto diagrama de clases–.

Nos permite trabajar con applets ,archivos .jar –veremos posteriormente que es eso–, y J2ME –Java para dispositivos móviles–.

La instalación de BlueJ constituirá la parte de prácticas de este tema –junto con la realización de ejercicios sobre dicho IDE, por supuesto–.



Icono de BlueJ, un entorno de desarrollo Java apto para tareas de enseñanza-aprendizaje.



Logo del IDE Eclipse, una herramienta que permite programar en múltiples lenguajes.



Logo de Java Development Kit –JDK–

Nombre del programa

El nombre del archivo fuente ha de coincidir con el nombre de la clase principal, que es la que contiene el método `main()`

Import //importa clases desde otros paquetes

Public static void main() // metodo desde el que empieza a ejecutarse el programa

Método principal main

```
Public static void main(string[] ar)
{
    //declaracion de variables
    // sentencias
}
```

Definiciones de otros métodos dentro de la clase

```
static tipo func1()
{
}
```

```
static tipo func2()
{
}
```

En Java tenemos una serie de palabras reservadas, que forman parte de la sintaxis del lenguaje, y no se pueden usar para nombrar variables, por ejemplo. Estas son: `abstract, assert, boolean, break, byte, case, catch, char, class, continue, default, do, double, else, enum, extends, final, finally, float, for, if, implements, import, instanceof, int, interface, long, native, new, package, private, protected, public, return, short, static, strictfp, super, switch, synchronized, this, throw, throws, transient, try, void, volatile, while`

3. Fases de creación de un programa en Java

Las fases de creación de un programa en Java son las siguientes:

- Crear una carpeta de proyecto en la que se guarden todos los archivos que van a formar parte de la aplicación Java.
- Empleando un editor escribir el programa Java, generando las clases, y dentro de ellas los métodos y atributos necesarios.
- Convertir este código fuente en bytecode al compilarlo, depurando para detectar errores y en ese caso corrigiendo en el código fuente las instrucciones oportunas.
- Pasar ese bytecode a la máquina virtual de Java, donde es ejecutado instrucción a instrucción –no olvidemos que es un intérprete–

4. Estructura general de un programa en Java

4.1. Elementos básicos

Un programa en Java es una colección de clases relacionadas entre sí mediante mensajes. Entre todas estas clases debe haber una que actúe como la clase principal y que llevará el nombre del programa.

Dentro de esta clase habrá un método `main()` que actuará como método principal, y después una serie de métodos que representarán el comportamiento de la clase.

De una manera general, un programa Java tendrá los siguientes elementos

- Nombre del paquete, que se coloca en la primera línea del programa, y va asociado al nombre del proyecto que se crea en Eclipse.
- Declaración `import`, para importar clases de otros paquetes

- Declaraciones de clases, entre ellas la clase principal
- Método `main()` dentro de la clase principal
- Otros métodos definidos por el usuario en la clase principal
- Comentarios del programa

En el recuadro de la derecha podemos ver un programa real en Java.

4.2. Importación de clases

En Java todas las clases están agrupadas en paquetes, semejantes en concepto a las librerías de C `-stdio.h`, `conio.h`, etc...-.

Para añadir estas clases al programa Java se emplea la declaración `import` -en C era `include`-.

La declaración `import` toma el siguiente aspecto

`import nombrepaquete.nombreclase`

Donde `nombreclase` es una clase concreta del paquete. Si queremos añadir todas las clases empleamos `*` en vez de `nombreclase`.

De la misma manera que en C el programador podía generar TADs o paquetes propios que luego podía llamar al principio de la ejecución de un programa, aquí también los programadores pueden crear paquetes y emplear `import` para añadirlos a los programas que creen.

4.3. Definición de clases

Como ya se ha comentado un programa en Java es una colección de clases. Para crear cada clase se sigue la siguiente estructura

- Una palabra clave que indica el acceso

```
public class prueba
{
    public static void main() throws IOException
    {
        int x,y;
        int suma, resta, multiplicacion;
        String cadena;

        InputStreamReader flujo;
        BufferedReader teclado;
        flujo= new InputStreamReader(System.in);
        teclado= new BufferedReader(flujo);

        System.out.print("Introduzca el valor del primer sumando: ");
        cadena=teclado.readLine();
        x=Integer.parseInt(cadena);
        System.out.println();

        System.out.print("Introduzca el valor del segundo sumando: ");
        cadena=teclado.readLine();
        y=Integer.parseInt(cadena);
        System.out.println();

        suma=operacionsuma(x,y);
        System.out.println("El resultado de la suma es " + suma);

    }
}

// instance variables - replace the example below with your own
static int operacionsuma(int a, int b)
{
    return a+b;
}
```

Ejemplo de un programa en lenguaje Java.

```
* @author (your name)
* @version (a version number or a date)
*/
import java.io.*; //archivo de clases de entrada/salida
public class prueba
{
    public static void main() throws IOException
    {
        int x,y;
        int suma, resta, multiplicacion;
        String cadena;

        InputStreamReader flujo;
        BufferedReader teclado;
        flujo= new InputStreamReader(System.in);
```

Ejemplo de uso de `import` en un programa Java.

```
public class Hola
{
    /** este es el método principal
    */
    public static void main (String[]
args)
    {
        // No hay declaraciones
        System.out.println ("Hola, Que
tal estas?");
    }
}
```

Ejemplo de clase definida por un usuario. Puede verse el método `main()` con la sintaxis indicada en el texto.


```
public class Resistencias
{
    private double r1, r2, r3; //Kohms

    /** Constructor*/
    public Resistencias(double res1,
        double res2, double res3)
    {
        r1=res1;
        r2=res2;
        r3=res3;
    }

    /** Calcula la resistencia
    equivalente a 3 resistencias en
    paralelo */
    public double calculaResEquiv()
    { // Kohms
        return 1/((1/r1)+(1/r2)+(1/r3));
    }
}
```

Ejemplo de clase en Java con un método constructor. En el tema anterior hemos visto que el método constructor es llamado para instanciar un objeto de una clase determinada.

