

**FRANCISCO ARAGÓN MESA**

INTRODUCCIÓN A LA  
PROGRAMACIÓN  
ORIENTADA A OBJETOS



TEMARIO



- 1- **Definición y principales características del lenguaje Java**
- 2- **Instalación, configuración y utilización del entorno de desarrollo Eclipse . :** Instalación del IDE Eclipse; Configuración del entorno de desarrollo; Nociones básicas de utilización; La primera aplicación.
- 3- **Elementos básicos de Java. :** Identificadores; Comentarios; Sentencias; Bloques de código; Expresiones; Variables; Los tipos básicos de datos; Las cadenas de caracteres o strings; Palabras clave.
- 4- **Operadores. :** Operadores aritméticos; Concatenación de strings; Precedencia de operadores; Conversión automática y promoción; operadores unarios; Operadores relacionales; Operadores lógicos.
- 5- **Sentencias de control de flujo. :** Sentencias condicionales; Sentencias iterativas.
- 6- **Conceptos básicos de programación orientada a objetos. :** Concepto; Clase; Miembro dato; Constructor; Método; Objeto.
- 7- **Paquetes (package). :** El paquete (package); El comando *import*; Paquetes estándar
- 8- **Composición. :** La clase *Punto*; La clase *Rectángulo*; Objetos de la clase *Rectángulo*
- 9- **La clase String. :** La clase *String*; Cómo se obtiene información acerca del string; Comparación de strings; Extraer un substring de un string; Convertir un número a string; Convertir un string en número; La clase *StringBuffer*.
- 10- **Arrays. :** Decalarar y crear un array ; Inicializar y usar el array ; Arrays multidimensionales.
- 11- **La clase Random. :** Importar y crear objetos de la clase *Random*; Comprobación de la uniformidad de números aleatorios .
- 12- **La clase Lista (análisis de un caso práctico).**
- 13- **Modificadores de variables. :** Diferencia entre modificador static y final; Relación con variables de instancia y variables de clase.
- 14- **La clase Math. :** Miembros dato constantes; Funciones miembro.
- 15- **Modificadores de acceso. :** public y private.
- 16- **Función miembro toString.**
- 17- **Herencia. :** La clase base; La clase derivada; Controles de acceso (public, private y protected); La clase base *Object*.
- 18- **Jerarquía de clases. :** Clases y métodos abstractos; El operador instanceof.
- 19- **La palabra clave final. :** Clases y métodos finales.
- 20- **Interfaces. :** Definición; Diferencia entre un interface y una clase abstracta.
- 21- **Excepciones. :** Definición; Captura de excepciones; Lanzar excepciones; La cláusula finally.
- 22- **El interface Cloneable. :** Duplicación de objeto.
- 23- **La clase Vector. :** Crear un vector; Añadir elementos al vector; Acceso a los elementos de un vector.
- 24- **La clase StringTokenizer. :** Función; Obtención de tokens.
- 25- **Archivos y directorios. :** La clase File; Creación de filtros.
- 26- **Entrada/salida estándar. :** Los objetos *System.in* y *System.out* ; La clase *Reader*.
- 27- **Entrada/salida a un archivo en disco. :** Lectura de un archivo de texto; Lectura/escritura
- 28- **Leer y escribir datos primitivos. :** Flujos de datos *DataInputStream* y *DataOutputStream*.
- 29- **Leer y escribir objetos. :** El interface *Serializable*; Lectura/escritura; El modificador *transient*.

# **DEFINICIÓN Y PRINCIPALES CARACTERÍSTICAS DEL LENGUAJE JAVA**

### Definición:

Java es un **lenguaje de programación orientado a objetos**, desarrollado por Sun Microsystems a principios de 1991, con el que se van a poder crear tanto programas asociados a páginas HTML (applets) como programas independientes de éstas (aplicaciones). Y todo ello, independiente de la plataforma de computación. Los programas hechos en Java podrán ejecutarse en INTEL, MOTOROLA, Solaris, Windows y Macintosh, entre otros.

### Características principales:

- **Orientado a Objetos:** Java organiza sus programas en una colección de objetos. Esto nos va a permitir estructurar los programas de una manera más eficiente y en un formato más fácil de comprender.
- **Distribuido:** Java dispone de una serie de librerías para que los programas se puedan ejecutar en varias máquinas y puedan interactuar entre sí.
- **Robusto:** Java está diseñado para crear software altamente fiable.
- **Seguro:** Java cuenta con ciertas políticas que evitan que se puedan codificar virus con este lenguaje, sin olvidar además que existen muchas otras restricciones que limitan lo que se puede o no se puede hacer con los recursos críticos de una máquina.
- **Interpretado:** La interpretación y ejecución se hace a través de la **Máquina Virtual Java (JVM)** es el entorno en el que se ejecutan los programas Java, su misión principal es la de garantizar la ejecución de las aplicaciones Java en cualquier plataforma.
- **Independiente de la Arquitectura:** El código compilado de Java se va a poder usar en cualquier plataforma.
- **Multiejecución:** Java permite elaborar programas que permitan ejecutar varios procesos al mismo tiempo sobre la misma máquina.

# **INSTALACIÓN, CONFIGURACIÓN Y UTILIZACIÓN DEL ENTORNO DE DESARROLLO ECLIPSE**

## Instalación del IDE Eclipse

Ahora procederemos a la instalación del IDE (Entorno de desarrollo) , el cual nos proveerá de completa asistencia a la hora de escribir nuestro código, como así depurarlo, compilarlo y ejecutarlo. En esta Sección se brindará soporte para el **IDE Eclipse SDK**, [www.eclipse.org](http://www.eclipse.org) , que es una de las herramientas gratuitas de desarrollo que dispone de interesantes utilidades para Java .

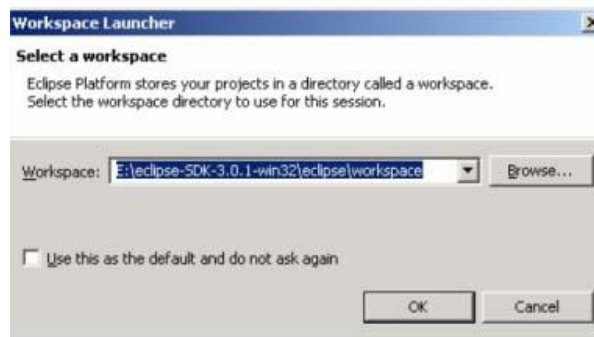
Antes necesitamos instalar las librerías y clases principales de Java SDK 1.6 ([www.sun.com](http://www.sun.com)) para poder trabajar con java en Eclipse.

Para la **instalación**, se siguen los siguientes pasos:

- Instalar JRE o JDK ejecutando el ejecutable *jdk-6-windows-i586.exe* y siguiendo los pasos del asistente de instalación.
- Descomprimir el directorio “eclipse” del archivo comprimido *eclipse-SDK-3.2-win32.zip* al lugar deseado del disco duro (por ejemplo, a *C:\Archivos de programas*).
- Para ejecutarlo solo hay que arrancar el fichero *Eclipse.exe* que se encuentra dentro del directorio que hemos descomprimido. También podemos crear un acceso directo en el escritorio.

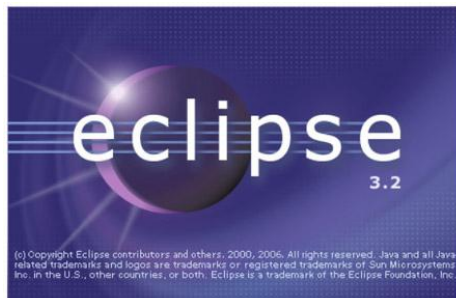
Para la **configuración**, se siguen los siguientes pasos:

- Una vez que hemos arrancado el programa *Eclipse.exe* nos pedirá que le demos la ruta por defecto donde queramos que eclipse nos vaya guardando los proyectos que creemos, marcamos la casilla para que no nos haga otra vez la pregunta:

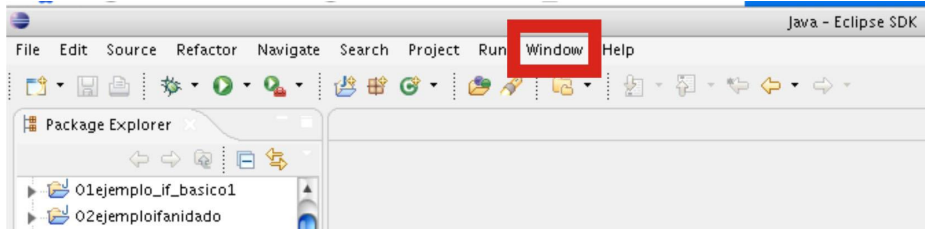


Podemos darle otra ruta diferente a la que nos sugiere donde pone Browse...

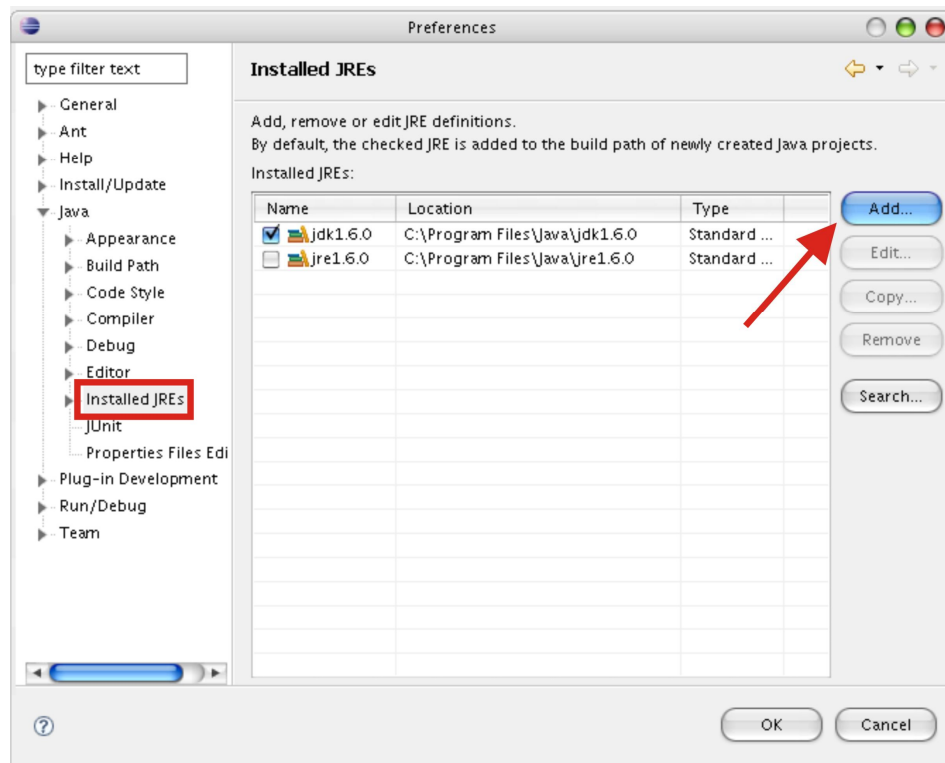
- Después se iniciará el programa e iniciamos workbench.



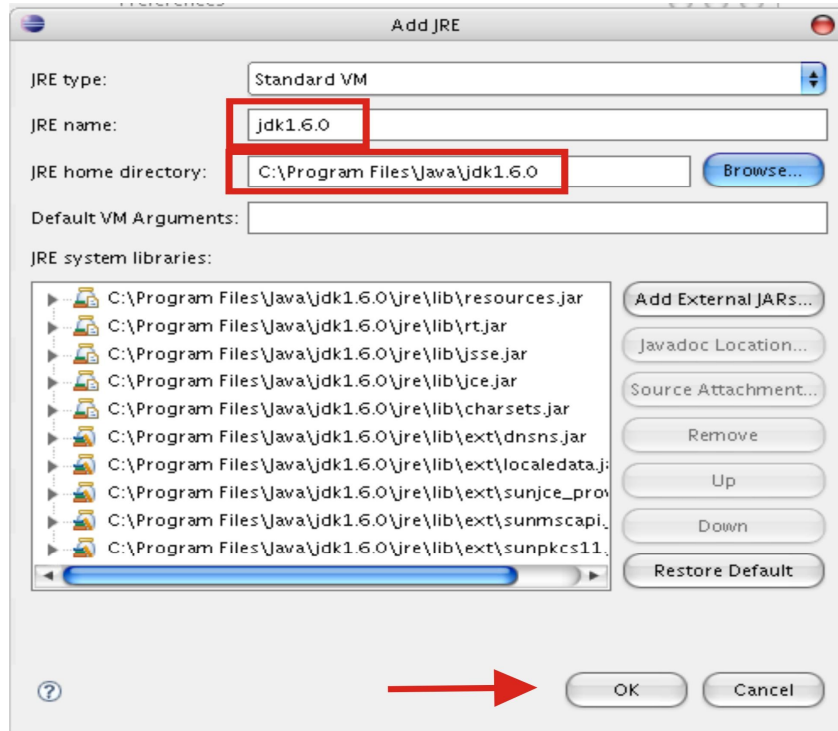
- Una vez inicializado nos dirigimos a window -> preferences...



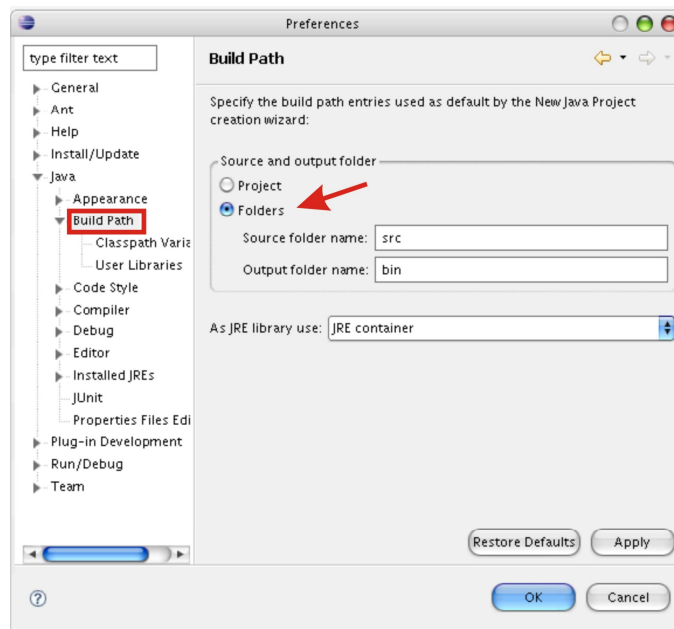
- En el cuadro de preferencias nos dirigimos a Java -> Installed JREs y accionar en Add...



- En el cuadro que nos aparece escribimos en *JRE name* el nombre del kit de desarrollo que anteriormente hemos instalado, en este caso *jdk1.6.0*. Donde pone *JRE home directory* le indicamos la ruta donde esta instalado mediante el botón Browse... seguidamente le damos a OK.



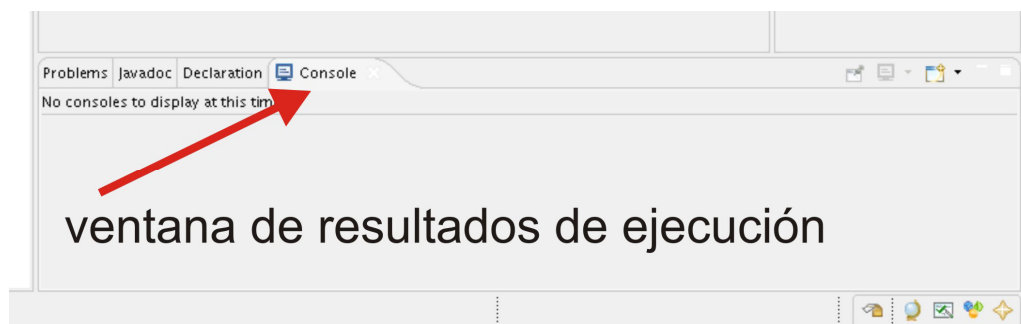
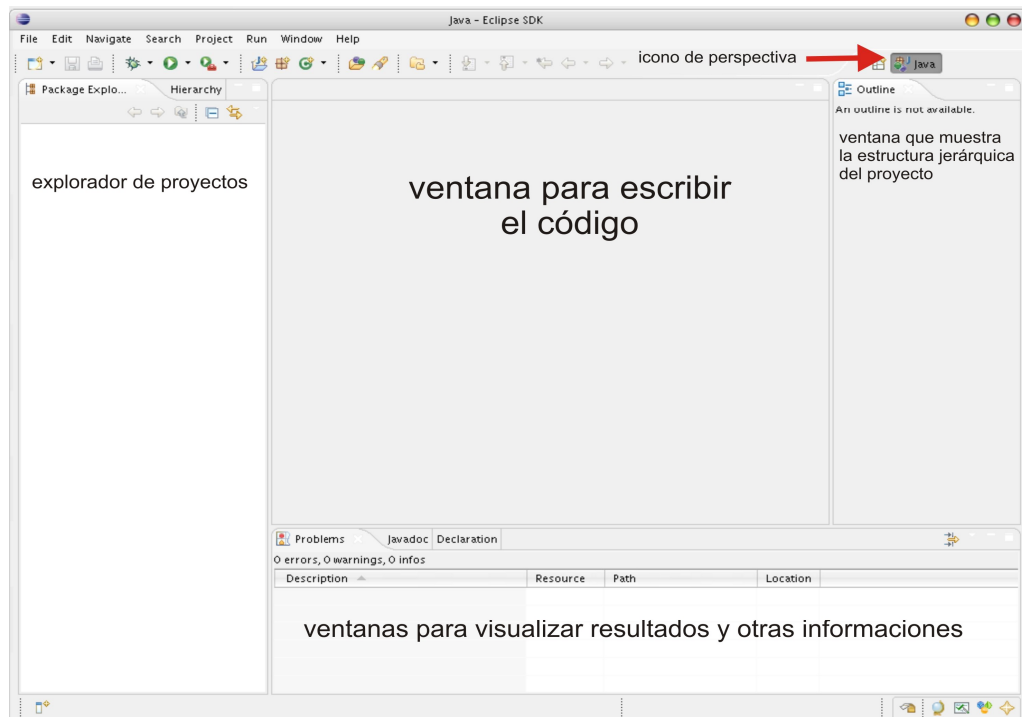
- Lo siguiente será indicar como se guardarán los proyectos. En Build Path señalamos Folders, de esta manera cada p royecto lo organizará en 2 carpetas, una llamada src donde se guardará el código fuente y la otra llamada bin donde se guardaran los archivos que contendrán las clases.



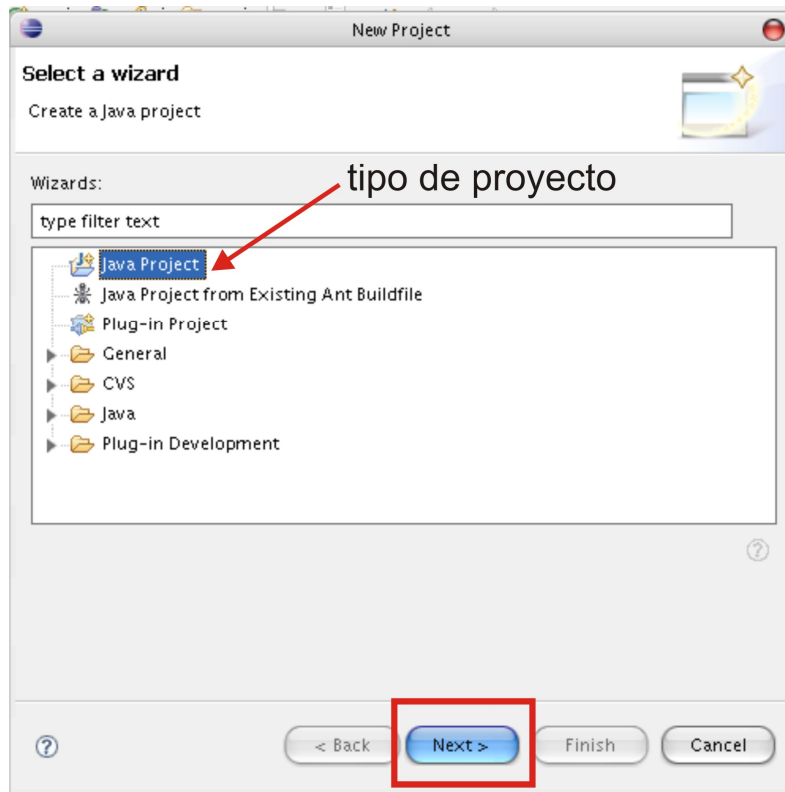


## Nociones básicas de utilización

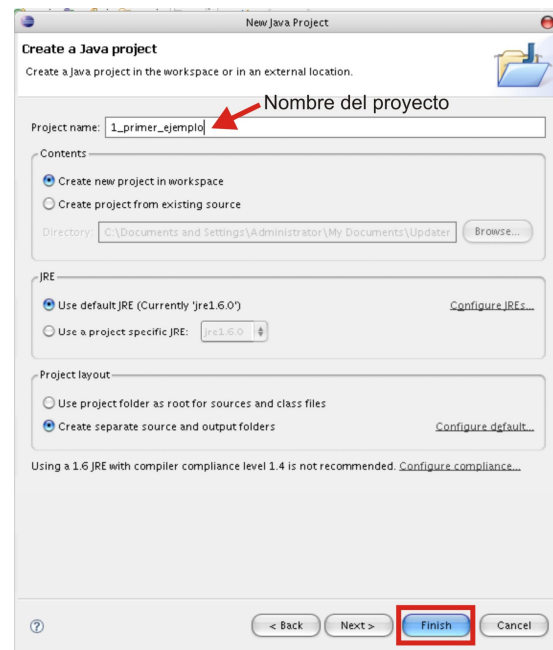
- Para comenzar fijémonos en como esta estructurado el **espacio de trabajo**. En *Window->Show View* podremos añadir mas ventanas al entorno de trabajo. Una de ellas se denomina *console* que es donde visualizaremos los resultados de ejecución de un proyecto.



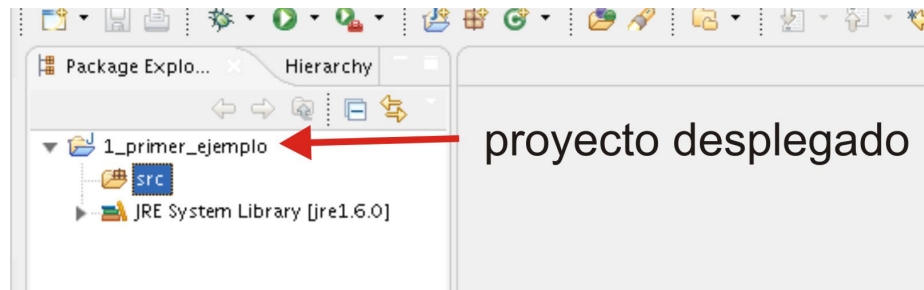
- Ahora vamos a proceder a crear un nuevo proyecto. Para ello pinchamos con el botón derecho del ratón en la ventana del explorador de proyectos y le damos a *new ->Project*. Nos aparecerá una ventana para elegir el tipo de proyecto, elegimos *Java Project* y le damos a *Next*.



- Seguidamente nos mostrará un cuadro para poner nombre al proyecto. Después de ponerle nombre le damos a *Finish*.



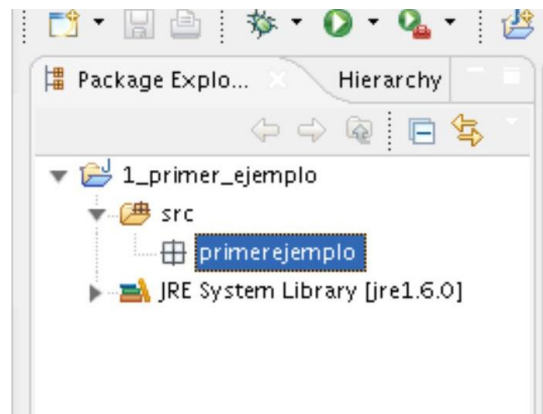
- Nuestro proyecto se habrá añadido en la ventana del explorador. Desplegamos el proyecto, damos con el botón derecho del mouse en SRC y elegimos new ->package



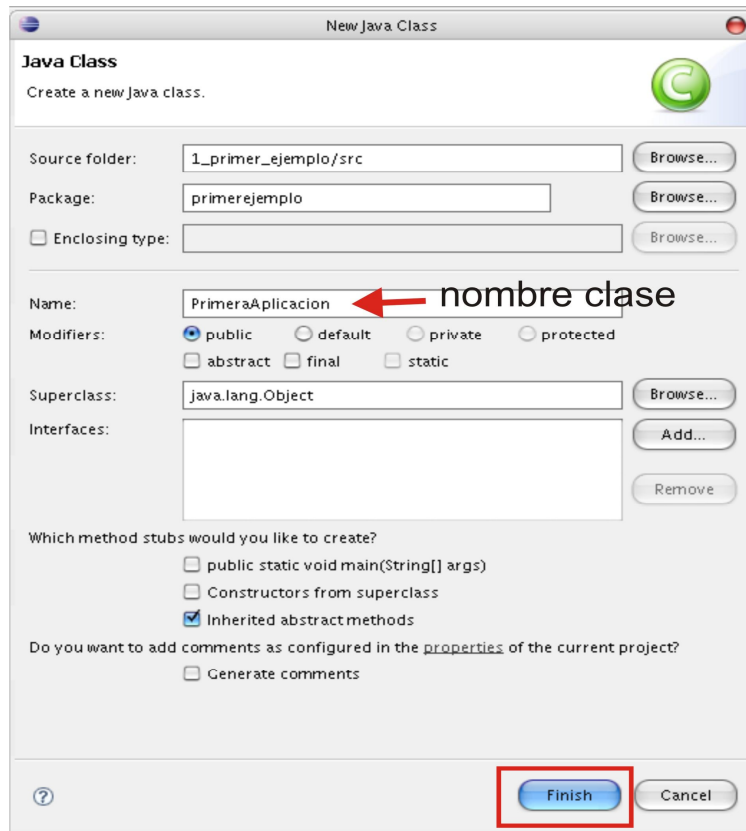
- Nos saldrá un cuadro donde introduciremos el nombre del paquete (package) que contendrá las clases de la aplicación. El nombre del paquete no debe tener espacios en blanco ni comenzar por letra mayúscula. Cuando acabemos de introducir el nombre le damos a *Finish*.



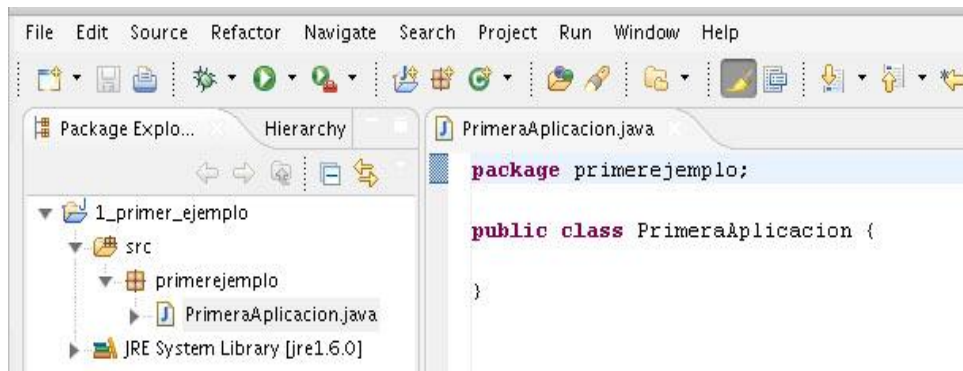
- El paquete se habrá añadido a la carpeta src. Desplegamos src, damos con el botón derecho del mouse en el paquete y elegimos new ->class..



- Nos saldrá un cuadro donde introduciremos el nombre de la clase. El nombre debe empezar por letra mayúscula y no tener espacios en blanco. Cuando acabemos de introducir el nombre le damos a *Finish*.



- Ahora ya podremos observar que en la zona del código fuente se ha añadido el nombre del paquete y de la clase para poder comenzar a escribir lo demás.



## La primera aplicación

En nuestra primera aplicación mostraremos un mensaje en pantalla. Para ello comenzaremos por añadir una función principal( *main*) dentro de la clase (dentro de las llaves { })

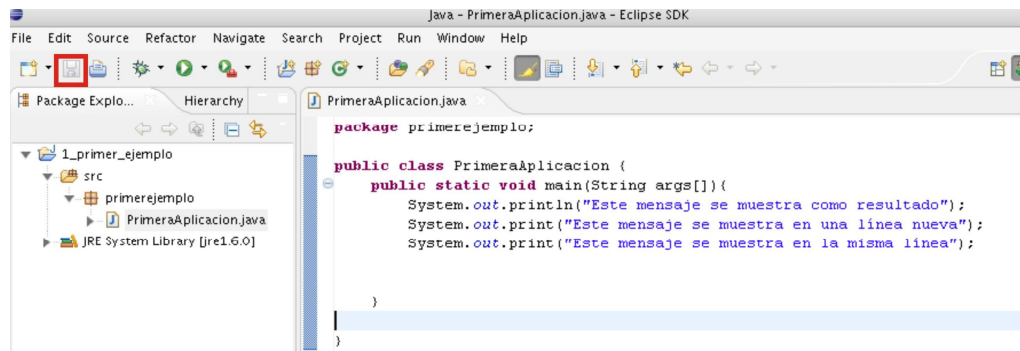
La función principal main será:

```
public static void main(String args[]){ }
```

y dentro de la función principal:

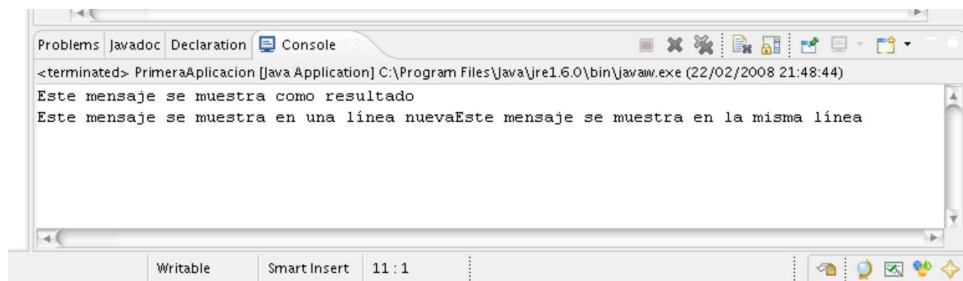
```
System.out.println("Este mensaje se muestra como resultado");
System.out.print("Este mensaje se muestra en una línea nueva");
System.out.print("Este mensaje se muestra en la misma línea");
```

- Así es como quedaría el código escrito en la pantalla:



**Antes de probar el resultado es conveniente guardar los cambios.**

Para ejecutar el programa hay que dar con el botón derecho del ratón en el archivo con la extensión .java y seleccionar *Run as->1 java application*.



**Este sería el resultado de la ejecución.**

System.out.print : muestra una salida en pantalla y seguidamente no hace un salto de línea.  
 System.out.println : muestra una salida en pantalla y seguidamente hace un salto de línea.  
 public static void main(String args[]): estos elementos los trataremos a continuación.

# **ELEMENTOS BÁSICOS DE JAVA**

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

- Identificadores: los nombres que se dan a las variables
- Tipos de datos
- Palabras reservadas: las palabras que utiliza el propio lenguaje
- Sentencias
- Bloques de código
- Comentarios
- Expresiones
- Operadores

A lo largo de las páginas que siguen examinaremos en detalle cada uno de estos elementos.

### Identificadores

Un identificador es un nombre que identifica a una variable, a un método o función miembro, a una clase. Todos los lenguajes tienen ciertas reglas para componer los identificadores:

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

Tipo de identificador	Convención	Ejemplo
nombre de una clase	Comienza por letra mayúscula	String, Rectangulo, CinematicaApplet
nombre de función	comienza con letra minúscula	calcularArea, getValue, setColor
nombre de variable	comienza por letra minúscula	area, color, appletSize
nombre de constante	En letras mayúsculas	PI, MAX_ANCHO

## Comentarios

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

En Java existen tres tipos de comentarios

- Comentarios en una sola línea *// comentario*
- Comentarios de varias líneas */\* comentario \*/*
- Comentarios de documentación *\*\* comentario \*/*

Como podemos observar un comentario en varias líneas es un bloque de texto situado entre el símbolo de comienzo del bloque */\**, y otro de terminación del mismo *\*/*. Ejemplo:

```
/*-----|
| (C) Angel Franco García |
| fecha: Marzo 1999      |
| programa: PrimeroApp.java |
|-----*/
```

Los comentarios de documentación es un bloque de texto situado entre el símbolo de comienzo del bloque *\*\**, y otro de terminación del mismo *\*/*.

Ejemplo:

```
** Este es el primer programa de una
serie dedicada a explicar los fundamentos del lenguaje Java */
```

Comentarios en una sola línea comienzan por *//*. En la ventana de edición del Entorno Integrado de Desarrollo (IDE) los comentarios se distinguen del resto del código por el color del texto. Ejemplo:

```
public class PrimeroApp{
    public static void main(String[] args) {
//imprime un mensaje
        System.out.println("El primer programa");
    }
}
```



## Sentencias

Una sentencia es una orden que se le da al programa para realizar una tarea específica, estas pueden ser:

```
int i=1; // declarar una variable e inicializarla
import java.awt.*; // importar clases
System.out.println("El primer programa"); // mostrar un mensaje en pantalla
rect.mover(10, 20); // llamar a una función
```

Las sentencias acaban con ; este carácter separa una sentencia de la siguiente.

## Bloques de código

Un bloque de código es un grupo de sentencias que se comportan como una unidad. Un bloque de código está limitado por las llaves de apertura { y cierre }.

## Expresiones

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

```
x=123;
y=(x+100)/4;
area=circulo.calcularArea(2.5);
Rectangulo r=new Rectangulo(10, 10, 200, 300);
```

La primera expresión asigna un valor a la variable *x*.

La segunda, realiza una operación .

La tercera, es una llamada a una función miembro *calcularArea* desde un objeto *circulo* de una clase determinada .

La cuarta, reserva espacio en memoria para un objeto de la clase *Rectangulo* mediante la llamada a una función especial denominada constructor.

## Variables

Una variable es un nombre que se asocia con una porción de la memoria del ordenador, en la que se guarda el valor asignado a dicha variable.

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

Java tiene tres tipos de variables:

- de instancia
- de clase
- locales

**Las variables de instancia o miembros dato:** se usan para guardar los atributos de un objeto particular.

**Las variables de clase o miembros dato estáticos :** son similares a las variables de instancia, con la excepción de que los valores que guardan son los mismos para todos los objetos de una determinada clase.

En el siguiente ejemplo, *PI* es una variable de clase y *radio* es una variable de instancia. *PI* guarda el mismo valor para todos los objetos de la clase *Circulo*, pero el radio de cada círculo (objeto) puede ser diferente.

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

**Las variables locales :** se utilizan dentro de las funciones miembro o métodos.

En el siguiente ejemplo *area* es una variable local a la función *calcularArea* en la que se guarda el valor del área de un objeto de la clase *Circulo*.

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

Las variables locales se declaran en el momento en el que son necesarias. También se pueden inicializar en el momento en el que son declaradas.

```
int x=0;
String nombre="Angel";
double a=3.5, b=0.0, c=-2.4;
boolean bNuevo=true;
int[] datos;
```

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

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

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

- Declarar
- Inicializar
- Usar

**int i; // declarar**

**int i=2; // inicializar**

**i+i; // usar**

## Tipos de datos primitivos

Tipo	Descripción
<b>boolean</b>	Tiene dos valores <b>true</b> o <b>false</b> .
<b>char</b>	Caracteres Unicode de 16 bits. Los caracteres alfa-numéricos son los mismos que los ASCII con el bit alto puesto a 0. El intervalo de valores va desde 0 hasta 65535 (valores de 16-bits sin signo).
<b>byte</b>	Tamaño 8 bits. El intervalo de valores va desde $-2^7$ hasta $2^7 - 1$ (-128 a 127)
<b>short</b>	Tamaño 16 bits. El intervalo de valores va desde $-2^{15}$ hasta $2^{15} - 1$ (-32768 a 32767)
<b>int</b>	Tamaño 32 bits. El intervalo de valores va desde $-2^{31}$ hasta $2^{31} - 1$ (-2147483648 a 2147483647)
<b>long</b>	Tamaño 64 bits. El intervalo de valores va desde $-2^{63}$ hasta $2^{63} - 1$ (-9223372036854775808 a 9223372036854775807)
<b>float</b>	Tamaño 32 bits. Números en coma flotante de simple precisión. Estándar IEEE 754-1985 (de 1.40239846e-45f a 3.40282347e+38f)
<b>double</b>	Tamaño 64 bits. Números en coma flotante de doble precisión. Estándar IEEE 754-1985. (de 4.94065645841246544e-324d a 1.7976931348623157e+308d.)

Los tipos básicos que utilizaremos en la mayor parte de los programas serán **boolean**, **int** y **double**.

## Caracteres

Un carácter está siempre rodeado de comillas simples como 'A', '9', 'ñ', etc. El tipo de dato **char** sirve para guardar estos caracteres.

Un tipo especial de carácter es la **secuencia de escape** que se utilizan para representar caracteres de control o caracteres que no se imprimen.

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

## Variables booleanas

Una variable booleana solamente puede guardar uno de los dos posibles valores: true (verdadero) y false (falso).

```
boolean encontrado=false;
{...}
encontrado=true;
```

## Variables enteras

Una variable entera consiste es cualquier cifra precedida por el signo más (opcional), para los positivos, o el signo menos, para los negativos. Son ejemplos de números enteros:

12, -36, 0, 4687, -3598

Como ejemplos de declaración de variable enteras tenemos:

```
int numero=1205;
int x,y;
long m=30L;
```

En la tercera línea 30 es un número de tipo **int** por defecto, le ponemos el sufijo **L** en mayúsculas o minúsculas para indicar que es de tipo **long**. Existen como vemos en la tabla varios tipos de números enteros ( **byte**, **short**, **int**, **long**).

## Variables en coma flotante

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

```
double PI=3.14159;
double g=9.7805, c=2.9979e8;
```

## Valores constantes

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

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

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

## Las cadenas de caracteres o strings

Las cadenas de caracteres o strings en Java son objetos de la clase *String*.

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

Empleando strings, el primer programa quedaría de la forma equivalente

```
public class PrimeroApp{
    public static void main(String[] args) {
//imprime un mensaje
        String mensaje="El primer programa";
        System.out.println(mensaje);
    }
}
```

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

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

## Palabras reservadas

En el siguiente cuadro se listan las palabras reservadas, aquellas que emplea el lenguaje Java, y que el programador no puede utilizar como identificadores (nombre de una variable).

Las palabras reservadas se pueden clasificar en las siguientes categorías:

- Tipos de datos: **boolean, float, double, int, char**
- Sentencias condicionales: **if, else, switch**
- Sentencias iterativas: **for, do, while, continue**
- Tratamiento de las excepciones: **try, catch, finally, throw**
- Estructura de datos: **class, interface, implements, extends**
- Modificadores y control de acceso: **public, private, protected, transient**
- Otras: **super, null, this.**

# OPERADORES

Todos los lenguajes de programación permiten realizar operaciones entre los tipos de datos básicos: suma, resta, producto, cociente, etc.