Podemos verlo en el siguiente ejemplo

```
public class
ResistenciasEnParalelo
{
    public static void main(String[]
args)
    {
        double r1=3.5; // Kohms
        double r2=5.6; // Kohms
        double r3=8.3; // Kohms
        double reff; // Kohms
        Resistencias res=new
Resistencias(r1,r2,r3);

        reff=res.calculaResEquiv();
        System.out.println
("Resist. en paralelo:
"+r1+" "+r2+" "+r3);

        System.out.println("Resistencia
efectiva = "+reff);
    }
}
```

- El identificativo class
- El nombre de la clase
- El método constructor, si no es una clase principal – tal y como se explicó en el tema 12.
- El método main(), si es una clase principal
- Los métodos y atributos de la clase.

Se ha explicado en el tema anterior, pero no está de más recordar que las declaraciones de métodos y atributos en una clase son comunes a todos los miembros de la misma.

4.4. El método main

El método *main()* es el equivalente al *main()* de C, esto es, el punto de inicio del programa. El método *main()* va dentro de la clase principal, esto es la que tiene el nombre del programa Java.

Puede llevar parámetros de entrada tipo *String*, de cara a que al ejecutarse, se puedan introducir datos en forma de cadenas de caracteres.

Se define siempre como un método público, estático y que no devuelve nada, con la sintaxis siguiente.

```
public static void main(String [] ar)
```

Una norma de buena programación establece que el método *main()* solo debe de contener llamadas a otros métodos u objetos y nunca debe contener grandes cantidades de código, de cara a respetar uno de los principios básicos de la programación, la modularidad.

4.5. Métodos definidos por el usuario

En el tema 12 explicábamos que los métodos definían el comportamiento de los objetos que los incluían. Para poder

especificar las diferentes tareas que van a ejecutar los objetos debemos generar métodos, de tal manera que cada uno de ellos realice una tarea diferente –conservar la modularidad y reducir la cohesión es fundamental para programar correctamente.

Los métodos son invocados dentro de su propio objeto empleando su nombre y los parámetros de entrada. Si se invocan desde otro objeto, debe colocarse delante el nombre del objeto de origen separado por un punto del nombre del método y los parámetros.

La sintaxis de definición de un método es la siguiente

tipo_devuelto nombre (param1, param2,)

Donde *tipo_devuelto* es el tipo de dato que devuelve el método y param1, param2, etc... son los parámetros de entrada.

5. Tipos de datos

Un tipo de datos es un conjunto de datos que puede tomar una variable. En Java, como en los demás lenguajes modernos disponemos de una serie básica de tipos de datos llamados predefinidos.

Cada tipo de datos precisa una determinada cantidad de espacio en memoria.

Aparte de los tipos básicos, Java puede definir sus propios tipos de datos empleando la construcción *class*.

En el cuadro de la derecha se pueden ver los tipos de datos usados en Java, con el tamaño de memoria que ocupan y el rango de valores que pueden tomar.

6. Constantes

En Java, los elementos con nombre cuyo valor no va a cambiar se denominan constantes. Estas pueden ser constantes literales o constantes por declaración. En el primer caso se escriben como tales en el programa, en el

```
public class esfera
{
    /* clase esfera */
    public static void main (String[]
args)
    {
        double x;
        x=volumen();
        System.out.println ("Hola, Que
tal estas?");
    }
    double volumen()
    {
        double v;
        int r;
        double pi=3,1416;
        r=3;
        v=(4/3)*r*r*pi;
        return v;
    }
}
```

Ejemplo de clase donde podemos ver definido un método volumen que calcula el volumen de una esfera.

Pueden también observarse comentarios –entre /* y */ –, que es texto que no forma parte del código, sino que sirve para poder comprender mejor el programa. Es el primer nivel de documentación del software.

Tipo datos	Ejemplo	N_byte	Rango
Char	'a'	2	Unicode
Byte	-15	1	-2 ⁷ -2 ⁷ -1
Short	1024	2	-2 ¹⁶ -2 ¹⁶ -1
Int	59874	4	-2 ³² -2 ³² -1
Long	324543	8	-2 ⁶⁴ -2 ⁶⁴ -1
Float	10.5f	4	3.4*10 ⁻³⁸ - 3.4 *10 ³⁸
double	0.003	8	1.7*10 ⁻³⁰⁸ - 1.7*10 ³⁰⁸
boolean	false	1 bit	False, true

Java, por sus características, evita el problema del desbordamiento de memoria que se produce cuando un dato ocupa más espacio del asignado, con la consiguiente pérdida de información, o acceso a posiciones no reservadas para el código.

La Real Academia de la Lengua Española define variable como la *magnitud que puede tener un valor cualquiera de los comprendidos en un conjunto*

```
public class Notas
{
    //Atributos de la clase
    private int nota1, nota2, nota3;

    /** Pone los valores de las tres
    notas */
    public void ponNotas (int n1, int
    n2, int n3)
    {
        nota1=n1;
        nota2=n2;
        nota3=n3;
    }

    /** Calcula la media real */
    public double media()
    {
        return (nota1+nota2+nota3)/3.0;
    }

    /** Calcula la media entera */
    public int mediaEntera()
    {
        return (nota1+nota2+nota3)/3;
    }
}
```

Ejemplo de código donde podemos ver una definición de variables de clase *private*, que solo serán accesible a través de los métodos que han sido definidos dentro de la clase.

Una regla de estilo en programación es escribir las palabras que forman un identificador poniendo en mayúscula la primera letra de cada palabra

segundo se declaran como variables que no se pueden modificar. La sintaxis de estas últimas es:

static final tipo nombre = valor

static indica que el elemento se define a nivel de clase, mientras que *final* impide que se pueda cambiar su valor a lo largo de la ejecución del programa.

7. Variables locales y globales

En Java, los datos se pueden almacenar en campos que denominamos variables, y que ocupan un espacio en memoria que se reserva al crear esa variable.

Para crear esa variable hay que definirla de acuerdo con la siguiente sintaxis

tipo nombre1, nombre2,.....

tipo nombre=valor;

Donde *tipo* es el tipo de datos al que pertenece la variable y *valor* el contenido que se puede asignar a la variable, en caso de que la definición y la asignación sean simultáneas.