## Los operadores aritméticos

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

Operador	Nombre	Ejemplo
+	Suma	3+4
-	Diferencia	3-4
*	Producto	3*4
/	Cociente	20/7
%	Módulo	20%7

El cociente entre dos enteros da como resultado un entero. Por ejemplo, al dividir 20 entre 7 nos da como resultado 2.

El operador módulo da como resultado el resto de la división entera. Por ejemplo 20%7 da como resultado 6 que es el resto de la división entre 20 y 7.

## El operador asignación

Nos habremos dado cuenta que el operador más importante y más frecuentemente usado es el operador asignación `=`, que hemos empleado para la inicialización de las variables. Así,

```
int numero;
numero=20;
```

la primera sentencia declara una variable entera de tipo `int` y le da un nombre (`numero`). La segunda sentencia usa el operador asignación para inicializar la variable con el número 20.

Consideremos ahora, la siguiente sentencia.

```
a=b;
```

que asigna a `a` el valor de `b`. A la izquierda siempre tendremos una variable tal como `a`, que recibe valores, a la derecha otra variable `b`, o expresión que tiene un valor.

Por tanto, tienen sentido las expresiones

```
a=1234;
double area=calculaArea(radio);
superficie=ancho*alto;
```

Sin embargo, no tienen sentido las expresiones

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

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

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

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

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

El operador asignación se puede combinar con los operadores aritméticos

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

Así, la sentencia

```
x=x+23;    equivale a    x+=23;
```

## Concatenación de strings

En Java se usa el operador + para concatenar cadenas de caracteres o strings. Veremos en el siguiente apartado una sentencia como la siguiente:

```
System.out.println("la temperatura centígrada es "+tC);
```

El operador + cuando se utiliza con strings y otros objetos, crea un solo string que contiene la concatenación de todos sus operandos. Si alguno de los operandos no es una cadena, se convierte automáticamente en una cadena.

Por ejemplo, en la sentencia anterior el número del tipo **double** que guarda la variable *tC* se convierte en un string que se añade al string "la temperatura centígrada es ".

Como veremos más adelante, un objeto se convierte automáticamente en un string si su clase redefine la función miembro [toString](#) de la clase base *Object*.

Como vemos en el listado, para mostrar un resultado de una operación, por ejemplo, la suma de dos números enteros, escribimos

```
iSuma=ia+ib;
System.out.println("El resultado de la suma es "+iSuma);
```

Concatena una cadena de caracteres con un tipo básico de dato, que se convierte automáticamente en un string.

El operador += también funciona con cadenas.

```
String nombre="Juan ";
nombre+="García";
System.out.println(nombre);
```



```

public class OperadorAp {
    public static void main(String[] args) {
        System.out.println("Operaciones con enteros");
        int ia=7, ib=3;
        int iSuma, iResto;
        iSuma=ia+ib;
        System.out.println("El resultado de la suma es "+iSuma);
        int iProducto=ia*ib;
        System.out.println("El resultado del producto es "+iProducto);
        System.out.println("El resultado del cociente es "+(ia/ib));
        iResto=ia%ib;
        System.out.println("El resto de la división entera es "+iResto);

        System.out.println("*****");
        System.out.println("Operaciones con números decimales");
        double da=7.5, db=3.0;
        double dSuma=da+db;
        System.out.println("El resultado de la suma es "+dSuma);
        double dProducto=da*db;
        System.out.println("El resultado del producto es "+dProducto);
        double dCociente=da/db;
        System.out.println("El resultado del cociente es "+dCociente);
        double dResto=da%db;
        System.out.println("El resto de la división es "+dResto);
    }
}

```

### La precedencia de operadores

El lector conocerá que los operadores aritméticos tienen distinta precedencia, así la expresión

$$a+b*c$$

es equivalente a

$$a+(b*c)$$

ya que el producto y el cociente tienen mayor precedencia que la suma o la resta.

Sin embargo, si queremos que se efectúe antes la suma que la multiplicación tenemos de emplear los paréntesis

$$(a+b)*c$$

Para realizar la operación  $\frac{a}{bc}$  escribiremos

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

Programa que convierte una temperatura en grados Fahrenheit en Celsius

```
public class PrecedeApp {
    public static void main(String[] args) {
        int tF=80;
        System.out.println("la temperatura Fahrenheit es "+tF);
        int tC=(tF-32)*5/9;
        System.out.println("la temperatura centígrada es "+tC);
    }
}
```

### La conversión automática y promoción (casting)

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

```
int a=5;
double b=3.2;
double suma=a+b;
```

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

```
int entero;
double real=3.20567;
```

¿qué ocurrirá cuando asignamos a la variable *entero* el número guardado en la variable *real*?. Como hemos visto se trata de dos tipos de variables distintos cuyo tamaño en memoria es de 32 y 64 bits respectivamente. Por tanto, la sentencia

```
entero=real;
```

convierte el número real en un número entero eliminando los decimales. La variable *entero* guardará el número 3.

Supongamos que deseamos calcular la división 7/3, como hemos visto, el resultado de la división entera es 2, aún en el caso de que tratemos de guardar el resultado en una variable del tipo **double**, como lo prueba la siguiente porción de código.

```
int ia=7;
int ib=3;
double dc=ia/ib;
```

Si queremos obtener una aproximación decimal del número 7/3, hemos de promocionar el entero *ia* a un número en coma flotante, mediante un procedimiento denominado promoción o *casting*.

```
int ia=7;
int ib=3;
double dc=(double)ia/ib;
```

Como aplicación, consideremos el cálculo del valor medio de dos o más números enteros

```
int edad1=10;
int edad2=15;
double media=(double)(edad1+edad2)/2;
```

El valor medio de 10 y 15 es 12.5, sin la promoción se obtendría el valor erróneo 12.

Imaginemos ahora, una función que devuelve un entero **int** y queremos guardarlo en una variable de tipo **float**. Escribiremos

```
float resultado=(float)retornaInt();
```

## Los operadores unarios

Los operadores unarios son:

- ++ Incremento
- -- Decremento

actúan sobre un único operando y el resultado de la operación depende de que el operador esté a la derecha  $i++$  o a la izquierda  $++i$ .

Conoceremos, primero el significado de estos dos operadores a partir de las sentencias equivalentes:

```
i=i+1;      //añadir 1 a i
i++;
```

Del mismo modo, lo son

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

Examinemos ahora, la posición del operador respecto del operando.

Consideremos en primer lugar que el operador unario ++ está a la derecha del operando. La sentencia

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

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

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

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

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

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

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

```
public class UnarioApp {
    public static void main(String[] args) {
        int i=8;
        int a, b, c;
        System.out.println("\ntantes\tdurante\tdespués");
        i=8; a=i; b=i++; c=i;
        System.out.println("i++\t"+a+"\t"+b+"\t"+c);
        i=8; a=i; b=i--; c=i;
        System.out.println("i--\t"+a+"\t"+b+"\t"+c);

        i=8; a=i; b=++i; c=i;
        System.out.println("++i\t"+a+"\t"+b+"\t"+c);
        i=8; a=i; b=--i; c=i;
        System.out.println("--i\t"+a+"\t"+b+"\t"+c);
    }
}
```

La salida del programa es, la siguiente

	antes	durante	después	
i++	8	8	9	9
i--	8	8	7	7
++i	8	9	9	9
--i	8	7	7	7

La primera columna (antes) muestra el valor inicial de *i*, la segunda columna (durante) muestra el valor de la expresión, y la última columna (después) muestra el valor final de *i*, después de evaluarse la expresión.

Se deberá de tener siempre el cuidado de inicializar la variable, antes de utilizar los operadores unarios con dicha variable.

### Los operadores relacionales

Los operadores relacionales son símbolos que se usan para comparar dos valores. Si el resultado de la comparación es correcto la expresión considerada es verdadera, en caso contrario es falsa.

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

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

En el programa se compara la variable `i` que guarda un 8, con un conjunto de valores, el resultado de la comparación es verdadero (**true**), o falso (**false**).

```
public class RelacionApp {
    public static void main(String[] args) {
        int x=8;
        int y=5;
        boolean compara=(x<y);
        System.out.println("x<y es "+compara);
        compara=(x>y);
        System.out.println("x>y es "+compara);
        compara=(x==y);
        System.out.println("x==y es "+compara);
        compara=(x!=y);
        System.out.println("x!=y es "+compara);
        compara=(x<=y);
        System.out.println("x<=y es "+compara);
        compara=(x>=y);
        System.out.println("x>=y es "+compara);
    }
}
```

## Los operadores lógicos

Los operadores lógicos son:

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

AND y OR trabajan con dos operandos y retornan un valor lógico basadas en las denominadas tablas de verdad. El operador NOT actúa sobre un operando. Estas tablas de verdad son conocidas y usadas en el contexto de la vida diaria, por ejemplo: "si hace sol Y tengo tiempo, iré a la playa", "si NO hace sol, me quedaré en casa", "si llueve O hace viento, iré al cine". Las tablas de verdad de los operadores AND, OR y NOT se muestran en las tablas siguientes

El operador lógico AND

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

## El operador lógico OR

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

## El operador lógico NOT

x	resultado
true	false
false	true

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

$$(a < b) \ \&\& \ (b < c)$$

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

$$(a < b) \ || \ (b < c)$$

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

La expresión "NO  $a$  es menor que  $b$ "

$$!(a < b)$$

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

$$(a \geq b)$$

La expresión "NO  $a$  es igual a  $b$ "

$$!(a == b)$$

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

$$(a != b)$$

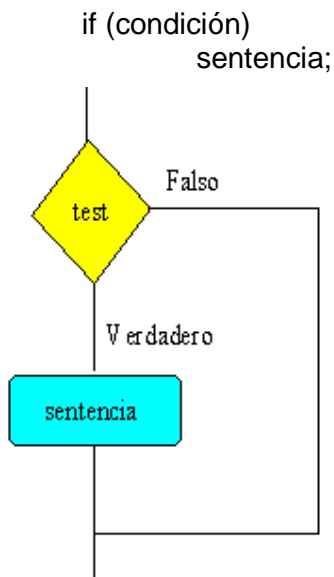
# SENTENCIAS DE CONTROL DE FLUJO

## Sentencias Condicionales

Del mismo modo que en la vida diaria, en un programa es necesario tomar decisiones basadas en ciertos hechos y actuar en consecuencia. El lenguaje Java tiene una sentencia básica denominada **if** (si condicional) que realiza un test y permite responder de acuerdo al resultado.

### La sentencia *if*

Si la condición es verdadera, la sentencia se ejecuta, de otro modo, se salta dicha sentencia, continuando la ejecución de l programa saltando a otras sentencias . La forma general de la sentencia **if** es:



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

```

if (condición){
    sentencias;
}

```

En el siguiente ejemplo, si el número del boleto que hemos adquirido coincide con el número aparecido en el sorteo, nos dicen que hemos obtenido un premio.

```

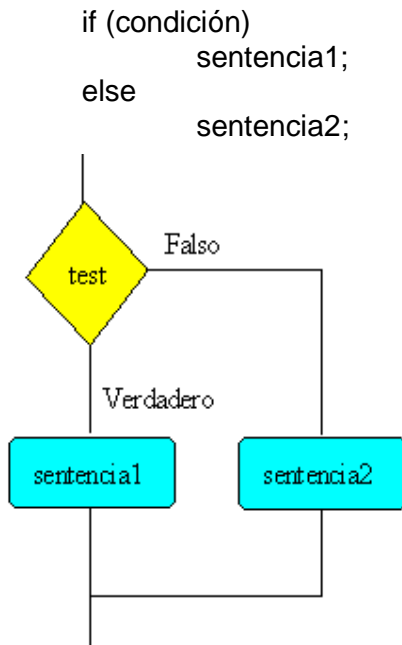
if(numeroBoleto==numeroSorteo)
    System.out.println("has obtenido un premio");

```



## La sentencia *if...else*

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



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

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

```

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

```

Existe una forma abreviada de escribir una sentencia condicional **if...else** como la siguiente:

```

if (numeroBoleto==numeroSorteo)
    premio=1000;
else
    premio=0;

```

en una sola línea

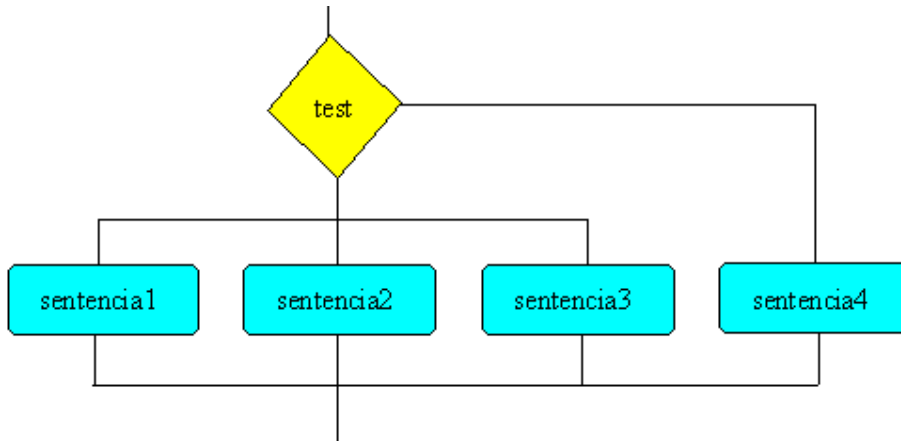
```

premio=(numeroBoleto==numeroSorteo) ? 1000 : 0;

```

## La sentencia *switch*

A veces, es necesario, elegir entre varias alternativas, como se muestra en la siguiente figura



Considerando la siguiente serie de sentencias **if...else**

```

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

El código resultante puede ser difícil de seguir .

El lenguaje Java proporciona una solución elegante a este problema mediante la sentencia condicional **switch** para agrupar a un conjunto de sentencias **if...else**.

```

switch(expresion){
    case valor1:
        sentencia1;
        break; //sale de switch
    case valor2:
        sentencia2;
        break; //sale switch
    case valor3:
        sentencia3;
        break; //sale de switch
    default:
        sentencia4;
}
  
```

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

Si el compilador no encuentra coincidencia, se ejecuta la sentencia **default**, si es que está presente en el código.

Veamos ahora un ejemplo sencillo: dado el número que identifica al mes (del 1 al 12) imprimir el nombre del mes.

```
public class SwitchApp1 {
    public static void main(String[] args) {
        int mes=3;
        switch (mes) {
            case 1: System.out.println("Enero"); break;
            case 2: System.out.println("Febrero"); break;
            case 3: System.out.println("Marzo"); break;
            case 4: System.out.println("Abril"); break;
            case 5: System.out.println("Mayo"); break;
            case 6: System.out.println("Junio"); break;
            case 7: System.out.println("Julio"); break;
            case 8: System.out.println("Agosto"); break;
            case 9: System.out.println("Septiembre"); break;
            case 10: System.out.println("Octubre"); break;
            case 11: System.out.println("Noviembre"); break;
            case 12: System.out.println("Diciembre"); break;
            default: System.out.println("Este mes no existe"); break;
        }
    }
}
```

Ahora un ejemplo más complicado, un programa que calcule el número de días de un mes determinado cuando se da el año.

Anotar primero, los meses que tienen 31 días y los que tienen 30 días. El mes de Febrero (2º mes) es el más complicado ya que tiene 28 días excepto en los años que son bisiestos que tiene 29. Son bisiestos los años múltiplos de cuatro, que no sean múltiplos de 100, pero si son bisiestos los múltiplos de 400.

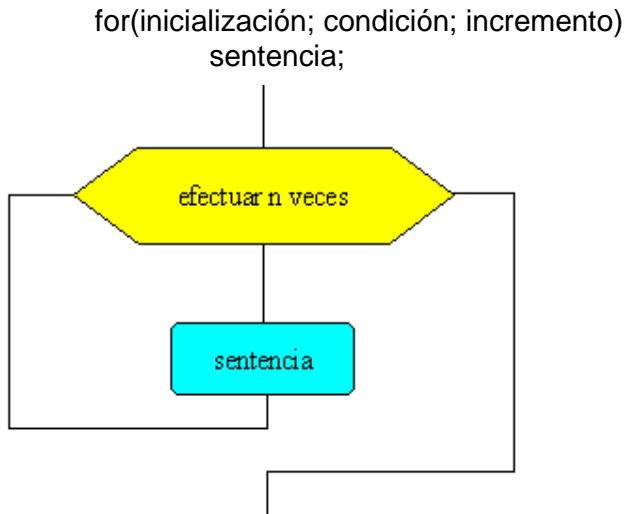
```
public class SwitchApp2 {
    public static void main(String[] args) {
        int mes=2;
        int año=1992;
        int numDias=30;
        switch (mes) {
            case 1:
            case 3:
            case 5:
            case 7:
            case 8:
            case 10:
            case 12:
                numDias = 31;
                break;
            case 4:
            case 6:
            case 9:
            case 11:
                numDias = 30;
                break;
            case 2:
                if ( ((año % 4 == 0) && !(año % 100 == 0)) || (año % 400 == 0) )
                    numDias = 29;
                else
                    numDias = 28;
                break;
            default:
                System.out.println("Este mes no existe");
                break;
        }
        System.out.println("El mes "+mes+" del año "+año+" tiene "+numDias+"
días");
    }
}
```

## Sentencias Iterativas

El lenguaje Java, como la mayoría de los lenguajes, proporciona sentencias que permiten realizar una tarea una y otra vez hasta que se cumpla una determinada condición, dicha tarea viene definida por un conjunto de sentencias agrupadas en un bloque. Las sentencias iterativas son **for**, **while** y **do...while**

### La sentencia **for**

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



El primer término *inicialización*, se usa para inicializar una variable índice, que controla el número de veces que se ejecutará el bucle.

La *condición* representa la condición que ha de ser satisfecha para que el bucle continúe su ejecución.

El *incremento* representa la cantidad que se incrementa la variable índice en cada repetición.

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

```

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

```

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

Ejemplo: Escribir un programa que imprima los números pares positivos menores o iguales que 20

```
for (int i=2; i <=20; i += 2) {
    System.out.println(i);
}
```

Ejemplo: Escribir un programa que imprima los números pares positivos menores o iguales que 20 en orden decreciente

```
for (int i=20; i >= 2; i -= 2) {
    System.out.println(i);
}
```

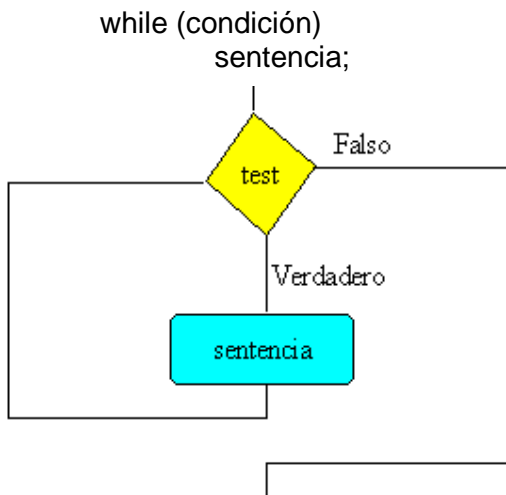
Ejemplo: Escribir un programa que calcule el factorial de un número empleando la sentencia iterativa **for**. Guardar el resultado en un número entero de tipo **long**.

Definición: el factorial de un número  $n$  es el resultado del producto  $1*2*3*...*(n-1)*n$ .

```
public class FactorialApp {
    public static void main(String[] args) {
        int numero=4;
        long resultado=1;
        for(int i=1; i<=numero; i++){
            resultado*=i;
        }
        System.out.println("El factorial es "+resultado);
    }
}
```

### La sentencia *while*

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



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

```
int i=0;

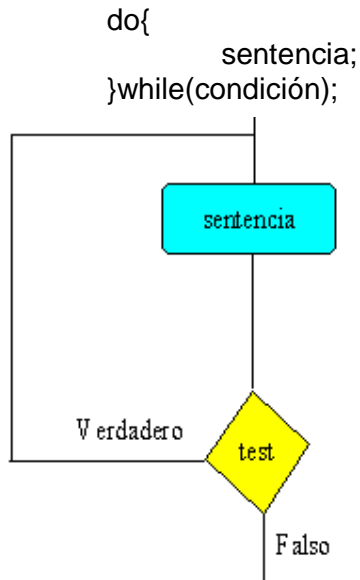
while (i<10) {
    System.out.println(i);
    i++;
}
```

Ejemplo: escribir un programa que calcule el factorial de un número empleando la sentencia iterativa **while**

```
public class FactorialApp1 {
    public static void main(String[] args) {
        int numero=4;
        long resultado=1;
        while(numero>0){
            resultado*=numero;
            numero--;
        }
        System.out.println("El factorial es "+resultado);
    }
}
```

### La sentencia **do...while**

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



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

```
int i=0;

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

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

### La sentencia *break*

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

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

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

El código anterior es equivalente a

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

### La sentencia *continue*

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

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



## Etiquetas

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

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

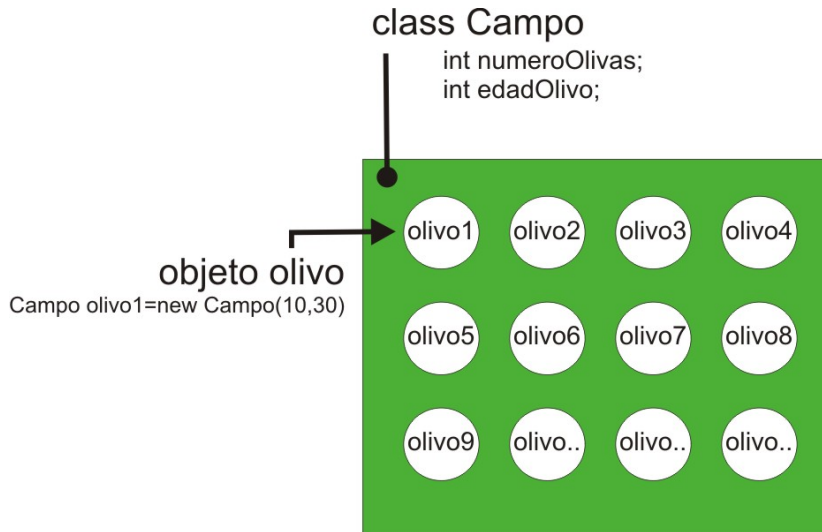
```
salida: for(int i=0; i<20; i++){  
    while(j<70){  
        if(i*j==500)           continue salida;  
        //...  
    }  
    //...  
}
```

# **CONCEPTOS BÁSICOS DE PROGRAMACIÓN ORIENTADA A OBJETOS**

## Concepto

El paso más importante en la introducción a la programación orientada a objetos es comprender su concepto. Para ello pondremos un ejemplo para tratar de explicarlo.

Supongamos que tenemos un campo, este a su vez tendrá un número determinado de olivos con sus atributos (edad, número de olivas, etc.). Nuestro campo tendrá una función que será generar una serie de beneficios por la producción.



- La **clase** que agrupará los atributos y funciones se llamará Campo.  

```
class Campo
```
- Las **variables** que guardarán los atributos propios de cada objeto, también en llamadas variables de instancia o miembros de instancia serán.  

```
int numeroOlivas;  
int edadOlivo;
```
- Los **objetos** de la clase Campo serán cada uno de los olivos que pasarán el valor de los atributos para ser guardados en las variables a través de un constructor.  

```
Campo olivo1=new Campo(10,30) // 10 y 30 son los valores de los atributos del olivo1  
Campo olivo2=new Campo(15,35) // 15 y 35 son los valores de los atributos del olivo2  
.....etc
```
- Los **Constructores** de la clase recibirán el valor del atributo para asignarlo a las variables de instancia de la clase.  

```
Campo(edad,numOlivas){ // recibe los valores de los objetos  
    this.edadOlivo=edad; // los valores son asignados a las variables de instancia.  
    this.numeroOlivas=numOlivas;  
}
```
- Los **métodos o funciones** se podrá llamar desde el objeto.  

```
olivo1.beneficios(); // llamada a una función que retornará un resultado.
```

## Clase

Para crear una clase se utiliza la palabra reservada **class** y a continuación el nombre de la clase. La definición de la clase se pone entre las llaves de apertura y cierre. El nombre de la clase empieza por letra mayúscula.

```
class Campo{
    //miembros dato
    //constructores
    //funciones miembro
}
```

## Miembro dato

Los valores de los atributos se guardan en los miembros dato o variables de instancia. Los nombres de dichas variables comienzan por letra minúscula.

La clase denominada *Campo*, describe las características comunes a los olivos(objetos):

- La edad del olivo.
- El número de olivas del olivo.

```
class Campo{
    int numeroOlivas; // variable de instancia o miembro dato
    int edadOlivo; // variable de instancia o miembro dato

    // constructores
    // métodos o funciones miembro
}
```

## Constructor

Un objeto de una clase se crea llamando a una función especial denominada constructor de la clase. El constructor se llama de forma automática cuando se crea un objeto, para situarlo en memoria e inicializar los miembros dato declarados en la clase. El constructor tiene el mismo nombre que la clase. Lo específico del constructor es que no retorna ningún valor .

```
class Campo{
    int numeroOlivas; // variable de instancia o miembro dato
    int edadOlivo; // variable de instancia o miembro dato

    Campo(edad,numOlivas){ // recibe los valores de los objetos
        this.edadOlivo=edad; // los valores son asignados a las variables de instancia.
        this.numeroOlivas=numOlivas;
    }
}
```

El constructor recibe dos valores que guardan los parámetros (*edad, numOlivas*) y con ellos inicializa los miembros dato *edadOlivo* y *numeroOlivas*.

Una clase puede tener más de un constructor. Por ejemplo, el segundo constructor crea un olivo cuya edad es siempre 30 .

```
class Campo{
    int numeroOlivas; // variable de instancia o miembro dato
    int edadOlivo; // variable de instancia o miembro dato

    Campo(edad,numOlivas){ // recibe los valores de los objetos
        this.edadOlivo=edad; // los valores son asignados a las variables de instancia.
        this.numeroOlivas=numOlivas;
    }

    Campo(numOlivas){ // recibe los valores de los objetos
        this.edadOlivo=30; // los valores son asignados a las variables de instancia.
        this.numeroOlivas=numOlivas;
    }
}
```

## Método

En el lenguaje Java las funciones miembro o métodos se definen en la clase y se llaman mediante los objetos.

El nombre de las funciones miembro o métodos comienza por letra minúscula . La definición de una función tiene el siguiente formato:

```
tipo nombreFuncion(tipo parm1, tipo parm2, tipo parm3){
    //...sentencias
}
```

Entre las llaves de apertura y cierre se coloca la definición de la función. *tipo* indica el tipo de dato que puede ser **int**, **double**, etc

Para llamar a una función miembro o método se escribe

```
objeto.nombreFuncion(arg1, arg2, arg3);
```

o también:

```
retorno=objeto.nombreFuncion(arg1, arg2, arg3);
```

Cuando se llama a la función, los argumentos *arg1*, *arg2*, *arg3* se copian en los parámetros *parm1*, *parm2*, *parm3* y se ejecutan las sentencias dentro de la función. La función al finalizar el valor resultante es devuelto mediante la sentencia **return** y este valor resultante de retorno se puede asignar a una variable.

Cuando una función no devuelve nada se dice de tipo **void**.

Cualquier variable declarada dentro de la función tiene una vida temporal, existiendo en memoria, mientras la función esté activa. Se trata de variables locales a la función. Por ejemplo:

```
void nombreFuncion(int parm){
    //...
    int i=5;
    //...
}
```

```

class Campo{

    int numeroOlivas; // variable de instancia o miembro dato
    int edadOlivo; // variable de instancia o miembro dato
    static final double PVPOLIVA=0.05; // variable de clase

    Campo(edad,numOlivas){ // primer constructor recibe los valores de los objetos
        this.edadOlivo=edad; // los valores son asignados a las variables de instancia.
        this.numeroOlivas=numOlivas;
    }

    Campo(numOlivas){ // segundo constructor recibe los valores de los objetos
        this.edadOlivo=30; // los valores son asignados a las variables de instancia.
        this.numeroOlivas=numOlivas;
    }

    double beneficio(){ // función calculo beneficio
        dinero=numeroOlivas*PVPOLIVA;
        return dinero;
    }

    void sumaEdad (int incremento){ // función incremento edad del olivo no devuelve nada
        edadOlivo+=incremento;
    }

    void sumaOlivas (int sumaOlivas){ // función incremento de olivas no devuelve nada
        numeroOlivas+=sumaOlivas;
    }
}

```

- La función **beneficio** devuelve el total de multiplicar el valor de una oliva por el número de olivas. Devuelve un dato de tipo decimal (double)
- La función **sumaEdad** incrementa la edad del olivo en caso que tengamos que incrementarla. Esta función no devuelve nada por eso es de tipo void.
- La función **sumaOlivas** incrementa el número de olivas y también es de tipo void.

## Objeto

Para crear un objeto de una clase se usa la palabra reservada **new**.

Por ejemplo,

```
Campo olivo1=new Campo(10, 30);
```

**new** reserva espacio en memoria para los miembros dato y devuelve una referencia que se guarda en la variable *olivo1* del tipo *Campo* que denominamos ahora objeto.

El objeto denominado *olivo1* de la clase *Campo* llama al primer constructor en el listado. El olivo tendrá una edad de *edad=10* y un numero de olivas *numOlivas=30*.

```
Campo olivo2=new Campo(40);
```

Crea un objeto denominado *oliva2* de la clase *Campo* llamando al segundo constructor, dicho olivo tendrá un numero de olivas *numOlivas=40* y su edad ya la asigna automáticamente el constructor *edadOlivo=30* .

### Acceso a miembros dato o funciones miembro

Desde un objeto se puede **acceder a los miembros dato o variables de instancia** mediante la siguiente sintaxis

```
objeto.miembro;
```

Por ejemplo, podemos acceder al miembro dato *edadOlivo*, para cambiar la edad de un objeto olivo .

```
olivo1.edadOlivo=20;
```

El olivo *1* que tenía inicialmente una edad de 10, mediante esta sentencia se la cambiamos a 20.

Desde un objeto **llamamos a las funciones miembro** para realizar una determinada tarea.

Por ejemplo, desde *olivo1* llamamos a la función *beneficio* para obtener la cantidad de dinero del total de olivas del olivo *1* .

```
olivo1.beneficio();
```

La función miembro *beneficio* devuelve un decimal(double), que guardaremos en una variable decimal *beneficios1*, para luego usar este dato.

```
double beneficios1=olivo1.beneficio();  
System.out.println("El beneficio del primer olivo es: "+beneficios1);
```

Para incrementar la edad de un olivo *olivo2*, en 40 años.

```
Olivo2.sumaEdad(40);
```

Para incrementar el numero de olivas de un olivo *olivo2*, en 20 olivas.

```
Olivo2.sumaOlivas(20);
```

Veamos la aplicación como quedaría. Para la aplicación creamos otra clase llamada CampoAplicación.

```
public class CampoAplicación {
    public static void main(String[] args) { // función principal
        Campo olivo1=new Campo(10, 30); // objeto
        Campo olivo2=new Campo(40); // objeto

        System.out.println("La edad del olivo1 es: "+olivo1.edadOlivo);
        olivo1.edadOlivo=20; //cambio de edad del olivo1
        System.out.println("La nueva edad del olivo1 es: "+olivo1.edadOlivo);

        double beneficios1=olivo1.beneficio();
        System.out.println("El beneficio del primer olivo es: "+beneficios1);

        double beneficios2=olivo2.beneficio();
        System.out.println("El beneficio del segundo olivo es: "+beneficios2);

        System.out.println("La edad del olivo2 es: "+olivo2.edadOlivo);
        olivo2.sumaEdad(40); // incremento de la edad mediante funcion
        System.out.println("La nueva edad del olivo2 es: "+olivo2.edadOlivo);

        System.out.println("El número de olivas del olivo2 es: "+olivo2.numeroOlivas);
        olivo2.sumaOlivas(20); // incremento el numero de olivas
        System.out.println("El nuevo número de olivas del olivo2 es: "+olivo2.numeroOlivas);

    }
}
```



# PAQUETES (PACKAGE)

## El paquete (package)

- Los paquetes son una forma de organizar grupos de clases. Un paquete contiene un conjunto de clases relacionadas bien por finalidad, por ámbito o por herencia.
- Los paquetes resuelven el problema del conflicto entre los nombres de las clases. Al crecer el número de clases crece la probabilidad de designar con el mismo nombre a dos clases diferentes.
- Las clases tienen ciertos privilegios de acceso a los miembros de datos y a las funciones miembro de otras clases dentro de un mismo paquete.

Se pueden agregar nuevas clases al proyecto, todas ellas contenidas en archivos .java. La primera sentencia que encontramos en el código fuente de las distintas clases que forman el proyecto es **package** seguido del nombre del paquete.

```
//archivo MiAplicacion.java

package nombrePaquete;
public class MiApp{
    //miembros de datos
    //funciones miembro
}
```

```
//archivo MiClase.java

package nombrePaquete;
public class MiClase{
    //miembros de datos
    //funciones miembro
}
```

## El comando *import*

Para importar clases de un paquete se usa el comando **import**. Se puede importar una clase individual

```
import nombrePaquete.NombreClase;
```

o bien, se puede importar las clases declaradas públicas de un paquete completo, utilizando un asterisco (\*) para reemplazar los nombres de clase individuales.

```
import nombrePaquete.*;
```

Para crear un objeto de esa clase importada:

```
import nombrePaquete.NombreClase;
NombreClase objeto=new NombreClase();
```

```
import nombrePaquete.*;
NombreClase objeto=new NombreClase();
```

## Paquetes estándar

Java contiene una serie de paquetes estándar con sus respectivas clases que pueden importarse para usar sus métodos.

Paquete	Descripción
java.applet	Contiene las clases necesarias para crear applets que se ejecutan en la ventana del navegador web.
java.awt	Contiene clases para crear una aplicación guiada independiente de la plataforma
java.io	Entrada/Salida. Clases que definen distintos flujos de datos
java.lang	Contiene clases esenciales, se importa implícitamente sin necesidad de una sentencia <b>import</b> .
java.net	Se usa en combinación con las clases del paquete java.io para leer y escribir datos en la red.
java.util	Contiene otras clases útiles que ayudan al programador

# COMPOSICIÓN

Hay dos formas de reutilizar el código, mediante la composición y mediante la herencia. La composición significa utilizar objetos dentro de otros objetos.

Lo vamos a tratar a fondo mediante el estudio de un ejemplo.

### La clase *Punto*

La clase *Punto* tiene dos miembros dato, las coordenadas  $(x, y)$  de un punto del plano. Definimos dos constructores uno por defecto que sitúa el punto en el origen, y otro constructor explícito que proporciona las coordenadas  $x$  e  $y$  de un punto concreto.

```
public class Punto {
    int x;
    int y;
    //constructores
    //funciones miembro
}
```

El constructor de la clase *Punto*. La palabra reservada **this** apunta al miembro dato de la clase para asignarle el valor que se le pasa al constructor.

```
public Punto(int x, int y) {
    this.x = x;
    this.y = y;
}
```

La función miembro *desplazar* simplemente cambia la posición del punto desde  $(x, y)$  a  $(x+dx, y+dy)$ . La función *desplazar* cuando es llamada recibe en sus dos parámetros  $dx$  y  $dy$  el valor de desplazamiento del punto y actualiza las coordenadas  $x$  e  $y$  del punto. La función no retorna ningún valor.

```
public void desplazar(int dx, int dy){
    x+=dx;
    y+=dy;
}
```

Para crear un objeto de la clase *Punto* cuyas coordenadas  $x$  e  $y$  valgan respectivamente 10 y 23 escribimos

```
Punto p=new Punto(10, 23);
```

Para desplazar el punto  $p$  10 unidades hacia la izquierda y 40 hacia abajo, llamamos desde el objeto  $p$  a la función *desplazar* y le pasamos el desplazamiento horizontal y vertical.

```
p.desplazar(10, 40);
```

El código completo de la clase *Punto*, es el siguiente

```
public class Punto {
    int x = 0;
    int y = 0;

    public Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public Punto() {
        x=0;
        y=0;
    }
    void desplazar(int dx, int dy){
        x+=dx;
        y+=dy;
    }
}
```

### La clase *Rectangulo*

La clase *Rectangulo* tiene como miembros dato, el *origen* que es un objeto de la clase *Punto* y las dimensiones *ancho* y *alto*.

```
public class Rectangulo {
    Punto origen;
    int ancho ;
    int alto ;
    //constructores
    //funciones miembro
}
```

El constructor por defecto, crea un rectángulo situado en el punto 0,0 y con dimensiones nulas

```
    public Rectangulo() {
        this.origen = new Punto(0, 0);
        this.ancho=0;
        this.alto=0;
    }
```

El constructor explícito crea un rectángulo situado en un determinado punto *p* y con unas dimensiones que se le pasan en el constructor

```
    public Rectangulo(Punto p, int w, int h) {
        this.origen = p;
        this.ancho = w;
        this.alto = h;
    }
```

Podemos definir otros constructores en términos del constructor explícito usando la palabra reservada **this** que apuntará al constructor explícito.

```
public Rectangulo(Punto p) {
    this(p, 0, 0);
}
public Rectangulo(int w, int h) {
    this(new Punto(0, 0), w, h);
}
}
```

El primero crea un rectángulo de dimensiones nulas situado en el punto  $p$ . El segundo, crea un rectángulo de unas determinadas dimensiones situándolo en el punto 0, 0. Dentro del cuerpo de cada constructor se llama al constructor explícito mediante **this** pasándole en sus parámetros los valores apropiados.

Para desplazar un rectángulo, trasladamos su origen (esquina superior izquierda) a otra posición, sin cambiar su anchura o altura. Desde el objeto *origen* inicializado a través de los constructores, llamamos a la función *desplazar* de la clase *Punto*

```
void desplazar(int dx, int dy) {
    origen.desplazar(dx, dy);
}
}
```

El código completo de la nueva clase *Rectangulo*, es el siguiente.

```
public class Rectangulo {
    Punto origen;
    int ancho ;
    int alto ;

    public Rectangulo() {
        origen = new Punto(0, 0);
        ancho=0;
        alto=0;
    }
    public Rectangulo(Punto p) {
        this(p, 0, 0);
    }
    public Rectangulo(int w, int h) {
        this(new Punto(0, 0), w, h);
    }
    public Rectangulo(Punto p, int w, int h) {
        origen = p;
        ancho = w;
        alto = h;
    }
    void desplazar(int dx, int dy) {
        origen.desplazar(dx, dy);
    }
    int calcularArea() {
        return ancho * alto;
    }
}
}
```

## Objetos de la clase *Rectangulo*

Para crear un rectángulo *rect1* situado en el punto (0, 0) y cuyas dimensiones son 100 y 200 escribimos

```
Rectangulo rect1=new Rectangulo(100, 200);
```

Para crear un rectángulo *rect2*, situado en el punto de coordenadas 44, 70 y de dimensiones nulas escribimos

```
Punto p=new Punto(44, 70);  
Rectangulo rect2=new Rectangulo(p);
```

O bien, en una sólo línea

```
Rectangulo rect2=new Rectangulo(new Punto(44, 70));
```

Para desplazar el rectángulo *rect1* desde el punto (100, 200) a otro punto situado 40 unidades hacia la derecha y 20 hacia abajo, sin modificar sus dimensiones, escribimos

```
rect1.desplazar(40, 20);
```

Para hallar y mostrar el área del rectángulo *rect1* podemos escribir

```
System.out.println("el área es "+rect1.calcularArea());
```

Para hallar el área de un rectángulo de 100 unidades de largo y 50 de alto y guardar el resultado en la variable entera *areaRect*, escribimos en una sólo línea.

```
int areaRect=new Rectangulo(100, 50).calcularArea();
```

```
public class RectanguloApp {  
    public static void main(String[] args) {  
        Rectangulo rect1=new Rectangulo(100, 200);  
        Rectangulo rect2=new Rectangulo(new Punto(44, 70));  
        Rectangulo rect3=new Rectangulo();  
        rect1.desplazar(40, 20);  
        System.out.println("el área es "+rect1.calcularArea());  
        int areaRect=new Rectangulo(100, 50).calcularArea();  
        System.out.println("el área es "+areaRect);  
    }  
}
```



# LA CLASE STRING

## La clase *String*

Los strings u objetos de la clase *String* se pueden crear explícitamente o implícitamente.