El nombre o identificador debe cumplir una serie de condiciones, que son las mismas que han de cumplir los nombres de métodos o de objetos. Esto es, han de ser únicos, comenzar siempre por una letra, un subrayado o un dólar, donde los siguientes caracteres pueden ser letras, dígitos, subrayados o dólares, recordando que se distingue entre mayúsculas y minúsculas y sin longitud máxima.

Las variables se pueden definir dentro de una clase, como atributo de la misma, o dentro de un método, como variable propia de este.

Las variables pueden ser locales o globales/de clase. En el primer caso solo existen cuando se ejecuta el método, puede haber varias con el mismo nombre siempre que estén en métodos distintos, y no se puede acceder a ellas desde fuera del método, para garantizar la encapsulación.

Las variables de clase se declaran fuera de los métodos, inmediatamente después de la definición de la clase y pueden ser públicos *-public-* o privados *-private-*, si se quiere que sean accesibles directamente desde otras clases o solo a través de métodos de la propia clase respectivamente. Existe también el modificador *protected*, que permite que una variable sea accesible por las clases derivadas de aquella en la que se definió así como por las clases que pertenezcan al mismo paquete de datos o archivo de código fuente.

8. Entrada y salida de datos

La entrada/salida de datos en Java es controlada por una clase, llamada *System*. *System* posee dos objetos que gestionan las operaciones de entrada y salida de datos respectivamente: *System.in* y *System.out*.

La salida de datos es particularmente sencilla, dado que ya hay varios métodos en *System.out* listos para ser empleados, en concreto los tres siguientes:

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

`System.out.print(...);`

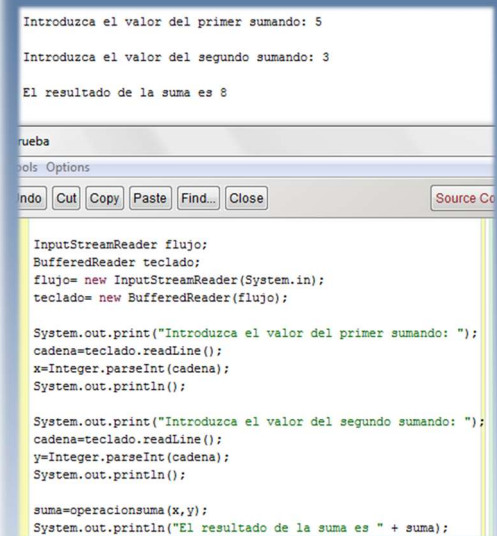
`System.out.flush();`

El primero pasa a la pantalla una cadena de caracteres, mientras que el segundo hace lo mismo, pero además pasa un carácter de fin de línea.

El método `.flush` transfiere todas las cadenas almacenadas en un buffer de salida hacia la pantalla.

Por su parte la entrada de datos se puede realizar de varias formas, de las que veremos dos:

- La primera requiere la creación de un objeto de la clase *InputStreamReader*, que se usa a su vez como argumento para instanciar otro objeto, este de la clase *BufferedReader*, que permite captar cadenas de



```

Introduzca el valor del primer sumando: 5
Introduzca el valor del segundo sumando: 3
El resultado de la suma es 8

rueba
Tools Options
Find... Close Source Code
InputStreamReader flujo;
BufferedReader teclado;
flujo= new InputStreamReader(System.in);
teclado= new BufferedReader(flujo);

System.out.print("Introduzca el valor del primer sumando: ");
cadena=teclado.readLine();
x=Integer.parseInt(cadena);
System.out.println();

System.out.print("Introduzca el valor del segundo sumando: ");
cadena=teclado.readLine();
y=Integer.parseInt(cadena);
System.out.println();

suma=operacionsuma(x,y);
System.out.println("El resultado de la suma es " + suma);
    
```

En este ejemplo podemos ver los métodos `System.out.print` y `System.out.println` en funcionamiento, así como el método de entrada de datos `readLine`, y el método `parseInt` de transformación de una cadena de caracteres en entero.

APIs de Java

Los lenguajes de programación modernos como Java disponen de bibliotecas de herramientas que constan de clases que resuelven tareas como lectura y escritura de datos, cálculos complejos, manejo de errores y otras. Dichas bibliotecas se denominan API, o Interfaces de Programación de Aplicaciones. Dichas clases se agrupan en paquetes y se identifican indicando la estructura de paquetes seguida del nombre de la clase, por ejemplo

```
Java.lang.Math
```

es la denominación completa de la biblioteca de funciones matemáticas.

Las clases también tienen funcionalidades llamadas métodos. Para cargar dichas clases en un programa se usa `import`. Por ejemplo

```
Import java.util.Scanner
```

Nos incluye en el programa la clase `Scanner`. En caso de que queramos incluir todas las clases de un paquete escribiremos

```
Import java.util.*
```

Tabla de operadores de asignación

Nombre	operador	ejemplo de uso
asignación	=	a=b
Incremento	+=	a+=3
decremento	-=	a-=4
Multiplicación	*=	a*=6
División	/=	a/=8
Resto división	%=	a%=4

caracteres del teclado. La sintaxis de este procedimiento es la siguiente

```
InputStreamReader flujo;
```

```
BufferedReader teclado;
```

```
flujo= new InputStreamReader(System.in);
```

```
teclado= new BufferedReader(flujo);
```

```
String cd=teclado.readLine();
```

Para poder convertir estas cadenas de caracteres a datos numéricos de los diferentes tipos vistos en Java *-int, double, etc...*- tenemos que hacer referencia otra vez al concepto de clase.

Y es que Java define no solo los tipos básicos *int, float, double, etc...* sino también las clases correspondientes con una serie de métodos asociados.

De esta manera podemos definir una variable entera como un objeto de la clase *Integer* de la siguiente forma

```
Integer nombre_var
```

```
nombre_var= new Integer(valor)
```

El procedimiento es el mismo para un tipo *Long, Double, Float, etc...*

Estas clases tienen métodos que permiten el paso de cadena de caracteres a enteros, reales, etc....

En el caso de los enteros, el método que se usa es *ParseInt*, que funciona de la siguiente manera.

```
String cd=teclado.readLine() //continuación
```

```
int dato=Integer.parseInt(cd);
```

En el caso de los datos tipo *double*, el procedimiento es un poco más largo dado que requiere un paso intermedio. El procedimiento podemos verlo en el siguiente cuadro

```
String cd=teclado.readLine() //continuación

Double d=Double.valueOf(cd);

double x=d.doubleValue();
```

Donde *Double* es el objeto envoltorio del tipo de datos *double*.

- La segunda forma requiere del uso del API de Java empleando la clase *Scanner*, que se emplea de forma no estática, esto es creando un objeto de dicha clase como se muestra a continuación

```
Scanner sc =new Scanner (System.in)
```

Una vez creado el objeto se pueden leer los datos de teclado empleando los métodos siguientes

```
int l =sc.nextInt()

double d=sc.nextDouble()

String s=sc.nextLine()
```

9. Operadores y expresiones

Los programas implementados en Java, como en cualquier otro lenguaje de programación, constan de datos, llamadas a funciones y expresiones. Las expresiones son un conjunto de variables, constantes y funciones unidas entre sí por operadores. Estos operadores pueden relacionar varios datos en una expresión devolviéndonos un resultado alfanumérico o lógico. Los operadores se llaman binarios si tienen dos operandos, unarios si solo actúan sobre un

Tabla de operadores aritméticos

nombre	operador	ejemplo de uso
multiplicación	*	a*b
Resto division	%	a%b
Suma	+	a+b
Diferencia	-	a-b
Incremento	++	aA++
Decremento	--	a--
Cambio de signo	-	-a
división	/	a/b

El operador decremento disminuye en una unidad el valor de la variable a la que se le aplica. El operador incremento realiza la tarea opuesta.

Tabla de operadores relacionales

nombre	operador	Ejemplo de uso
Mayor que	>	a>b
Mayor o igual que	>=	a>=b
Menor que	<	a<b
Menor o igual que	<=	a<=b
Igual que	==	a==b
distinto	!=	a!=b

Tabla de operadores lógicos

nombre	operador	ejemplo de uso
Y logico	&&	a&&b
O logico		a b
NO logico	!	!a

Tabla de operadores a nivel de bit

nombre	operador	Ejemplo de uso
Y logico	&	a&b
O logico		a b
XOR	^	a^b
desp izq	<<	a=b<<2
desp der	>>	a=b>>2
complemento	~	~a

Ejemplo de uso de InstanceOf

```
cadena= cadena InstanceOf String
```

Comprueba que la variable *cadena* es del tipo *String*

Ejemplo de conversión explícita

```
int a;  
a=(int) 5.34;
```

Orden de prioridad para los operadores de cara a la asociatividad

Prioridad	Operadores	Asociatividad
1	New	
2	. [] ()	I - D
3	++ -- (prefijo)	D - I
4	++ -- (postfijo)	I - D
5	~ - +	D - I
6	(type)	D - I
7	* / %	I - D
8	+ - (binarios)	I - D
9	<< >> >>>	I - D

La asociatividad I-D quiere decir que primero se aplica el operador de la izquierda, mientras que la asociatividad D-I implica que primero se aplica el operador de la derecha.

operando y ternarios si devuelven un valor seleccionado entre dos posibles.

Existen diferentes tipos de operadores en Java, que podemos clasificar en la siguiente tabla

Tipo	Descripción
<i>Aritméticos</i>	Llevar a cabo operaciones aritméticas
<i>Relacionales</i>	Comparan números o caracteres
<i>Asignación</i>	Transfieren datos de una variable a otra, o el resultado de una serie de operaciones a una variable.
<i>Lógicos</i>	Permiten evaluar expresiones lógicas
<i>Operadores a nivel de bit</i>	Alteran o comparan valores numéricos a nivel de bit
<i>InstanceOf</i>	Identifica la clase a la que pertenece un objeto dado
<i>this</i>	Identifica la clase a la que pertenece el objeto actualmente ejecutado
<i>new</i>	Instancia un objeto
<i>Operador ?:</i>	Lleva a cabo una asignación condicionada
<i>Operador + con caracteres</i>	Permite concatenar cadenas

En los cuadros laterales podemos ver todos los operadores y ejemplos de su funcionamiento.

Debemos hacer alguna apreciación en el tema de los operadores y expresiones, en concreto cuando se plantea el problema de la conversión de tipos, esto es, cuando debemos convertir un tipo de datos a otro sin alterar su valor.

Las conversiones pueden ser de dos tipos

- *Implícitas*: se producen cuando se hacen asignaciones de un tipo numérico a una variable de otro tipo, o cuando se asigna a una variable el resultado de una expresión compleja formada por varios operadores y operandos.
- *Explícitas*: cuando se fuerzan mediante el operador de conversión cast, que tiene el formato

(tipo_datos) valor

donde *tipo_datos* es el tipo de datos al que queremos convertir *valor* –que será una constante o una variable–.

10. Estructuras de control

Los programas que hemos visto hasta ahora en Java solo ejecutan un conjunto de sentencias de manera secuencial, unas después de otras. Sin embargo esto solo es útil para programas muy sencillos. A la hora de resolver tareas más complejas es preciso establecer una serie de estructuras de control de flujo que permitan que la ejecución del programa siga distintas rutas en función de las condiciones que se planteen –estructuras selectivas– o que se permita ejecutar múltiples veces la misma instrucción en tanto en cuanto no cambien las condiciones del problema –estructuras repetitivas–. Veremos ambas en los próximos apartados

10.1. Estructuras condicionales

Las estructuras condicionales se basan la aparición de una determinada condición, el cumplimiento de la cual dirige el flujo de instrucciones en una dirección, y su no cumplimiento la dirige en otra distinta.

10.1.1. Estructura if-else

La estructura if es la estructura condicional más sencilla en los lenguajes de programación. Tiene la siguiente sintaxis

If (condición)

{

Instruccion1

Instruccion2

.....