- Para crear un string **implícitamente** basta poner una cadena de caracteres entre comillas dobles. Por ejemplo, cuando se escribe

```
System.out.println("El primer programa");
```

- Para crear un string **explícitamente** escribimos

```
String str=new String("El primer programa");
```

También se puede escribir, alternativamente

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

Para crear un **string nulo**. Un string nulo es aquél que no contiene caracteres, pero es un objeto de la clase *String*. se puede hacer de estas dos formas

```
String str="";  
String str=new String();
```

El objeto *str* de la clase *String* sin inicializar.

```
String str;
```

## Cómo se obtiene información acerca del string

Una vez creado un objeto de la clase *String* podemos obtener información relevante acerca del objeto a través de las funciones miembro.

Para obtener la longitud, número de caracteres que guarda un string se llama a la función miembro *length*.

```
String str="El primer programa";  
int longitud=str.length();
```

Podemos conocer si un string comienza con un determinado prefijo, llamando al método *startsWith*, que devuelve **true** o **false**, según que el string comience o no por dicho prefijo

```
String str="El primer programa";  
boolean resultado=str.startsWith("El");
```

En este ejemplo la variable resultado tomará el valor **true**.

De modo similar, podemos saber si un string finaliza con un conjunto dado de caracteres, mediante la función miembro *endsWith*.

```
String str="El primer programa";  
boolean resultado=str.endsWith("programa");
```

Si se quiere obtener la posición de la primera ocurrencia de la letra p, se usa la función *indexOf*.

```
String str="El primer programa";
int pos=str.indexOf('p');
```

Para obtener las sucesivas posiciones de la letra p, se llama a otra versión de la misma función

```
pos=str.indexOf('p', pos+1);
```

El segundo argumento le dice a la función *indexOf* que empiece a buscar la primera ocurrencia de la letra p a partir de la posición *pos+1*.

Otra versión de *indexOf* busca la primera ocurrencia de un substring (porción de string) dentro del string.

```
String str="El primer programa";
int pos=str.indexOf("pro");
```

## Comparación de strings

La comparación de strings nos da la oportunidad de distinguir entre el operador lógico **==** y la función miembro *equals* de la clase *String*. **equals** compara el contenido del mensaje. En el siguiente código

```
String str1="El lenguaje Java";
String str2=new String("El lenguaje Java");
if(str1==str2){
    System.out.println("Los mismos objetos");
}else{
    System.out.println("Distintos objetos");
}
if(str1.equals(str2)){
    System.out.println("El mismo contenido");
}else{
    System.out.println("Distinto contenido");
}
```

Esta porción de código devolverá que *str1* y *str2* son distintos objetos pero con el mismo contenido.

La función miembro **compareTo** devuelve un entero menor que cero si el objeto string es menor (en orden alfabético) que el string dado, cero si son iguales, y mayor que cero si el objeto string es mayor que el string dado.

```
String str="B";
int resultado=str.compareTo("A");
```

La variable entera *resultado* tomará un valor mayor que cero, ya que B está después de A en orden alfabético.

```
String str="A";
int resultado=str.compareTo("B");
```

La variable entera *resultado* tomará un valor menor que cero, ya que A está antes que B en orden alfabético.

## Extraer un substring de un string

En muchas ocasiones es necesario extraer una porción o substring de un string dado. Para este propósito hay una función miembro de la clase *String* denominada *substring*.

Para extraer un substring desde una posición determinada hasta el final del string escribimos

```
String str="El lenguaje Java";  
String subStr=str.substring(12);
```

Se obtendrá el substring "Java".

Una segunda versión de la función miembro *substring*, nos permite extraer un substring especificando la posición de comienzo y la el final.

```
String str="El lenguaje Java";  
String subStr=str.substring(3, 11);
```

Se obtendrá el substring "lenguaje". Recuérdese, que las posiciones se empiezan a contar desde cero.

## Convertir un número a string

Para convertir un número en string se emplea la función miembro estática *valueOf*

```
int valor=10;  
String str=String.valueOf(valor);
```

La clase *String* proporciona versiones de *valueOf* para convertir los datos primitivos: **int**, **long**, **float**, **double**.

## Convertir un string en número

Cuando introducimos caracteres en un control de edición a veces es inevitable que aparezcan espacios ya sea al comienzo o al final. Para eliminar estos espacios tenemos la función miembro *trim*

```
String str=" 12 ";  
String str1=str.trim();
```

Para convertir un string en número entero, primero quitamos los espacios en blanco al principio y al final y luego, llamamos a la función miembro estática *parseInt* de la clase *Integer*

```
String str=" 12 ";  
int numero=Integer.parseInt(str.trim());
```

Para convertir un string en número decimal ( **double** ) se requieren dos pasos: convertir el string en un objeto de la clase envolvente *Double*, mediante la función miembro estática *valueOf*, y a continuación convertir el objeto de la clase *Double* en un tipo primitivo **double** mediante la función *doubleValue*

```
String str="12.35 ";
double numero=Double.valueOf(str).doubleValue();
```

Se puede hacer el mismo procedimiento para convertir un string a número entero

```
String str="12";
int numero=Integer.valueOf(str).intValue();
```

### La clase *StringBuffer*

En la sección dedicada a los operadores hemos visto que es posible concatenar cadenas de caracteres, es, decir, objetos de la clase *String*. Ahora bien, los objetos de la clase *String* son constantes lo cual significa que por defecto, solamente se pueden crear y leer pero no se pueden modificar.

La clase *StringBuffer* nos permite crear objetos dinámicos, que pueden modificarse.

```
// ejemplo STRING BASICO inicializado desde un principio
// y que después es modificado agregando una cadena de texto
// extra a la ya inicializada

public class stringbasic {
    public static void main(String args[]){
        // la clase StringBuffer es para almacenar cadenas de texto en memoria que
        // que cambian con mucha frecuencia
        StringBuffer cadena_a_modificar= new StringBuffer("Paco");

        cadena_a_modificar.append(" Aragon");
        //con append se modifica objeto inicializado

        System.out.println(cadena_a_modificar);

    }
}
```

# ARRAYS

Un array es un medio de guardar un conjunto de objetos de la misma clase.

### Declarar y crear un array

Para declarar un array se escribe

```
tipo_de_dato[] nombre_del_array;
```

Para declarar un array de enteros escribimos

```
int[] numeros;
```

Para crear un array de 4 número enteros escribimos

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

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

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

### Inicializar y usar los elementos del array

Para inicializar el array de 4 enteros escribimos

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

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

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

No necesitamos recordar el número de elementos del array, su miembro dato **length** nos proporciona la dimensión del array. Escribimos de forma equivalente

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

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

```
int[] numeros={2, -4, 15, -25};  
String[] nombres={"Juan", "José", "Miguel", "Antonio"};
```

Para imprimir a los elementos de array *nombres* se escribe

```
for(int i=0; i<nombres.length; i++){  
    System.out.println(nombres[i]);  
}
```

Para crear un array de tres objetos de la clase *Rectangulo* se escribe

- Declarar
 

```
Rectangulo[] rectangulos;
```
- Crear el array
 

```
rectangulos=new Rectangulo[3];
```
- Inicializar los elementos del array
 

```
rectangulos[0]=new Rectangulo(10, 20, 30, 40);
rectangulos[1]=new Rectangulo(30, 40);
rectangulos[2]=new Rectangulo(50, 80);
```

O bien, en una sola línea

```
Rectangulo[] rectangulos={new Rectangulo(10, 20, 30, 40),
                           new Rectangulo(30, 40), new Rectangulo(50, 80)};
```
- Usar el array
 

Para calcular y mostrar el área de los rectángulos escribimos

```
for(int i=0; i<rectangulos.length; i++){
    System.out.println(rectangulos[i].calcularArea());
}
```

## Arrays multidimensionales

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

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

- La primer fila tiene cuatro elementos {1,2,3,4}
- La segunda fila tiene dos elementos {5,6}
- La tercera fila tiene seis elementos {7,8,9,10,11,12}
- La cuarta fila tiene un elemento {13}

Para mostrar los elementos de este array bidimensional escribimos el siguiente código

```
for (int i=0; i < matriz.length; i++) {
    for (int j=0; j < matriz[i].length; j++) {
        System.out.print(matriz[i][j]+" ");
    }
    System.out.println("");
}
```

Como podemos apreciar, *matriz.length* nos proporciona el número de filas (cuatro), y *matriz[i].length*, nos proporciona el número de elementos en cada fila.

Mostramos los elementos de una fila separados por un tabulador. Una vez completada una fila se pasa a la siguiente mediante *println*.



Queremos crear y mostrar una matriz cuadrada unidad de dimensión 4. Una matriz unidad es aquella cuyos elementos son ceros excepto los de la diagonal principal  $i=j$ , que son unos. Mediante un doble bucle **for** recorreremos los elementos de la matriz especificando su fila  $i$  y su columna  $j$ . En el siguiente programa

- Se crea una matriz cuadrada de dimensión cuatro
- Se inicializa los elementos de la matriz (matriz unidad)
- Se muestra la matriz una fila debajo de la otra separando los elementos de una fila por tabuladores.

```
public class MatrizUnidadApp {
    public static void main (String[] args) {
        double[][] mUnidad= new double[4][4];

        for (int i=0; i < mUnidad.length; i++) {
            for (int j=0; j < mUnidad[i].length; j++) {
                if (i == j) {
                    mUnidad[i][j]=1.0;
                }else {
                    mUnidad[i][j] = 0.0;
                }
            }
        }

        for (int i=0; i < mUnidad.length; i++) {
            for (int j=0; j < mUnidad[i].length; j++) {
                System.out.print(mUnidad[i][j]+" ");
            }
            System.out.println("");
        }
    }
}
```

Un ejemplo del uso de [break](#) con etiqueta y arrays multidimensionales .

Busca un número de una matriz y muestra su posición en el array.

```
int[][] matriz={{32, 87, 3, 589},{12, -30, 190, 0},{622, 127, 981, -3, -5}};
int numero=12;
int i=0, j=0;

buscado:
for(i=0; i<matriz.length; i++){
    for(j=0; j<matriz[i].length; j++){
        if(matriz[i][j]==numero) {
            break buscado;
        }
    }
}
System.out.println("buscado: matriz["+ i+ ", "+j+"]="+matriz[i][j]);
```

# LA CLASE RANDOM

## Importar y crear objetos de la clase *Random*

La clase *Random* proporciona un generador de números aleatorios.

Para crear una secuencia de números aleatorios tenemos que seguir los siguientes pasos:

1. Proporcionar a nuestro programa información acerca de la clase *Random*. Al principio del programa escribiremos la siguiente sentencia.

```
import java.util.Random;
```

2. Crear un objeto de la clase *Random*

```
Random rnd = new Random();
```

3. Llamar a una de las funciones miembro que generan un número aleatorio
4. Usar el número aleatorio.

## Constructores

La clase dispone de dos constructores, el primero crea un generador de números aleatorios.

```
Random rnd = new Random();
```

El segundo, inicializa la generación con un número del tipo **long**.

```
Random rnd = new Random(3816L);
```

El sufijo L no es necesario, ya que aunque 3816 es un número **int** por defecto, es promocionado automáticamente a **long**.

## Funciones miembro

Podemos cambiar la inicialización de generación de los números aleatorios en cualquier momento, llamando a la función miembro *setSeed*.

```
rnd.setSeed(3816);
```

Podemos generar números aleatorios en cuatro formas diferentes:

genera un número aleatorio entero de tipo **int**

```
rnd.nextInt();
```

genera un número aleatorio entero de tipo **long**

```
rnd.nextLong();
```

genera un número aleatorio de tipo **float** entre 0.0 y 1.0, aunque siempre menor que 1.0

```
rnd.nextFloat();
```

genera un número aleatorio de tipo **double** entre 0.0 y 1.0, aunque siempre menor que 1.0

```
rnd.nextDouble();
```

Para generar una secuencia de 10 números aleatorios entre 0.0 y 1.0 escribimos

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

Para crear una secuencia de 10 números aleatorios enteros comprendidos entre 0 y 9 ambos incluidos escribimos

```
int x;
for (int i = 0; i < 10; i++) {
    x = (int)(rnd.nextDouble() * 10.0);
    System.out.println(x);
}
```

*(int)* transforma un número decimal **double** en entero **int** eliminando la parte decimal.

### Comprobación de la uniformidad de los números aleatorios

Podemos comprobar la uniformidad de los números aleatorios generando una secuencia muy grande de números aleatorios enteros comprendidos entre 0 y 9 ambos inclusive. Contamos cuantos ceros aparecen en la secuencia, cuantos unos, ... cuantos nueves, y guardamos estos datos en los elementos de un array.

Primero creamos un array *ndigitos* de 10 de elementos que son enteros.

```
int[] ndigitos = new int[10];
```

Inicializamos los elementos del array a cero.

```
for (int i = 0; i < 10; i++) {
    ndigitos[i] = 0;
}
```

Creamos una secuencia de 100000 números aleatorios enteros comprendidos entre 0 y 9 ambos inclusive (véase el apartado anterior)

```
for (long i=0; i < 100000L; i++) {
    n = (int)(rnd.nextDouble() * 10.0);
    ndigitos[n]++;
}
```

Si *n* sale cero suma una unidad al contador de ceros, *ndigitos[0]*. Si *n* sale uno, suma una unidad al contador de unos, *ndigitos[1]*, y así sucesivamente.

Finalmente, se imprime el resultado, los números que guardan cada uno de los elementos del array *ndigitos*

```
    for (int i = 0; i < 10; i++) {
        System.out.println(i+": " + ndigitos[i]);
    }
}
```

Observaremos en la consola que cada número 0, 1, 2...9 aparece aproximadamente 10000 veces.

```
package azar;

import java.util.Random;

/**
 * programa que cuenta cuantas veces sale cada numero del intervalo (0 -2)
 * de 100 veces que se consigue un numero aleatorio de ese intervalo
 */
public class Azaruniformidad {
    public static void main(String[] args) {
        Random aleatorio = new Random();
        int[] ndigitos = new int[3]; // declarado y creado array
        /**
         * la dimension del array tiene que ser igual al numero de
         * valores diferentes que pueden salir del generador de numeros aleatorios
         * en este caso son solo 3 numero diferentes que pueden llegar a salir(0 -1-2)
         */

        for (int i = 0; i < ndigitos.length; i++) {
            ndigitos[i] = 0;
        }
        /**
         * inicializado array mediante bucle for
         * inicializado cada uno de los 3 elementos a cero(contador)
         * los 3 numeros que comprenden el intervalo(0-2) del generador
         * de numero aleatorio
         */

        for (int i = 0; i < 100; i++) {
            int n = (int) (aleatorio.nextDouble() * 3.0);
            ndigitos[n]++;
        }
        /**
         * bucle que genera 100 numeros aleatorios comprendidos entre (0-2)
         * en cada ciclo del bucle el numero aleatorio generado del intervalo(0 -2)
         * suma un valor en el contador de ese numero que ha salido
         */

        for (int i = 0; i < 3; i++) {
            System.out.println("El numero " + i + " del intervalo(0-2) ha salido: \n" +
                ndigitos[i] + " veces de las 100 veces que se elegido al azar un numero comprendido entre (0 -2)");
            System.out.println("");
        }
        /**
         * muestra por cada posicion del array(0 1 2)
         * el numero de veces que ha salido ese numero en los 100 ciclos del
         * bucle anterior
         * importante que el numero de ciclos de este bucle sea el mismo que
         * el numero de valores del intervalo
         */
    }
}
}
```

# **LA CLASE LISTA (ANÁLISIS DE UN CASO PRÁCTICO)**

Crear una clase denominada *Lista* cuyo miembro dato sea un array de números enteros y cuyas funciones miembro realicen las siguientes tareas:

- Hallar y devolver el valor mayor
- Hallar y devolver el valor menor
- Hallar y devolver el valor medio
- Ordenar los números enteros de menor a mayor
- Mostrar la lista ordenada separando los elementos por un tabulador

### Definición de la clase *Lista*

Empezamos la definición de la clase escribiendo la palabra reservada **class** y a continuación el nombre de la clase *Lista*.

#### Los miembros dato

Los miembros dato de la clase *Lista* serán un array de enteros *x*, y opcionalmente la dimensión del array *n*.

```
public class Lista {  
    int[] x; //array de datos  
    int n; //dimensión
```

#### El constructor

Al constructor de la clase *Lista* se le pasará un array de enteros para inicializar los miembros dato

```
    public Lista(int[] x) {  
        this.x=x;  
        n=x.length;  
    }
```

#### Las funciones miembro

Las funciones miembro tienen acceso a los miembros dato, el array de enteros *x* y la dimensión del array *n*.

- El valor medio

Para hallar el valor medio, se suman todos los elementos del array y se divide el resultado por el número de elementos.

```
    double valorMedio(){  
        int suma=0;  
        for(int i=0; i<n; i++){  
            suma+=x[i];  
        }  
        return (double)suma/n;  
    }
```

La división de dos enteros *suma* y *n* (número de elementos del array) es un número entero. Por tanto, se ha de promocionar el entero *suma* de **int** a **double** para efectuar la división y devolver el resultado de esta operación.