}

Ejemplo de instrucción condicional if

```
String cadena;

InputStreamReader a = new
InputStreamReader(System.in);

BufferedReader      b=new
BufferedReader(a);
System.out.print("introduzca un
numero: ");
cadena=b.readLine();
int
numero=Integer.parseInt(cadena)

if (numero>5)
{
    System.out("el      número
introducido es mayor que 5");
}
```

Ejemplo de instrucción condicional if-else

```
String cadena;

InputStreamReader a = new
InputStreamReader(System.in);

BufferedReader      b=new
BufferedReader(a);
System.out.print("introduzca un
numero: ");
cadena=b.readLine();
int numero;
numero=Integer.parseInt(cadena)

if (numero>5)
{
    System.out.println("el      número
introducido es mayor que 5");
}
else
{
    System.out.println(" el numero es
menor o igual que 5");
}
```


Ejemplo de instrucción condicional if-else-if

```
String cadena;

InputStreamReader a = new
InputStreamReader(System.in);

BufferedReader      b=new
BufferedReader(a);
System.out.print("introduzca un
numero: ");
cadena=b.readLine();
int num;
num=Integer.parseInt(cadena)
if ((num>1)&&(num<5))
{
    System.out.println("el número
introducido es mayor que 1 y
menor que 5");
}
else if ((num>=5)&&(num<10))
{
    System.out.println(" el numero es
mayor o igual que 5 y menor que
10");
}
else
{
    System.out.println(" el numero es
menor o igual que 1 o mayor o
igual que 10");
}
```

Ejemplo de instrucción switch

```
String cadena;
InputStreamReader a = new
InputStreamReader(System.in);
BufferedReader      b=new
BufferedReader(a);
System.out.print("introduzca un
numero: ");
cadena=b.readLine();
int num;
num=Integer.parseInt(cadena)
switch (num)
{
    case 1: System.out.println("1");
            break;
    case 2: System.out.println("2");
            break;
    default: System.out.println("0");
}
}
```

El funcionamiento de esta instrucción es muy sencillo. Si se cumple la condición encerrada entre paréntesis, se ejecutan las instrucciones situadas entre las llaves. En caso contrario, el programa salta a la siguiente instrucción después de la llave de cierre y continúa su ejecución.

Una versión más avanzada de esta sentencia es la instrucción if-else, que tiene la siguiente sintaxis.

if (condición)

{
Instruccion1

Instruccion2

.....

}

else

{
Instruccion3

Instruccion4

}

En este caso si se cumple la condición se ejecutan las instrucciones dentro del if. En caso contrario se ejecutan las instrucciones dentro del else.

Por último nos podemos encontrar el llamado *if-else-if*, o *if anidado*, que tiene la siguiente sintaxis:

if (condicion1)

{
Instruccion1

Instruccion2

.....

}

else if (condicion2)

{
Instruccion3

Instruccion4

}

else

{
.....

}

En este caso si se cumple la primera condición, se ejecutan las instrucciones definidas dentro del primer *if*, en caso de que no se cumpla la primera condición, y se cumpla la segunda, se ejecutan las instrucciones definidas en el segundo *if*, y si tampoco se cumple la segunda condición, entonces se ejecutan las instrucciones por defecto, esto es las que están dentro del *else*.

10.1.2. Estructura *switch*

La estructura *switch* es una extensión de la estructura *if* anidada que se emplea en el momento en el que tenemos más de dos bloques alternativos. La sintaxis de *switch* es la siguiente:

```
switch (expresión) {
    case valor1: sentencias;
        break;
    case valor2: sentencias;
        break;
    .....
    default: sentencias;
}
```

En este caso se evalúa una expresión que ha de ser de tipo *int* o *char* –nunca *double*–.

En función el valor que tome se ejecuta un bloque de sentencias determinado. En caso de que no tome ninguno de los valores indicados, se ejecutan las instrucciones indicadas en *default*.

10.2. Estructuras repetitivas

Las estructuras repetitivas evitan el tener que repetir la misma instrucción muchas veces, al permitir la iteración de una instrucción dada un número determinado de veces. De esta manera se reduce el tamaño del código.

Ejemplo de código Java con *while*

```
String cadena;
InputStreamReader a = new
InputStreamReader(System.in);
BufferedReader      b=new
BufferedReader(a);
System.out.print("introduzca un
numero: ");
cadena=b.readLine();
int num;
num=Integer.parseInt(cadena)
int i;
i=0;
while (i<num)
{
    System.out.println(" aun no
        hemos salido del bucle");
    i++;
}
System.out.println("Ya      hemos
salido del bucle");
```

Ejemplo de código Java con for

```
String cadena;
InputStreamReader a = new
InputStreamReader(System.in);
BufferedReader      b=new
BufferedReader(a);
System.out.print("introduzca un
numero: ");
cadena=b.readLine();
int num;
num=Integer.parseInt(cadena)
int i;

for (i=0; i<=10;i++)
{
    System.out.println(" aun no
        hemos salido del bucle");
}
System.out.println("Ya      hemos
salido del bucle");
```

Ejemplo de código Java con do...while

```
String cadena;
InputStreamReader a = new
InputStreamReader(System.in);
BufferedReader      b=new
BufferedReader(a);
System.out.print("introduzca un
numero: ");
cadena=b.readLine();
int num;
num=Integer.parseInt(cadena)
int i;
i=0;
do
{
    System.out.println(" aun no
        hemos salido del bucle");
    i++;
}while (i<num)
System.out.println("Ya      hemos
salido del bucle");
```

10.2.1. La estructura while

La estructura `while` permite ejecutar una serie de instrucciones mientras que se cumpla una condición determinada. La sintaxis de esta estructura es

```
while (condicion)
```

```
{
```

```
    instruccion1
```

```
    instruccion2
```

```
    instruccion3
```

```
    .....
```

```
}
```

Cuando usamos la estructura `while` tenemos que tener en cuenta que la expresión que conforma la condición variará dentro del `while`, de tal manera que en algún momento no se cumpla y salgamos de bucle. En caso contrario el programa permanecería continuamente dentro del bucle.

10.2.2. La estructura for

La estructura `for`, a diferencia de `while`, repite una serie de instrucciones un número determinado de veces, empleando un contador, de acuerdo con la siguiente sintaxis:

```
for (valor_i;valor_f;inc/dec)
```

```
{
```

```
    instruccion1
```

```
    instruccion2
```

```
    instruccion3
```

```
    .....
```

```
}
```

En este caso, el contador empieza en un valor inicial, llega hasta un valor final y se incrementa o decrementa en cada paso de la ejecución del bucle.

Una norma que se debe de seguir es no modificar el contador dentro del bucle, dado que esa alteración

genera una modificación en el número de repeticiones del bucle.

Un tipo especial de bucle es el bucle infinito, que es aquel que se ejecuta de manera indefinida. En este caso, la sintaxis es la siguiente

```
for (valor_i;valor_f;inc/dec)
{
    instruccion1
    instruccion2
    instruccion3
    .....
}
```

10.2.3. La estructura do...while

Do...while representa una estructura en la que la condición se ejecuta la menos una vez. Tiene la siguiente sintaxis:

```
do
{
    instruccion1
    instruccion2
    instruccion3
    .....
}while (condición)
```

11. Arrays

Un array es una secuencia de elementos del mismo tipo que ocupan posiciones de memoria contiguas, y se numeran consecutivamente empleando un índice entero, de tal manera que el elemento que ocupa la primera posición en un array a de n elementos será a[0], el segundo a[1],y el n-ésimo será a[n].

Para poder acceder al contenido de una posición del array hay que emplear el valor del índice correspondiente a esa posición, tal y como aparece a continuación

```
auxiliar=a[n];
```

Ejemplo del uso de arrays en Java

```
Public class Media {

    Public static void main() {

        int nums[] ={3,4,7,8,4,5,6};
        int suma=0;
        double media;
        for (int i=0;i<nums.length; i++)
        {
            Suma=suma+nums[i];
        }

        media=(double)suma/nums.length;
        System.out.println("La      media
es"+media);
        Systema.out.println("Longitud
es"+nums.length);
    }
}
```

Ejemplo de uso de arrays multidimensionales en Java. En concreto este ejemplo permite leer y escribir datos en un array bidimensional.

```
Int tabla[] [] = new int[3][4];
double   resistencias[][]=      new
double[4][5];
boolean  asientoslibres[] []=    new
boolean[28][18];
BufferedReader entrada = new
BufferedReader(new
InputStreamReader(System.in));
Tabla[1][1]=Integer.parseInt(entrada
.readLine());
System.out.println(tabla[1][1]);
Resistencias[2][1]=Double.valueOf
(entrada.readLine()).doubleValue();
If (asientoslibres[3][1])
System.out.println("asiento
disponible");
Else
System.out.println("asiento
ocupado");
```

Código del algoritmo de recorrido de un array de n elementos.

```
.....
int n=sc.nextInt();
int lista[] =new int[n];
int aux;
for (int i=0;i<n; i++)
{
    lista[i]=sc.nextInt();
}
for ( i =0; i<n; i++)
{
    System.out.println(lista[i]);
}
.....
```

Código del algoritmo de recorrido de un array bidimensional de nxm elementos.

```
.....
int n=sc.nextInt();
int m =sc.nextInt();
int lista[][] =new int[n][m];
int aux;
for (int i=0;i<n; i++)
{
    for(int j=0; j<m; j++)
    {
        lista[i][j]= sc.nextInt();
    }
}
for (int i=0;i<n; i++)
{
    for(int j=0; j<m; j++)
    {
        System.out.println(lista[i][j]);
    }
}
.....
```

Donde *auxiliar* es una variable del tipo de los elementos que conforman el array. En caso de intentar acceder a una posición no reservada, por ejemplo con un índice negativo o un índice mayor que el número de elementos reservados, se produce un error de ejecución.

La declaración de un array tiene la forma

tipo_dato n_variable[]

n_variable= new tipo_dato[n_elementos]

dónde *tipo_dato* es el tipo de datos, *n_variable* el nombre de la variable y *n_elementos* el número de elementos del array.

Para definir un array también se puede hacer de la siguiente manera

tipo_dato n_variable [] ={a₁,a₂,a₃,.....a_n};

Para Java, cada array es un objeto, que como tal tiene atributos. Uno de ellos es el campo *length*, que nos devuelve el número de elementos del array.

Los arrays no solo pueden ser de una dimensión, sino que también podemos encontrarlos de 2, 3,n dimensiones.

En este caso la definición del array se hace de la siguiente manera.

Tipo_dato n_variable [][]...[];

N_variable = new tipo_dato[n₁][n₂][n₃]...[n_i];

Para acceder al contenido de una posición concreta de un array multidimensional tenemos que indicar los índices que posicionan a ese elemento, de esta manera

n_variable[i][j]....[z]

Para inicializar el contenido de un array multidimensional tenemos que emplear llaves, encerrando entre ellas la lista de elementos de que consta cada fila, separada por comas.

tipo_dato n_variable [] ={{a₁,a₂,a₃,...,a_n}{b₁,b₂,b₃,...b_n}}

Para mostrar por pantalla el contenido de cada posición del array se emplea la siguiente instrucción

System.out.println(n_variable[i]);

donde i es el índice de la posición del array que se muestra por pantalla

Si queremos asignar a un array un valor vacío se emplea la palabra clave *null*.

int n_variable[];
n_variable=null;

En el caso de que se asigne *null* a un array que tuviera contenido, dicho contenido al no ser referenciado por ninguna variable será borrado por el recolector de basura.

Cuando se quiere pasar un array como parámetro de una función se hace de la siguiente manera

int n_variable[]=new int[n];
n_funcion(n_variable);

.....
tipo_datos n_funcion(int n_var[])
{
.....
}

Código del algoritmo de recorrido de un array de n elementos.

```
.....
int n=sc.nextInt();
int lista[]=new int[n];
int aux[];
aux=lista;
/*ahora aux y lista apuntan al mismo
array en memoria*/
lista=null;
/*ahora solo aux apunta a la estructura
en memoria*/
aux=null;
/*ahora ninguna variable apunta a la
estructura en memoria y será borrada
por el recolector de basura*/
.....
```

API Arrays

Es un API de Java que permite realizar operaciones con arrays. Está en `Java.util.Arrays`.

Algunas de las operaciones que se realizan con dicha API, dado un array `n_variable[]`, son:

- **n_variable.length:** devuelve la longitud del array
- **Arrays.toString(n_variable):** combinado con `System.out.println()` nos va a devolver el contenido del array sin usar una estructura for para recorrerlo.
- **Arrays.fill(n_variable,valor):** devuelve el array relleno con *valor*
- **Arrays.fill(n_variable,inicio,fin,valor):** devuelve el array relleno con *valor* solo en las posiciones entre *inicio* y *fin*
- **Arrays.equals(n_variable,n_variablebis):** nos indica con `true` o `false` si los dos arrays son iguales o no.
- **Arrays.binarySearch(n_variable, a):** búsqueda dicotómica de *a* en *n_variable* -ordenada-
- **Arrays.sort(n_variable):** ordena *n_variable* de forma creciente.