- El valor mayor

```
int valorMayor(){
    int mayor=x[0];
    for(int i=1; i<n; i++){
        if(x[i]>mayor) mayor=x[i];
    }
    return mayor;
}
```

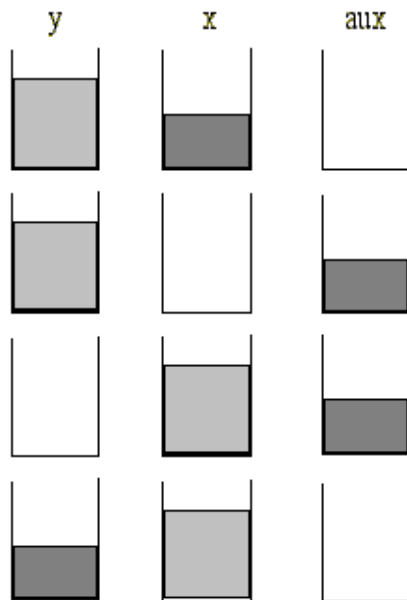
Se compara cada elemento del array con el valor de la variable local *mayor*, que inicialmente tiene el valor del primer elemento del array, si un elemento del array es mayor que dicha variable auxiliar se guarda en ella el valor de dicho elemento del array. Finalmente, se devuelve el valor *mayor* calculado

- El valor menor

```
int valorMenor(){
    int menor=x[0];
    for(int i=1; i<n; i++){
        if(x[i]<menor) menor=x[i];
    }
    return menor;
}
```

El código es similar a la función *valorMayor*

- Ordenar un conjunto de números



Para intercambiar el contenido de dos recipientes *x* e *y* sin que se mezclen, precisamos de un recipiente auxiliar *aux* vacío. Se vuelca el contenido del recipiente *x* en el recipiente *aux*, el recipiente *y* se vuelca en *x*, y por último, el recipiente *aux* se vuelca en *y*. Al final del proceso, el recipiente *aux* vuelve a estar vacío como al principio. En la figura se esquematiza este proceso.

```
aux=x;
x=y;
y=aux;
```



Para ordenar una lista de números se compara el primer elemento, índice 0, con todos los demás elementos de la lista, si el primer elemento es mayor que el elemento  $j$ , se intercambian sus valores, siguiendo el procedimiento explicado en la figura anterior. Se continúa este procedimiento con todos los elementos del array menos el último. La figura explica de forma gráfica este procedimiento.

```

void ordenar(){
    int aux;
    for(int i=0; i<n-1; i++){
        for(int j=i+1; j<n; j++){
            if(x[j]<x[i]){
                aux=x[j];
                x[j]=x[i];
                x[i]=aux;
            }
        }
    }
}
    
```

Caben ahora algunas mejoras en el programa, así la función *ordenar* la podemos utilizar para hallar el valor mayor, y el valor menor. Si tenemos una lista ordenada en orden ascendente, el último elemento de la lista será el valor mayor y el primero, el valor menor.

De este modo, podemos usar una función en otra funciones, lo que resulta en un ahorro de código, y en un aumento de la legibilidad del programa.

```

int valorMayor(){
    ordenar();
    return x[n-1];
}
int valorMenor(){
    ordenar();
    return x[0];
}
    
```

- Imprimir la lista ordenada

Imprimimos la lista ordenada separando sus elementos por un tabulador. Primero, se llama a la función *ordenar*, y después se imprime un elemento a continuación del otro mediante *System.out.print*. Recuerdese, que *System.out.println* imprime y a continuación pasa a la siguiente línea.

```

void imprimir(){
    ordenar();
    for(int i=0; i<n; i++){
        System.out.print("\t"+x[i]);
    }
    System.out.println("");
}
    
```

El código completo de la clase *Lista*, es el siguiente

```

public class Lista {
    int[] x; //array de datos
    int n; //dimensión

    public Lista(int[] x) {
        this.x=x;
        n=x.length;
    }

    double valorMedio(){
        int suma=0;
        for(int i=0; i<n; i++){
            suma+=x[i];
        }
        return (double)suma/n;
    }

    int valorMayor(){
        int mayor=x[0];
        for(int i=1; i<n; i++){
            if(x[i]>mayor) mayor=x[i];
        }
        return mayor;
    }

    int valorMenor(){
        int menor=x[0];
        for(int i=1; i<n; i++){
            if(x[i]<menor) menor=x[i];
        }
        return menor;
    }

    /**
     * funcion para ordenar de menor a mayor los valores del array
     * se pone n-1 para evitar que se salga de la dimension del array en la
     * comprobacion de los valores
     * j se refiere al valor que esta una posicion por delante que i en el array
     * aux es una variable que nos permite el intercambio de valores en la ordenacion
     */
    void ordenar(){
        int aux;
        for(int i=0; i<n-1; i++){
            for(int j=i+1; j<n; j++){
                if(x[j]<x[i]){
                    aux=x[j];
                    x[j]=x[i];
                    x[i]=aux;}}}}

    void imprimir(){
        ordenar();
        for(int i=0; i<n; i++){
            System.out.print("\t"+x[i]);
        }
        System.out.println("");
    }
}

```

## Los objetos de la clase *Lista*

A partir de un array de enteros podemos crear un objeto *lista* de la clase *Lista*.

```
int[] valores={10, -4, 23, 12, 16};  
Lista lista=new Lista(valores);
```

Estas dos sentencias las podemos convertir en una

```
Lista lista=new Lista(new int[] {10, -4, 23, 12, 16});
```

En el resto del código, el objeto *lista* llama a las funciones miembro

```
System.out.println("Valor mayor "+lista.valorMayor());  
System.out.println("Valor menor "+lista.valorMenor());  
System.out.println("Valor medio "+lista.valorMedio());  
lista.imprimir();
```

# **MODIFICADORES DE VARIABLES**

## Diferencia entre modificador static y final

Ya mencionamos la diferencia entre variables de instancia, de clase y locales . Consideremos la clase *Circulo*, con dos miembros dato, el *radio* específico para cada círculo y el número *PI* que tiene el mismo valor para todos los círculos. La primera es una variable de instancia y la segunda es una variable de clase.

Para indicar que el miembro *PI* es una variable de clase se le antepone el modificador **static**. El modificador **final** indica que es una constante que no se puede modificar, una vez que la variable *PI* ha sido inicializada.

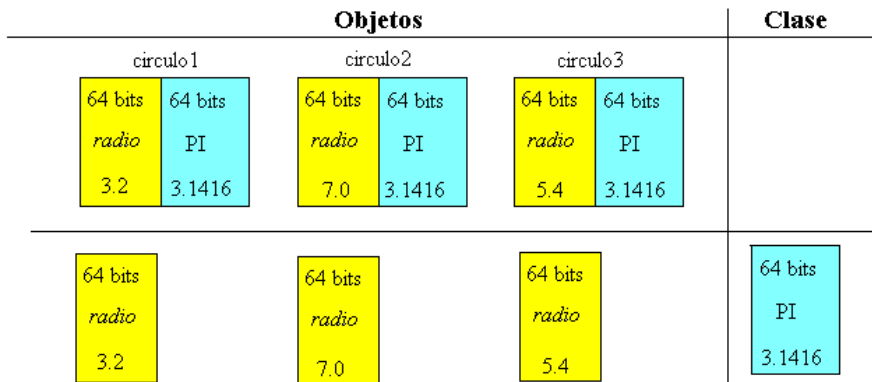
Definimos también una función miembro denominada *calcularArea* que devuelva el área del círculo

```
public class Circulo{
    static final double PI=3.1416;
    double radio;
    public Circulo(double radio){
        this.radio=radio;
    }
    double calcularArea(){
        return (PI*radio*radio);
    }
}
```

Para calcular el área de un círculo, creamos un objeto *circulo* de la clase *Circulo* dando un valor al radio. Desde este objeto llamamos a la función miembro *calcularArea*.

```
Circulo circulo=new Circulo(2.3);
System.out.println("área: "+circulo.calcularArea());
```

Veamos ahora las ventajas que supone declarar la constante *PI* como miembro estático.



Si *PI* y *radio* fuesen variables de instancia

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

Creamos tres objetos de la clase *Circulo*, de radios 3.2, 7.0, y 5.4

```
Circulo circulo1=new Circulo(3.2);
Circulo circulo2=new Circulo(7.0);
Circulo circulo3=new Circulo(5.4);
```

Al crearse cada objeto se reservaría espacio para el dato *radio* (64 bits), y para el dato PI (otros 64 bits). Como vemos en la parte superior de la figura, se desperdicia la memoria del ordenador, guardando tres veces el mismo dato PI.

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

Declarando PI estático (**static**), la variable PI queda ligada a la clase *Circulo*, y se reserva espacio en memoria una sólo vez, tal como se indica en la parte inferior de la figura. Si además la variable PI no cambia, e s una constante, le podemos anteponer la palabra **final**.

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

## Relación con variables de instancia y variables de clase

Las variables de clase o miembros estáticos son aquellos a los que se antepone el modificador **static**. Vamos a comprobar que un miembro dato estático guarda el mismo valor en todos los objetos de dicha clase.

Sea una clase denominada *Alumno* con una variable de instancia o miembro dato, la nota de selectividad, y un miembro estático constante denominado nota de corte.

```
double nota;
static double notaCorte=6.0;
```

La nota es un atributo que tiene un valor distinto para cada uno de los alumnos u objetos de la clase *Alumno*, mientras que la nota de corte es un atributo que tiene el mismo valor para a un conjunto de alumnos.

Se define también en dicha clase una función miembro que determine si está (**true**) o no (**false**) admitido.

```
public class Alumno {
    double nota;
    static double notaCorte=6.0;
    public Alumno(double nota) {
        this.nota=nota;
    }
    boolean estaAdmitido(){
        return (nota>=notaCorte);
    }
}
```

Creamos ahora un array de cuatro alumnos y asignamos a cada uno de ellos una nota.

```
Alumno[] alumnos={new Alumno(5.5), new Alumno(6.3), new Alumno(7.2), new Alumno(5.0)};
```

Contamos el número de alumnos que están admitidos

```
int numAdmitidos=0;
for(int i=0; i<alumnos.length; i++){
    if (alumnos[i].estaAdmitido()){
        numAdmitidos++;
    }
}
System.out.println("admitidos "+numAdmitidos);
```

Accedemos al miembro dato *notaCorte* desde un objeto de la clase *Alumno*, para cambiarla a 7.0

```
alumnos[1].notaCorte=7.0;
```

Comprobamos que todos los objetos de la clase *Alumno* tienen dicho miembro dato estático *notaCorte* cambiado a 7.0

```
for(int i=0; i<alumnos.length; i++){
    System.out.println("nota de corte "+alumnos[i].notaCorte);
}
```

El miembro dato *notaCorte* tiene el modificador **static** y por tanto está ligado a la clase más que a cada uno de los objetos de dicha clase. Se puede acceder a dicho miembro con la siguiente sintaxis

```
Nombre_de_la_clase.miembro_estático;
```

Si ponemos

```
Alumno.notaCorte=6.5;
for(int i=0; i<alumnos.length; i++){
    System.out.println("nota de corte "+alumnos[i].notaCorte);
}
```

Veremos que todos los objetos de la clase *Alumno* habrán cambiado el valor del miembro dato estático *notaCorte* a 6.5.

Un miembro dato estático de una clase se puede acceder desde un objeto de la clase, o mejor, desde la clase misma.

# LA CLASE MATH



La clase *Math* tiene miembros dato y funciones miembro estáticas, vamos a conocer algunas de estas funciones, cómo se llaman y qué tarea realizan.

### Miembros dato constantes

La clase *Math* define dos constantes muy útiles, el número pi y el número e.

```
public final class Math {  
    public static final double E = 2.7182818284590452354;  
    public static final double PI = 3.14159265358979323846;  
    //...  
}
```

El modificador **final** indica que los valores que guardan no se pueden cambiar, son valores constantes

Se accede a estas constantes desde la clase *Math*, de la siguiente forma

```
System.out.println("Pi es " + Math.PI);  
System.out.println("e es " + Math.E);
```

### Funciones miembro

La clase *Math* define muchas funciones y versiones distintas de cada función.

Por ejemplo, para hallar el valor absoluto de un número define las siguientes funciones. Se llama a una u otra dependiendo del tipo de dato que se le pasa en su único argumento.

Por ejemplo, hallar el valor absoluto de los siguientes números

```
int i = -9;  
double x = 0.3498;  
System.out.println("el valor absoluto de " + i + " es: " + Math.abs(i));  
System.out.println("el valor absoluto de " + x + " es: " + Math.abs(x));
```

### Función potencia y raíz cuadrada

Para elevar un número *x* a la potencia *y*, se emplea *pow(x, y)*

```
System.out.println("pow(10.0, 3.5) es " + Math.pow(10.0,3.5));
```

Para hallar la raíz cuadrada de un número, se emplea a la función *sqrt*

```
System.out.println("La raíz cuadrada de " + x + " is " + Math.sqrt(x));
```

### Aproximación de un número decimal

Para expresar un número real con un número especificado de números decimales empleamos la función *round*. Por ejemplo, para expresar los números *x* e *y* con dos cifras decimales escribimos

```
double x = 72.3543;
double y = 0.3498;
System.out.println(x + " es aprox. " + (double)Math.round(x*100)/100);
System.out.println(y + " es aprox. " + (double)Math.round(y*100)/100);
```

Se obtiene 72.35 y 0.35 como cabría esperar. *round* devuelve un número redondeado.

Si empleamos la función *floor* en vez de *round* obtendríamos

```
System.out.println(x + " es aprox. " + Math.floor(x*100)/100);
System.out.println(y + " es aprox. " + Math.floor(y*100)/100);
```

Se obtiene 72.35 y 0.34 pero sin redondear, además no hace falta promocionar a *double*.

### El mayor y el menor de dos números

Para hallar el mayor y el menor de dos números se emplean las funciones *min* y *max* que comparan números del mismo tipo.

```
int i = 7;
int j = -9;
double x = 72.3543;
double y = 0.3498;
// para hallar el menor de dos número
System.out.println("min(" + i + ", " + j + ") es " + Math.min(i,j));
System.out.println("min(" + x + ", " + y + ") es " + Math.min(x,y));
// Para hallar el mayor de dos números
System.out.println("max(" + i + ", " + j + ") es " + Math.max(i,j));
System.out.println("max(" + x + ", " + y + ") es " + Math.max(x,y));
```

### Números aleatorios

La clase *Math* define una función denominada *random* que devuelve un número pseudo aleatorio comprendido en el intervalo [0.0, 1.0). Existe otra alternativa, se pueden generar números pseudo aleatorios a partir de un objeto de la clase *Random*, que llame a la función miembro *nextDouble*.

```
System.out.println("Número aleatorio: " + Math.random());
System.out.println("Otro número aleatorio: " + Math.random());
```

```

package matematicas;

public class MatematicasApp {
    public static void main(String args[]) {

        int i = 7;
        int j = -9;
        double x = 72.3543;
        double y = 0.3498;

        System.out.println("i es " + i);
        System.out.println("j es " + j);
        System.out.println("x es " + x);
        System.out.println("y es " + y);

        // Valor absoluto de un número el valor absoluto siempre muestra el positivo de un negativo
        System.out.println("|" + i + "|" es " + Math.abs(i));
        System.out.println("|" + j + "|" es " + Math.abs(j));
        System.out.println("|" + x + "|" es " + Math.abs(x));
        System.out.println("|" + y + "|" es " + Math.abs(y));

        // aproximación decimal empleando (round)
        System.out.println(x + " es " + Math.round(x)); //eliminamos toda la parte decimal y se redondea
        System.out.println(y + " es " + Math.round(y));
        //expresar los números x e y con dos cifras decimales escribimos
        System.out.println(x + " es aprox." + (double)Math.round(x*100)/100);
        System.out.println(y + " es aprox." + (double)Math.round(y*100)/100);

        //empleando floor.. con floor no hay redondeo
        System.out.println("The floor of " + x + " es " + Math.floor(x*100)/100);
        System.out.println("The floor of " + y + " es " + Math.floor(y*100)/100);

        // para hallar el menor y mayor de dos número
        System.out.println("min(" + i + ", " + j + ") es " + Math.min(i,j));
        System.out.println("min(" + x + ", " + y + ") es " + Math.min(x,y));
        System.out.println("max(" + i + ", " + j + ") es " + Math.max(i,j));
        System.out.println("max(" + x + ", " + y + ") es " + Math.max(x,y));

        // las constantes PI y E
        System.out.println("Pi es " + Math.PI);
        System.out.println("e es " + Math.E);

        // pow(x,y) devuelve x elevado a y.
        System.out.println("pow(2.0, 2.0) es " + Math.pow(2.0,2.0));
        System.out.println("pow(10.0, 3.5) es " + Math.pow(10.0,3.5));
        System.out.println("pow(8, -1) es " + Math.pow(8,-1));

        // sqrt(x) devuelve la raíz cuadrada de x.
        System.out.println("La raíz cuadrada de " + y + " es " + Math.sqrt(y));

        // Devuelve un número pseudo-aleatorio comprendido entre 0.0 y 1.0
        System.out.println("Número aleatorio: " + Math.random());
        System.out.println("Otro número aleatorio: " + Math.random());

    }
}

```

# **MODIFICADORES DE ACCESO**

## Public y Private

El acceso a los miembros de una clase está controlado. Para usar una clase, solamente necesitamos saber que funciones miembro se pueden llamar y a qué datos podemos acceder, no necesitamos saber como está hecha la clase, como son sus detalles internos.

Para acceder a un miembro público (dato o función) basta escribir.

```
objeto_de_la_clase.miembro_público_no_estático  
clase.miembro_público_estático
```

Delante de los miembros dato, como podemos ver en el listado hemos puesto las palabras reservadas **public** y **private**.

- Miembros públicos

Los miembros públicos son aquellos que tienen delante la palabra **public**, y se puede acceder a ellos sin ninguna restricción.

- Miembros privados

Los miembros privados son aquellos que tienen delante la palabra **private**, y se puede acceder a ellos solamente dentro del ámbito de la clase.

- Por defecto (a nivel de paquete)

Cuando no se pone ningún modificador de acceso delante de los miembros (funciones o datos), se dice que son accesibles dentro del mismo paquete ([package](#)).

**package** es la primera sentencia que se pone en un archivo .java. Cada archivo .java contiene habitualmente una clase. Si tiene más de una solamente una de ellas es pública. El nombre de dicha clase coincide con el nombre del archivo.

```
public class Lista {  
    private int[] x; //solo se accede a ellos dentro de la propia clase  
    private int n;  
  
    public Lista(int[] x) {  
        this.x=x;  
        n=x.length;  
    }  
}
```

Si desde otra clase se intenta acceder a ellos mediante objeto.miembro\_dato dará un error de compilación.

# **FUNCIÓN MIEMBRO TOSTRING**

La función miembro pública *toString* da una representación en forma de texto en un contexto gráfico.

Si los miembros dato de la clase *Lista* son privados (**private**) hemos de definir una función que hemos denominado *imprimir* para mostrar los valores que guardan los miembros dato de los objetos de la clase *Lista*.

```
public class Lista {
    private int[] x;    //array de datos
    private int n;    //dimensión
    //...
    public void imprimir(){
        for(int i=0; i<n; i++){
            System.out.print("\t"+x[i]);
        }
        System.out.println("");
    }
}
```

La llamada a esta función miembro se efectúa desde un objeto de la clase *Lista*

```
Lista lista=new Lista(new int[]{60, -4, 23, 12, -16});
System.out.println("Mostrar la lista");
lista.imprimir();
```

Sustituimos la función miembro *imprimir* por la redefinición de *toString*.

```
public class Lista {
    private int[] x;    //array de datos
    private int n;    //dimensión
    //...
    public String toString(){
        String texto="";
        for(int i=0; i<n; i++){
            texto+="\t"+x[i];
        }
        return texto;
    }
}
```

La llamada a la función *toString* se realiza implícitamente en el argumento de la función *System.out.println*, o bien, al concatenar un string y un objeto de la clase *Lista*.

```
Lista lista=new Lista(new int[]{60, -4, 23, 12, -16});
System.out.println("Mostrar la lista");
System.out.println(lista);
```

```
public class Lista {
    private int[] x; //array de datos
    private int n; //dimensión

    public Lista(int[] x) {
        this.x=x;
        n=x.length;
        ordenar();
    }

    public double valorMedio(){
        int suma=0;
        for(int i=0; i<n; i++){
            suma+=x[i];
        }
        return (double)suma/n;
    }

    public int valorMayor(){
        return x[n-1];
    }

    public int valorMenor(){
        return x[0];
    }

    private void ordenar(){
        int aux;
        for(int i=0; i<n-1; i++){
            for(int j=i+1; j<n; j++){
                if(x[j]<x[i]){
                    aux=x[j];
                    x[j]=x[i];
                    x[i]=aux;
                }
            }
        }
    }

    public String toString(){
        String texto="";
        for(int i=0; i<n; i++){
            texto+="\t"+x[i];
        }
        return texto;
    }
}
```



# HERENCIA

La herencia es una propiedad esencial de la Programación Orientada a Objetos que consiste en la creación de nuevas clases a partir de otras ya existentes.