12. Caracteres

Tabla de caracteres especiales

carácter	nombre
\b	Borrado a la izquierda
\n	Nueva línea
\r	Retorno de carro
\t	tabulador
\f	Nueva página
\'	Comilla simple
\"	Comilla doble
\\	Barra invertida

Clase Character

Amplia las características del tipo char, haciendo más fáciles muchas tareas

Metodos de la clase Character

método	Funcion que realiza
boolean Character.isDigit(char c)	Indica si c es un dígito o no
boolean Character.isLetter(char c)	Determina si el carácter es una letra manuscrita o minúscula
boolean Character.isLetterOrDigit(char c)	Indica si el carácter es una letra o un dígito
Boolean Character.isLowerCase(char c)	Indica si es una letra minúscula
boolean Character.isUpperCase(char c)	Indica si es una letra mayúscula
Boolean Character.isSpaceChar(char c)	Indica si es un espacio de barra espaciadora
Boolean Character.isWhiteSpace(char c)	Indica si es un espacio en blanco, tabulador, retorno de carro o nueva línea
Char Character.toLowerCase(char c)	Devuelve la minúscula asociada
Char Character.toUpperCase(char c)	Devuelve la mayúscula asociada
Char Character.toString(char c)	Devuelve el string de longitud que corresponde al carácter c

El tipo carácter se define como una letra, número, ideograma o cualquier otro símbolo y se representa entre comillas simples.

En Java la asignación de caracteres a una variable se realiza de la siguiente manera

```
char n_variable='<carácter>;
```

Una manera alternativa de asignar un carácter a una variable de tipo char, es emplear el código Unicode del carácter

```
char n_variable='\uXXXX';
```

```
char n_variable=XX;
```

Donde '\uXXXX' es el codepoint o número que identifica el carácter en Unicode en hexadecimal y XX el codepoint en decimal.

Se pueden convertir los valores de la variable tipo char en decimal a carácter como se muestra aquí

```
char n_variable='<carácter>;
```

```
System.out.println((int)n_variable); //muestra el valor numérico de '<carácter>'
```

```
Int n_variable=XX
```

```
System.out.println((char)n_variable); //muestra el valor numérico de '<carácter>'
```

También se pueden realizar operaciones 'aritméticas' con caracteres, de modo que podemos desplazarnos entre ellos, tal y como se muestra a continuación.

```
char n_variable='<carácter>'+2; //vale el carácter situado dos posiciones más adelante
```

```
char n_variable='<carácter>' -2; //vale el carácter situado dos posiciones más atrás
```

El tipo carácter, `char`, que es uno de los tipos primitivos de Java, si bien nos permite trabajar con caracteres simples no nos permite usar cadenas de caracteres, como frases o textos completos.

Para ello empleamos las APIS de la clase `Character`, situada en `java.lang`. Algunas de los métodos asociados se pueden ver en la página anterior.

13. Cadenas (String)

Una cadena es una secuencia de caracteres limitada entre comillas dobles. En Java se usa la clase `String` para definir las cadenas de caracteres como objetos, no como variables.

El formato de definición e inicialización de un objeto del tipo `String` es

```
String n_variable="cadena de caracteres";
```

o como alternativa

```
String n_variable;  
n_variable="cadena de caracteres";
```

También se puede emplear el método constructor para crear cadenas, de la forma siguiente

```
String c;  
c=new String("cadena  
de caracteres");
```

```
String c1,c2;  
.....  
c=new String();  
c="cadena de  
caracteres"
```

Tabla de métodos de la clase `String`

método	función
<code>length()</code>	Devuelve el número de caracteres
<code>Concat(cadena)</code>	Concatena la cadena parametro detrás de la cadena invocante
<code>charAt(pos)</code>	Devuelve el carácter situado en la posición pos
<code>Substring(a,b)</code>	Devuelve la subcadena situada entre las posiciones a y b
<code>compareTo(cad) / compareToIgnoreCase(cad)</code>	Compara la cadena invocante y la cadena parametro
<code>Equals(cad) / equalsIgnoreCase(cadena)</code>	Indica si la cadena invocante y la cadena parametro son iguales
<code>Replace(c1,c2)</code>	Sustituye todos los caracteres c1 por c2
<code>toUpperCase()</code>	Pasa una cadena a mayúsculas
<code>valueOf</code>	Convierte cualquier tipo de dato a cadena
<code>indexOf(int c) / indexOf(cad)/indexOf(c.inicio) / indexOf(cad.inicio)</code>	Busca la primera ocurrencia de un carácter c o una cadena
<code>lastIndexOf(c) /lastIndexOf(cad) /lastIndexOf(c.inicio) / lastIndexOf(cad, inicio)</code>	Devuelve la última ocurrencia del carácter c o de la cadena cad
<code>isEmpty()</code>	Devuelve true si la cadena esta vacía
<code>Contains(cad)</code>	Devuelve true si contiene la cadena cad
<code>startsWith(cad)/startsWith(cad.inicio)</code>	Devuelve true si empieza con la cadena cad o esta a partir de la posición inicio
<code>endsWith(cad)</code>	Devuelve true si termina con la cadena cad.
<code>toLowerCase()</code>	Pasa la cadena a minúsculas
<code>toUpperCase()</code>	Pasa la cadena a mayúsculas
<code>Replace(car,otro)</code>	Devuelve la cadena pero cambiando el carácter car por otro

Ejemplo de código Java que emplea la clase String

```

import java.io.*
Class Saludo
{
    Public static void main() throws
    IOException
    {
        String nom;
        BufferedReader entrada= new
        BufferedReader(new
        InputStreamReader(System.in));
        do
        {
            System.out.println("Dime      tu
            nombre");
            System.out.flush();
            nom= entrada.readLine();
        }while(nom==null);
        System.out.println("Hola " + nom+
        "¿como estas?");
    }
}