Java permite heredar a las clases características y conductas de una o varias clases denominadas base. Las clases que heredan de clases base se denominan derivadas, estas a su vez pueden ser clases bases para otras clases derivadas. Se establece así una clasificación jerárquica.

## La clase base

Supongamos que tenemos una clase que describe la conducta de una ventana muy simple, aquella que no dispone de título en la parte superior, por tanto no puede desplazarse, pero si cambiar de tamaño actuando con el ratón en los bordes derecho e inferior.

La clase *Ventana* tendrá los siguientes miembros dato: la posición *x* e *y* de la ventana, de su esquina superior izquierda y las dimensiones de la ventana: *ancho* y *alto*.

```
public class Ventana {
    protected int x;
    protected int y;
    protected int ancho;
    protected int alto;

    public Ventana(int x, int y, int ancho, int alto) {
        this.x=x;
        this.y=y;
        this.ancho=ancho;
        this.alto=alto;
    }
    //...
}
```

Las funciones miembros, además del constructor serán las siguientes: la función *mostrar* que simula una ventana en un entorno gráfico, aquí solamente nos muestra la posición y las dimensiones de la ventana.

```
public void mostrar(){
    System.out.println("posición : x="+x+", y="+y);
    System.out.println("dimensiones : w="+ancho+", h="+alto);
}
```

La función *cambiarDimensiones* que simula el cambio en la anchura y altura de la ventana.

```
public void cambiarDimensiones(int dw, int dh){
    ancho+=dw;
    alto+=dh;
}
```

El código completo de la clase base *Ventana*, es el siguiente

```
package ventana;

public class Ventana {
    protected int x;
    protected int y;
    protected int ancho;
    protected int alto;

    public Ventana(int x, int y, int ancho, int alto) {
        this.x=x;
        this.y=y;
        this.ancho=ancho;
        this.alto=alto;
    }

    public void mostrar(){
        System.out.println("posición : x="+x+", y="+y);
        System.out.println("dimensiones : w="+ancho+", h="+alto);
    }

    public void cambiarDimensiones(int dw, int dh){
        ancho+=dw;
        alto+=dh;
    }
}
```

Objetos de la clase base

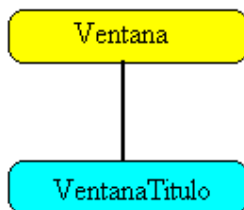
Como vemos en el código, el constructor de la clase base inicializa los cuatro miembros de datos. Llamamos al constructor creando un objeto de la clase *Ventana*

```
Ventana ventana=new Ventana(0, 0, 20, 30);
```

Desde el objeto *ventana* podemos llamar a las funciones miembro públicas

```
ventana.mostrar();
ventana.cambiarDimensiones(10, 10);
ventana.mostrar();
```

### La clase derivada



Incrementamos la funcionalidad de la clase *Ventana* definiendo una clase derivada denominada *VentanaTitulo*. Los objetos de dicha clase tendrán todas las características de los objetos de la clase base, pero además tendrán un título, y se podrán desplazar.

La clase derivada heredará los miembros dato de la clase base y las funciones miembro, y tendrá un miembro dato más, el título de la ventana.

```
public class VentanaTitulo extends Ventana{
    protected String titulo;

    public VentanaTitulo(int x, int y, int w, int h, String nombre) {
        super(x, y, w, h); // llamada al constructor de la clase base
        titulo=nombre;
    }
}
```

**extends** es la palabra reservada que indica que la clase *VentanaTitulo* deriva, o es una subclase, de la clase *Ventana*.

La primera sentencia del constructor de la clase derivada es una llamada al constructor de la clase base mediante la palabra reservada **super**. La llamada

```
super(x, y, w, h);
```

inicializa los cuatro miembros dato de la clase base *Ventana*: *x*, *y*, *ancho*, *alto*.

La función miembro denominada *desplazar* cambia la posición de la ventana, añadiéndole el desplazamiento.

```
public void desplazar(int dx, int dy){
    x+=dx;
    y+=dy;
}
```

Redefinición de la función miembro *mostrar* para mostrar una ventana con un título.

```
public void mostrar(){
    super.mostrar(); // llamada a la función mostrar de la clase base.
    System.out.println("titulo : "+titulo);
}
```

En la clase derivada se define una función que tiene el mismo nombre y los mismos parámetros que la de la clase base. Se dice que redefinimos la función *mostrar* en la clase derivada. La función miembro *mostrar* de la clase derivada *VentanaTitulo* hace una llamada a la función *mostrar* de la clase base *Ventana*, mediante

```
super.mostrar();
```

De este modo aprovechamos el código ya escrito, y le añadimos el código que describe la nueva funcionalidad de la ventana por ejemplo, que muestre el título.

Si nos olvidamos de poner la palabra reservada **super** llamando a la función *mostrar*, tendríamos una función recursiva. La función *mostrar* llamaría a *mostrar* indefinidamente.

```
public void mostrar(){ //¡¡ojo!, función recursiva
    System.out.println("titulo : "+titulo);
    mostrar();
}
```

La definición de la clase derivada *VentanaTitulo*, será la siguiente.

```
package ventana;

public class VentanaTitulo extends Ventana{
    protected String titulo;

    public VentanaTitulo(int x, int y, int w, int h, String nombre) {
        super(x, y, w, h);
        titulo=nombre;
    }

    public void mostrar(){
        super.mostrar();
        System.out.println("titulo   : "+titulo);
    }

    public void desplazar(int dx, int dy){
        x+=dx;
        y+=dy;
    }
}
```

### Objetos de la clase derivada

Creamos un objeto *ventana* de la clase derivada *VentanaTitulo*

```
VentanaTitulo ventana=new VentanaTitulo(0, 0, 20, 30, "Principal");
```

Mostramos la ventana con su título, llamando a la función *mostrar*, redefinida en la clase derivada

```
ventana.mostrar();
```

Desde el objeto *ventana* de la clase derivada llamamos a las funciones miembro definidas en dicha clase

```
ventana.desplazar(4, 3);
```

Desde el objeto *ventana* de la clase derivada podemos llamar a las funciones miembro definidas en la clase base.

```
ventana.cambiarDimensiones(10, -5);
```

Para mostrar la nueva ventana desplazada y cambiada de tamaño escribimos

```
ventana.mostrar();
```

### **Controles de acceso (public, private y protected)**

Ya hemos visto el significado de los modificadores de acceso **public** y **private**, así como el control de acceso por defecto a nivel de paquete, cuando no se especifica nada. En la herencia, surge un nuevo control de acceso denominado **protected**.

Hemos puesto **protected** delante de los miembros dato *x* e *y* de la clase base *Ventana*

```
public class Ventana {
    protected int x;
    protected int y;
    //..}
```

En la clase derivada la función miembro *desplazar* accede a dichos miembros dato

```
public class VentanaTitulo extends Ventana{
//...
    public void desplazar(int dx, int dy){
        x+=dx;
        y+=dy;
    }
}
```

Si cambiamos el modificador de acceso de los miembros *x* e *y* de la clase base *Ventana* de **protected** a **private**, veremos que el compilador se queja diciendo que los miembro *x* e *y* no son accesibles.

Los miembros *ancho* y *alto* se pueden poner con acceso **private** sin embargo, es mejor dejarlos como **protected** ya que podrían ser utilizados por alguna función miembro de otra clase derivada de *VentanaTitulo*.

Dentro de una jerarquía pondremos un miembro con acceso **private**, si estamos seguros de que dicho miembro solamente va a ser usado por dicha clase.

Como vemos hay cuatro modificadores de acceso a los miembros dato y a los métodos: **private**, **protected**, **public** y **default** (por defecto, o en ausencia de cualquier modificador).

Las clases dentro del mismo paquete tienen diferentes accesos que las clases de distinto paquete

Los siguientes cuadros tratan de aclarar el acceso.

Clases dentro del mismo paquete		
Modificador de acceso	Heredado	Accesible
Por defecto (sin modificador)	Si	Si
private	No	No
protected	Si	Si
public	Si	Si

Clases en distintos paquetes		
Modificador de acceso	Heredado	Accesible
Por defecto (sin modificador)	No	No
private	No	No
protected	Si	No
public	Si	Si

## La clase base *Object*

La clase *Object* es la clase raíz de la cual derivan todas las clases. Esta derivación es implícita.

La clase *Object* define una serie de funciones miembro que heredan todas las clases. Las más importantes son las siguientes

### Igualdad de dos objetos:

Hemos visto que el método *equals* de la clase *String* cuando compara un string y cualquier otro objeto. El método *equals* de la clase *Object* compara dos objetos uno que llama a la función y otro es el argumento de dicha función.

### Representación en forma de texto de un objeto

El método *toString* imprime por defecto el nombre de la clase a la que pertenece el objeto. La función *toString* se llama automáticamente siempre que pongamos un objeto como argumento de la función *System.out.println* o concatenado con otro string.

### Duplicación de objetos

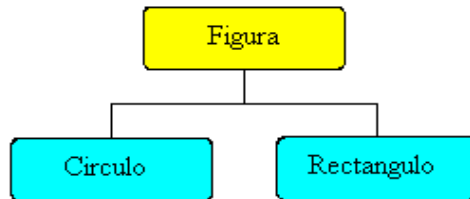
El método *clone* crea un objeto duplicado (clónico) de otro objeto.

# JERARQUÍA DE CLASES



## Clases y métodos abstractos

Consideremos las figuras planas cerradas como el rectángulo, y el círculo. Tales figuras comparten características comunes como es la posición de la figura, de su centro, y el área de la figura, aunque el procedimiento para calcular dicha área sea completamente distinto. Podemos por tanto, diseñar una jerarquía de clases, tal que la clase base denominada *Figura*, tenga las características comunes y cada clase derivada las específicas. La relación jerárquica se muestra en la figura



La clase *Figura* es la que contiene las características comunes a dichas figuras concretas por tanto, no tiene forma ni tiene área. Esto lo expresamos declarando *Figura* como una clase abstracta, declarando la función miembro *area* **abstract**.

Las clases abstractas solamente se pueden usar como clases base para otras clases. No se pueden crear objetos pertenecientes a una clase abstracta. Sin embargo, se pueden declarar variables de dichas clases.

- La clase *Figura*

La definición de la clase abstracta *Figura*, contiene la posición  $x$  e  $y$  de la figura particular, de su centro, y la función *area*, que se va a definir en las clases derivadas para calcular el área de cada figura en particular.

```

public abstract class Figura {
    protected int x;
    protected int y;

    public Figura(int x, int y) {
        this.x=x;
        this.y=y;
    }

    public abstract double area();
}
  
```

- La clase *Rectangulo*

Las clases derivadas heredan los miembros de datos  $x$  e  $y$  de la clase base, y definen la función *area*, declarada **abstract** en la clase base *Figura*, ya que cada figura particular tiene una fórmula distinta para calcular su área. Por ejemplo, la clase derivada *Rectangulo*, tiene como datos, aparte de su posición  $(x, y)$  en el plano, sus dimensiones, es decir, su anchura *ancho* y altura *alto*.

```

class Rectangulo extends Figura{
    protected double ancho, alto;

    public Rectangulo(int x, int y, double ancho, double alto){
        super(x,y);
        this.ancho=ancho;
        this.alto=alto;
    }

    public double area(){
        return ancho*alto;
    }
}

```

La primera sentencia en el constructor de la clase derivada es una llamada al constructor de la clase base, para ello se emplea la palabra reservada **super**. El constructor de la clase derivada llama al constructor de la clase base y le pasa las coordenadas del punto  $x$  e  $y$ . Después inicializa sus miembros de datos *ancho* y *alto*.

En la definición de la función *area*, se calcula el área del rectángulo como producto de la anchura por la altura, y se devuelve el resultado

- La clase *Circulo*

```

class Circulo extends Figura{
    protected double radio;

    public Circulo(int x, int y, double radio){
        super(x,y);
        this.radio=radio;
    }

    public double area(){
        return Math.PI*radio*radio;
    }
}

```

Como vemos, la primera sentencia en el constructor de la clase derivada es una llamada al constructor de la clase base empleando la palabra reservada **super**. Posteriormente, se inicializa el miembro de datos *radio*, de la clase derivada *Circulo*.

En la definición de la función *area*, se calcula el área del círculo mediante la conocida fórmula  $\pi \cdot r^2$ , o bien  $\pi \cdot r \cdot r$ . En este caso usamos la constante *Math.PI*.

### Uso de la jerarquía de clases

Creamos un objeto *c* de la clase *Circulo* situado en el punto (0, 0) y de 5.5 unidades de radio. Calculamos y mostramos el valor de su área.

```

Circulo c=new Circulo(0, 0, 5.5);
System.out.println("Area del círculo "+c.area());

```

Creamos un objeto *r* de la clase *Rectangulo* situado en el punto (0, 0) y de dimensiones 5.5 de anchura y 2 unidades de largo. Calculamos y mostramos el valor de su área.

```

Rectangulo r=new Rectangulo(0, 0, 5.5, 2.0);
System.out.println("Area del rectángulo "+r.area());

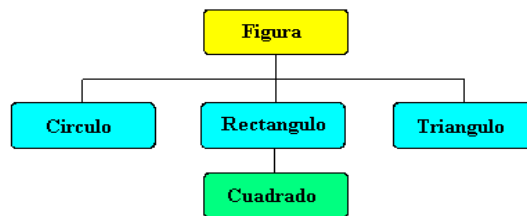
```

Veamos ahora, una forma alternativa, guardamos el valor devuelto por **new** al crear objetos de las clases derivadas en una variable *f* del tipo *Figura* (clase base).

```
Figura f=new Circulo(0, 0, 5.5);
System.out.println("Area del círculo "+f.area());
f=new Rectangulo(0, 0, 5.5, 2.0);
System.out.println("Area del rectángulo "+f.area());
```

## El operador instanceof

Ampliamos el árbol jerárquico de las clases.



- La clase *Cuadrado*

La clase *Cuadrado* es una clase especializada de *Rectangulo*, ya que un cuadrado tiene los lados iguales. El constructor solamente precisa de tres argumentos los que corresponden a la posición de la figura (x,y) y la longitud del lado

```
class Cuadrado extends Rectangulo{
    public Cuadrado(int x, int y, double dimension){
        super(x, y, dimension, dimension);
    }
}
```

El constructor de la clase derivada llama al constructor de la clase base y le pasa la posición *x* e *y* de la figura, el ancho y alto que tienen el mismo valor. No es necesario redefinir una nueva función *area*. La clase *Cuadrado* hereda la función *area* definida en la clase *Rectangulo*.

- La clase *Triangulo*

La clase derivada *Triángulo*, tiene como datos, aparte de su posición (*x, y*) en el plano, la base y la altura del triángulo.

```
class Triangulo extends Figura{
    protected double base, altura;

    public Triangulo(int x, int y, double base, double altura){
        super(x, y);
        this.base=base;
        this.altura=altura;
    }

    public double area(){
        return base*altura/2;
    }
}
```

El constructor de la clase *Triangulo* llama al constructor de la clase *Figura*, le pasa las coordenadas *x* e *y* de su centro, y luego inicializa los miembros dato *base* y *altura*.

En la definición de la función *area*, se calcula el área del triángulo como producto de la *base* por la *altura* y dividido por dos.

Creamos un array del tipo *Figura*, guardando en sus elementos las direcciones devueltas por **new** al crear cada uno de los objetos.

```
Figura[] fig=new Figura[4];
fig[0]=new Rectangulo(0,0, 5.0, 2.0);
fig[1]=new Circulo(0,0, 3.0);
fig[2]=new Cuadrado(0, 0, 5.0);
fig[3]=new Triangulo(0,0, 7.0, 12.0);
```

### El operador *instanceof*

El operador **instanceof** tiene dos operandos: un objeto en el lado izquierdo y una clase en el lado derecho. Esta expresión devuelve **true** o **false** dependiendo de que el objeto situado a la izquierda sea o no una instancia de la clase situada a la derecha o de alguna de sus clases derivadas.

Por ejemplo.

```
Rectangulo rect=new Rectangulo(0, 0, 5.0, 2.0);
rect instanceof String //false
rect instanceof Rectangulo //true
```

El objeto *rect* de la clase *Rectangulo* no es un objeto de la clase *String*. El objeto *rect* si es un objeto de la clase *Rectangulo*.

Veamos la relación entre *rect* y las clases de la jerarquía

```
rect instanceof Figura //true
rect instanceof Cuadrado //false
```

*rect* es un objeto de la clase base *Figura* pero no es un objeto de la clase derivada *Cuadrado*.

# **LA PALABRA CLAVE *FINAL***

## Clases y métodos finales

Se puede declarar una clase como **final**, cuando no nos interesa crear clases derivadas de dicha clase. La clase *Cuadrado* se puede declarar como **final**, ya que no se espera que ningún programador necesite crear clases derivadas de *Cuadrado*.

```
final class Cuadrado extends Rectangulo{
    public Cuadrado(int x, int y, double dimension){
        super(x, y, dimension, dimension);
    }
}
```

## Métodos finales

Como se ha comentado al introducir la herencia, una de las formas de aprovechar el código existente, es la de crear una clase derivada y redefinir algunos de los métodos de la clase base.

Para evitar que las clase derivadas redefinan una función miembro de una clase base, se le antepone la palabra clave **final**.

# INTERFACES

## Definición

Un interface es una colección de declaraciones de métodos (sin definirlos) y también puede incluir constantes.

Las clases que implementen (**implements**) el interface han de definir obligatoriamente la función del interface.

El papel del interface es el de describir algunas de las características de una clase.

## Diferencia entre un interface y una clase abstracta

Un interface es simplemente una lista de métodos no implementados, además puede incluir la declaración de constantes. Una clase abstracta puede incluir métodos implementados y no implementados o abstractos, miembros de datos, constantes y otros no constantes.

Una clase solamente puede derivar **extends** de una clase base, pero puede implementar varios interfaces. Los nombres de los interfaces se colocan separados por una coma después de la palabra reservada **implements**.

Vamos a crear un interface denominado *Parlanchin* que contenga la declaración de una función denominada *habla*.

```
public interface Parlanchin {  
    public abstract void habla();  
}
```

Hacemos que la jerarquía de clases que deriva de la clase base *Animal* implemente el interface *Parlanchin*

```
public abstract class Animal implements Parlanchin{  
    public abstract void habla();  
}  
  
class Perro extends Animal{  
    public void habla(){  
        System.out.println("¡Guau!");  
    }  
}  
  
class Gato extends Animal{  
    public void habla(){  
        System.out.println("¡Miau!");  
    }  
}
```



Ahora veamos otra jerarquía de clases completamente distinta, la que deriva de la clase base *Reloj*. Una de las clases de dicha jerarquía *Cucu* implementa el interface *Parlanchin* y por tanto, debe de definir obligatoriamente la función *habla* declarada en dicho interface.

```
public abstract class Reloj {
}

class Cucu extends Reloj implements Parlanchin{
    public void habla(){
        System.out.println("¡Cucu, cu cu, ..!");
    }
}
```

Definamos la función *hazleHablar* de modo que conozca al objeto que se le pasa no por una clase base, sino por el interface *Parlanchin*. A dicha función le podemos pasar cualquier objeto que implemente el interface *Parlanchin*, este o no en la misma jerarquía de clases.

```
public class PoliApp {

    public static void main(String[] args) {
        Gato gato=new Gato();
        hazleHablar(gato);
        Cucu cucu=new Cucu();
        hazleHablar(cucu);
    }

    static void hazleHablar(Parlanchin sujeto){
        sujeto.habla();
    }
}
```

Al ejecutar el programa, veremos que se imprime en la consola ¡Miau!, por que a la función *hazleHablar* se le pasa un objeto de la clase *Gato*, y después ¡Cucu, cucu, ..! por que a la función *hazleHablar* se le pasa un objeto de la clase *Cucu*.