```

Pseudocódigo del algoritmo de ordenación por selección

```

para i=1 hasta n-1
    minimo = i;
    para j=i+1 hasta n
        si lista[j] < lista[minimo] entonces
            minimo = j /* (!) */
        fin si
    fin para
    intercambiar(lista[i], lista[minimo])
fin para

```

Código del algoritmo de ordenación por selección en Java

```

public static void seleccion(int[] a)
{
    for (int i = 0; i < a.length - 1; i++)
    {
        int min = i;
        for (int j = i + 1; j < a.length; j++)
        {
            if (a[j] < a[min])
            {
                min = j;
            }
        }
        if (i != min)
        {
            int aux= a[i];
            a[i] = a[min];
            a[min] = aux;
        }
    }
}

```

14. Algoritmos de ordenación

Una de las tareas más comunes a realizar cuando se trabaja con grandes cantidades de datos es la ordenación de estos siguiendo diferentes criterios. Para ello se han desarrollado múltiples algoritmos de entre los cuales podemos destacar, dentro de la ordenación, las técnicas de selección, inserción y burbuja.

14.1. Ordenación por selección

La ordenación por selección es un algoritmo de ordenación que emplea n^2 operaciones para ordenar n elementos. Su funcionamiento es el siguiente:

- Buscar el mínimo elemento de la lista
- Intercambiarlo con el primero
- Buscar el siguiente mínimo en el resto de la lista
- Intercambiarlo con el segundo

Y en general:

- Buscar el mínimo elemento entre una posición i y el final de la lista
- Intercambiar el mínimo con el elemento de la posición i

14.2. Ordenación por inserción

La ordenación por inserción es un mecanismo sencillo de ordenación que requiere n^2 operaciones para ordenar una lista de n elementos.

Inicialmente se tiene un solo elemento, que obviamente es un conjunto ordenado. Después, cuando hay k elementos ordenados de menor a mayor, se toma el elemento $k+1$ y se compara con todos los elementos ya ordenados, deteniéndose cuando se encuentra un elemento menor (todos los elementos mayores han sido desplazados una posición a la derecha) o cuando ya no se encuentran elementos (todos los elementos fueron desplazados y este es el más pequeño). En este punto se inserta el elemento $k+1$ debiendo desplazarse los demás elementos.

14.3. Ordenación por burbuja

La ordenación por burbuja es un algoritmo de ordenación que funciona comparando cada elemento de la lista que va a ser ordenada con el siguiente, e intercambiándolos de posición en caso de que estén en orden equivocado. Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada. También recibe el nombre de *algoritmo de intercambio directo*.

15. Métodos

En el tema anterior hemos definido teóricamente lo que son los métodos y como están estructurados. Aquí haremos referencia únicamente a la implementación en Java de los mismos, y a los conceptos de paso por valor y paso por referencia.

La sintaxis de la declaración de un método en Java tiene la siguiente forma.

tipo nombre (tipo1 param1, tipo2 param2, tipo3 param3,...)

```
{
    //cuerpo del método
    return resultado;
}
```

Donde *tipo*, *tipo1*, *tipo2*, ... son los tipos de datos y *param1*, *param2*,son los parámetros de entrada del método.

Los métodos devuelven información a través de la sentencia *return resultado*, donde resultado es una variable del mismo tipo que el método.

En Java el paso de parámetros a los métodos se hace por valor, esto es, se crea una copia del valor que se pasa por parámetro, de tal manera que las modificaciones que sufre dentro del método no afectan a la variable original, que había sido definida fuera del método.

Pseudocódigo del algoritmo de ordenación por inserción en Java

```
algoritmo insercion
para i desde 2 hasta n
    j=i-1
    mientras (j>=1) Y (A[j]>A[j+1])
        hacer
            temporal=A[j+1]
            A[j+1]=A[j]
            A[j]=temporal
            j=j-1
    fin mientras
fin para
fin algoritmo
```

Código Java de ord. por selección

```
public void insertionSort(int[] vector){
    for (int i=1; i < vector.length; i++){
        int temp = vector[i];
        int j;
        for (j=i-1; j >= 0 && vector[j] > temp; j--){
            vector[j + 1] = vector[j];
        }
        vector[j+1] = temp;
    }
}
```

Pseudocódigo del algoritmo de ordenación por burbuja.

```
para i desde 1 hasta n-1 hacer
    para j desde 1 hasta n-i hacer
        si A[j] > A[j+1] entonces
            intercambiar A[j] y A[j+1]
    fin para
fin para
```

Código en Java de ord. burbuja.

```
public static void burbuja(int [] A){
    int i, j, aux;
    for(i=0; i<A.length-1; i++){
        for(j=0; j<A.length-i-1; j++){
            if(A[j+1]<A[j]){
                aux=A[j+1];
                A[j+1]=A[j];
                A[j]=aux;
            }
        }
    }
}
```

Ejemplo de función sobrecargada

```
static int suma(int a, int b)
{
    int suma;
    suma=a+b;
    return suma;
}

static double suma(int a, double
porcentaje a, int b, double
porcentaje b)
{
    double suma;

    suma=a*porcentaje a/(porcenta
je a+porcentaje b) +
b*porcentaje b/(porcentaje a+p
orcentaje b)
    return suma;
}
```

Ejemplo de función recursiva

```
int factorial (int n)
{
    int resultado;
    if (n==0)
    {
        resultado=1
    }
    else
    {
        resultado=n*factorial (n-1);
    }
    return resultado;
}
```

Los métodos pueden llevar los modificadores de acceso *public*, *protected* y *private*.

Los métodos *public* pueden ser accedidos por otros objetos, los *private* únicamente por métodos de la clase en la que se define el método privado, y los *protected* pueden ser llamados tanto por los métodos de su misma clase como por los de las clases derivadas.

Asimismo los métodos pueden sobrecargarse, esto es, tener dos métodos con el mismo nombre dentro de un programa diferenciándose únicamente en las listas de parámetros. En el cuadro de la derecha se puede ver un ejemplo de sobrecarga.

Por otro lado los métodos también pueden ser recursivos, llamándose a sí mismos a lo largo de su cuerpo de instrucciones. Un ejemplo es la función factorial, como se muestra en el cuadro.