Si solamente hubiese herencia simple, *Cucu* tendría que derivar de la clase *Animal* (lo que no es lógico) o bien no se podría pasar a la función *hazleHablar*. Con interfaces, cualquier clase en cualquier familia puede implementar el interface *Parlanchin*, y se podrá pasar un objeto de dicha clase a la función *hazleHablar*. Esta es la razón por la cual los interfaces proporcionan más posibilidades que el que se puede obtener de una simple jerarquía de clases.

# EXCEPCIONES

## Definición

En Java las situaciones que pueden provocar un fallo en el programa se denominan excepciones.

Java lanza una excepción en respuesta a una situación poco usual. El programador también puede lanzar sus propias excepciones. Las excepciones en Java son objetos de clases derivadas de la clase base *Exception*. La clase *Exception* es derivada de la clase base *Throwable*.

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

## Captura de excepciones

Empecemos por solucionar el error que se produce en el programa durante la compilación.

```
String str=" 12 ";
int numero;
try{
    numero=Integer.parseInt(str);
}catch(NumberFormatException ex){
    System.out.println("No es un número");
}
```

En el caso de que el string *str* contenga caracteres no numéricos como es éste el caso, el número 12 está acompañado de espacios en blanco sin eliminar con `trim`, se produce una excepción del tipo *NumberFormatException* que es capturada y se imprime el mensaje "No es un número".

En vez de un mensaje propio se puede imprimir el objeto *ex* de la clase *NumberFormatException* que informa del error.

```
try{
    //...
}catch(NumberFormatException ex){
    System.out.println(ex);
}
```

## Manejando varias excepciones

Vamos a crear un programa que divida dos números. Supongamos que los números se introducen en dos controles de edición. Se obtiene el texto de cada uno de los controles de edición que se guardan en dos strings. En esta situación se pueden producir dos excepciones *NumberFormatException*, si se introducen caracteres no numéricos y *ArithmeticException* si se divide entre cero.

Como vemos las sentencias susceptibles de lanzar una excepción se sitúan en un bloque **try...catch**. Si el denominador es cero, se produce una excepción de la clase *ArithmeticException* en la expresión que halla el cociente, que es inmediatamente capturada en el bloque **catch** que maneja dicha excepción, ejecutándose las sentencias que hay en dicho bloque. En este caso se guarda en el string *respuesta* el texto "División entre cero".

```

public class ExcepcionApp {
    public static void main(String[] args) {
        String str1="12";
        String str2="0";
        String respuesta;
        int numerador, denominador, cociente;
        try{
            numerador=Integer.parseInt(str1);
            denominador=Integer.parseInt(str2);
            cociente=numerador/denominador;
            respuesta=String.valueOf(cociente);
        }catch(NumberFormatException ex){
            respuesta="Se han introducido caracteres no numéricos";
        }catch(ArithmeticException ex){
            respuesta="División entre cero";
        }
        System.out.println(respuesta);
    }
}

```

## Lanzar excepciones

Una función miembro que lanza una excepción tiene la declaración habitual que cualquier otro método pero se le añade a continuación la palabra reservada **throws** seguido de la excepción o excepciones que puede lanzar.

```

static void rango(int num, int den) throws ExcepcionIntervalo{
    if((num>100)||den<-5){
        throw new ExcepcionIntervalo("Números fuera del intervalo");
    }
}

```

Cuando el numerador es mayor que 100 y el denominador es menor que 5 se lanza **throw** una excepción, un objeto de la clase *ExcepcionIntervalo*. Dicho objeto se crea llamando al constructor de dicha clase y pasándole un string que contiene el mensaje "Números fuera del intervalo".

Para extraer un mensaje de excepción se hace mediante la función miembro *getMessage*, del siguiente modo:

```

String respuesta;
try{
    ....
} catch(ExcepcionIntervalo ex) {
    respuesta=ex.getMessage();
}

System.out.println(respuesta);

```

```

public class ExcepcionApp3 {
    public static void main(String[] args) {
        String str1="120";
        String str2="3";
        String respuesta;
        int numerador, denominador, cociente;

        try{
            numerador=Integer.parseInt(str1);
            denominador=Integer.parseInt(str2);
            rango(numerador, denominador);
            cociente=numerador/denominador;
            respuesta=String.valueOf(cociente);
        }catch(ExcepcionIntervalo ex) {
            respuesta=ex.getMessage();
        }
        System.out.println(respuesta);
    }

    static void rango(int num, int den)throws ExcepcionIntervalo{
        if((num>100)||((den<-5)){
            throw new ExcepcionIntervalo("Números fuera de rango");
        }
    }
}

```

### La cláusula finally

Un bloque **finally** después del último **catch** se ejecutará siempre independientemente de que se produzca o no una excepción.

```

try{
    //Este código puede generar una excepción
}catch(Exception ex){
    //Este código se ejecuta cuando se produce una excepción
}finally{
    //Este código se ejecuta se produzca o no una excepción
}

```

## **EL INTERFACE *CLONEABLE***

## Duplicación de un objeto

La clase base *Object* del lenguaje Java tiene una función miembro denominada *clone*, que se redefine en la clase derivada para realizar una duplicación de un objeto de dicha clase.

Sea la clase *Punto*, para hacer una copia de un objeto de esta clase, se ha de agregar a la misma el siguiente código:

- se ha de implementar el interface *Cloneable*
- se ha de redefinir la función miembro *clone* de la clase base *Object*

```
public class Punto implements Cloneable{
    private int x;
    private int y;
    //constructores ...

    public Object clone(){
        Object obj=null;
        try{
            obj=super.clone();
        }catch(CloneNotSupportedException ex){
            System.out.println(" no se puede duplicar");
        }
        return obj;
    }
    //otras funciones miembro
}
```

En la redefinición de *clone*, se llama a la versión *clone* de la clase base desde **super**. Esta llamada se ha de hacer forzosamente dentro de un bloque **try... catch**, para capturar la excepción *CloneNotSupportedException* que nunca se producirá si la clase implementa el interface *Cloneable*.

Para crear un objeto *pCopia* que es una copia de otro objeto *punto* se escribe.

```
Punto punto=new Punto(20, 30);
Punto pCopia=(Punto)punto.clone();
```

La promoción (casting) es necesaria ya que *clone* devuelve un objeto de la clase base *Object* que ha de ser promocionado a la clase *Punto*.

Si hemos redefinido en la clase *Punto* la función miembro *toString* de la clase base *Object*, podemos comprobar que los objetos *punto* y *pCopia* guardan los mismos valores en sus miembros dato.

```
System.out.println("punto "+ punto);
System.out.println("copia "+ pCopia);
```

```
public class Punto implements Cloneable{
    private int x;
    private int y;

    public Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public Punto() {
        x=0;
        y=0;
    }

    public Object clone(){
        Object obj=null;
        try{
            obj=super.clone();
        }catch(CloneNotSupportedException ex){
            System.out.println(" no se puede duplicar");
        }
        return obj;
    }

    public void trasladar(int dx, int dy){
        x+=dx;
        y+=dy;
    }

    public String toString(){
        String texto="origen: (" +x+" , "+y+" )";
        return texto;
    }
}
```



# LA CLASE VECTOR

Un vector es similar a un array, la diferencia estriba en que un vector crece automáticamente cuando alcanza la dimensión inicial máxima. Además, proporciona métodos adicionales para añadir, eliminar elementos, e insertar elementos entre otros dos existentes.

### Crear un vector

Para usar la clase *Vector* hemos de poner al principio del archivo del código fuente la siguiente sentencia `import`

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

Cuando creamos un *vector* u objeto de la clase *Vector*, podemos especificar su dimensión inicial, y cuanto crecerá si rebasamos dicha dimensión.

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

Tenemos un vector con una dimensión inicial de 20 elementos. Si rebasamos dicha dimensión y guardamos 21 elementos la dimensión del vector crece a 25.

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

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

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

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

```
Vector vector=new Vector();
```

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

### Añadir elementos al vector

Hay dos formas de añadir elementos a un vector.

Podemos añadir un elemento a continuación del último elemento del vector, mediante la función miembro *addElement*.

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

Podemos también insertar un elemento en una determinada posición, mediante *insertElementAt*. El segundo parámetro indica el lugar que ocupará el nuevo objeto. Si tratamos de insertar un elemento en una posición que no existe todavía obtenemos una excepción del tipo *ArrayIndexOutOfBoundsException*. Por ejemplo, si tratamos de insertar un elemento en la posición 9 cuando el vector solamente tiene cinco elementos.

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

```
v.insertElementAt("tres", 2); // se comienza a contar desde la posición 0
```

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

```
Vector v=new Vector();
v.addElement("uno");
v.addElement("dos");
v.addElement("cuatro");
v.addElement("cinco");
v.addElement("seis");
v.addElement("siete");
v.addElement("ocho");
v.addElement("nueve");
v.addElement("diez");
v.addElement("once");
v.addElement("doce");
v.insertElementAt("tres", 2);
```

Para saber cuantos elementos guarda un vector, se llama a la función miembro *size*.

```
System.out.println("nº de elementos " +v.size());
```

Para saber la dimensión actual de un vector se llama a la función miembro *capacity*.

```
System.out.println("dimensión "+v.capacity());
```

Podemos eliminar todos los elementos de un vector, llamando a la función miembro *removeAllElements*. O bien, podemos eliminar un elemento concreto, por ejemplo el que guarda el string "tres".

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

Podemos eliminar dicho elemento, si especificamos su índice. (posición)

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

### Acceso a los elementos de un vector

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

Para acceder a todos lo elementos del vector, escribimos un código semejante al empleado para acceder a todos los elementos de un array.

```
for(int i=0; i<v.size(); i++){
    System.out.print(v.elementAt(i)+"\t");
}
```

Desde el objeto *enum* devuelto por la función miembro *elements* de la clase *Vector* llamamos a las funciones miembro *hasMoreElements* y *nextElement* de la clase *VectorEnumerator*.

La función *hasMoreElements* devuelve **true** mientras haya todavía más elementos que se puedan acceder en el vector *v*. Cuando se ha llegado al último elemento del vector, devuelve **false**.

La función *nextElement* devuelve una referencia al próximo elemento en la estructura de datos.

Para buscar objetos en un vector se puede usar las funciones del interface *Enumeration* y hacer una comparación elemento por elemento mediante *equals*, tal como vemos en la siguiente porción de código

```
Enumeration enum=v.elements();
while(enum.hasMoreElements()){
    String elemento=(String)enum.nextElement();
    if(elemento.equals("tres")){
        System.out.println("Encontrado tres");
        break;
    }
}
```

Podemos usar alternativamente, la función miembro *contains* para este propósito.

```
if(v.contains("tres")){
    System.out.println("Encontrado tres");
}
```

## **LA CLASE *STRINGTOKENIZER***

## Función

La clase *StringTokenizer* nos ayuda a dividir un string en substrings o tokens, en base a otro string (normalmente un carácter) separador entre ellos denominado delimitador.

Supongamos un string consistente en el nombre, y los dos apellidos de una persona separados por espacios en blanco. La clase *StringTokenizer* nos ayuda a romper dicho string en tres substrings basado en que el carácter delimitador es un espacio en blanco.

Un control área de texto, permite varias líneas de texto, cada línea está separada de la siguiente mediante un carácter nueva línea `\n` que se obtiene pulsando la tecla Enter o Retorno.

Mediante una función denominada *getText* obtenemos todo el texto que contiene dicho control. La clase *StringTokenizer* nos permite dividir el string obtenido en un número de substrings o tokens igual al número de líneas de texto, basado en que el carácter delimitador es `\n`.

Para usar la clase *StringTokenizer* tenemos que poner al principio del archivo del código fuente la siguiente sentencia `import`.

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

o bien

```
import java.util.StringTokenizer;
```

## Los constructores

Creamos un objeto de la clase *StringTokenizer* llamando a uno de los tres constructores que tiene la clase. Al primer constructor, se le pasa el string *nombre* que va a ser dividido teniendo en cuenta que el espacio en blanco es el delimitador por defecto.

```
String nombre="Angel Franco García";
StringTokenizer tokens=new StringTokenizer(nombre);
```

## Obtención de los tokens

Usamos las funciones miembro equivalentes *nextToken* y *hasMoreTokens*. Para extraer el nombre, el primer apellido y el segundo apellido en el primer ejemplo, escribiremos

```
String nombre="Angel Franco García";
StringTokenizer tokens=new StringTokenizer(nombre);
while(tokens.hasMoreTokens()){
    System.out.println(tokens.nextToken());
}
```

*hasMoreTokens* devuelve true mientras haya todavía strings separados por un espacio.

*nextToken* devuelve una referencia al próximo elemento en la estructura de strings o tokens.

Supongamos ahora que en un control área de texto introducimos los siguientes datos. Creamos un objeto *tokens* de la clase *StringTokenizer*, pasándole el string *strDatos* y el delimitador "\n"

```
String strDatos="6.3\n6.2\n6.4\n6.2";
StringTokenizer tokens=new StringTokenizer(strDatos, "\n");
```

Extraemos los tokens del string *strDatos* y convertimos cada uno de los substrings en un valor numérico de tipo **double** para guardarlos en el array *datos* del mismo tipo.

```
String str=tokens.nextToken();
datos[i]=Double.valueOf(str).doubleValue();
```

El número de tokens o de datos *nDatos* que hay en un string *strDatos*, se obtiene mediante la función miembro *countTokens*. Con este dato establecemos la dimensión del array *datos*.

```
int nDatos=tokens.countTokens();
double[] datos=new double[nDatos];
```

El código completo para extraer los tokens del string *strDatos* y guardarlos en un array *datos*, es el siguiente.

```
String strDatos="6.3\n6.2\n6.4\n6.2";
StringTokenizer tokens=new StringTokenizer(strDatos, "\n");
int nDatos=tokens.countTokens();
double[] datos=new double[nDatos];
int i=0;
while(tokens.hasMoreTokens()){
    String str=tokens.nextToken();
    datos[i]=Double.valueOf(str).doubleValue();
    System.out.println(datos[i]);
    i++;
}
```

# ARCHIVOS Y DIRECTORIOS



## La clase *File*

Antes de proceder al estudio de las clases que describen la entrada/salida vamos a estudiar la clase *File*, que nos proporciona información acerca de los archivos, de sus atributos, de los directorios, etc. También explicaremos como se crea un filtro mediante el interface *FilenameFilter* para obtener la lista de los archivos que tengan por ejemplo, la extensión **.java**.

La clase *File* tiene tres constructores

- *File(String path)*
- *File(String path, String name)*
- *File(File dir, String name)*

El parámetro *path* indica el camino hacia el directorio donde se encuentra el archivo, y *name* indica el nombre del archivo. Los métodos más importantes que describe esta clase son los siguientes:

- *String getName()*
- *String getPath()*
- *String getAbsolutePath()*
- *boolean exists()*
- *boolean canWrite()*
- *boolean canRead()*
- *boolean isFile()*
- *boolean isDirectory()*
- *boolean isAbsolute()*
- *long lastModified()*
- *long length()*
- *boolean mkdir()*
- *boolean mkdirs()*
- *boolean renameTo(File dest);*
- *boolean delete()*
- *String[] list()*
- *String[] list(FilenameFilter filter)*

Mediante un ejemplo explicaremos algunos de los métodos de la clase *File*.

Creamos un objeto *fichero* de la clase *File*, pasándole el nombre del archivo, en este caso, el nombre del archivo código fuente *ArchivoApp1.java*.

```
File fichero=new File("ArchivoApp1.java");
```

Si este archivo existe, es decir, si la función *exists* devuelve **true**, entonces se obtiene información acerca del archivo:

- *getName* devuelve el nombre del archivo
- *getPath* devuelve el camino relativo
- *getAbsolutePath* devuelve el camino absoluto.
- *canRead* nos indica si el archivo se puede leer.
- *canWrite* nos indica si el archivo se puede escribir
- *length* nos devuelve el tamaño del archivo, si dividimos la cantidad devuelta entre 1024 obtenemos el tamaño del archivo en KB.

```
if(fichero.exists()){
    System.out.println("Nombre del archivo "+fichero.getName());
    System.out.println("Camino      "+fichero.getPath());
    System.out.println("Camino absoluto  "+fichero.getAbsolutePath());
    System.out.println("Se puede escribir "+fichero.canRead());
    System.out.println("Se puede leer   "+fichero.canWrite());
    System.out.println("Tamaño      "+fichero.length());
}
```

La salida del programa es la siguiente:

```
Nombre del archivo ArchivoApp1.java
Camino      ArchivoApp1.java
Camino absoluto  c:\ejemplos\ArchivoApp1.java
Se puede escribir true
Se puede leer   true
Tamaño      1366
```

Para obtener la lista de los archivos del directorio actual se crea un nuevo objeto de la clase *File*

```
fichero=new File(".");
```

Para obtener la lista de los archivos que contiene este directorio se llama a la función miembro *list*, la cual nos devuelve un array de strings.

```
String[] listaArchivos=fichero.list();
for(int i=0; i<listaArchivos.length; i++){
    System.out.println(listaArchivos[i]);
}
```

La salida es la siguiente

```
ArchivoApp1.java
ArchivoApp1.~jav
Filtro.java
```

## Creación de filtros

Un filtro es un objeto de una clase que implemente el interface *FilenameFilter*, y tiene que redefinir la única función del interface denominada *accept*. Esta función devuelve un dato de tipo **boolean**.

En este caso, la hemos definido de forma que si el nombre del archivo termina con una determinada extensión devuelve **true** en caso contrario devuelve **false**. La función *endsWith* de la clase *String* realiza esta tarea tal como se ve en la porción de código que viene a continuación. La extensión se le pasa al constructor de la clase *Filtro* para inicializar el miembro dato *extension*.

```
import java.io.*;

public class Filtro implements FilenameFilter{
    String extension;

    Filtro(String extension){
        this.extension=extension;
    }
    public boolean accept(File dir, String name){
        return name.endsWith(extension);
    }
}
```

Para obtener la lista de archivos con extensión *.java* en el directorio actual, creamos un objeto de la clase *Filtro* y se lo pasamos a la función *list* miembro de la clase *File*.

```
String[] listaArchivos2=ficheros.list(new Filtro(".java"));
for(int i=0; i<listaArchivos2.length; i++){
    System.out.println(listaArchivos2[i]);
}
```

La salida es ahora la siguiente

```
ArchivoApp1.java
Filtro.java
```

# ENTRADA/SALIDA ESTÁNDAR

## Los objetos *System.in* y *System.out*

La entrada/salida estándar (normalmente el teclado y la pantalla, respectivamente) se definen mediante dos objetos.

La clase *System* tiene un miembro dato denominado *in* que es una instancia de la clase *InputStream* que representa al teclado o flujo de entrada estándar.

El miembro *out* de la clase *System* es un objeto de la clase *PrintStream*, que imprime texto en la pantalla (la salida estándar).

Para leer un carácter solamente tenemos que llamar a la función *read* desde *System.in*.

```
try{
    System.in.read();
}catch (IOException ex) { }
```

Obligatoriamente, el proceso de lectura ha de estar en un bloque **try..catch**.

Para leer un conjunto de caracteres hasta que se pulse la tecla RETORNO escribimos:

```
StringBuffer str=new StringBuffer();
char c;
try{
    /**
     * conectar el objeto System.in con un objeto de la clase
     * InputStreamReader para leer los caracteres tecleados por el usuario.
     */
    Reader entrada=new InputStreamReader(System.in);//introduccion d e caracteres

    while ((c=(char)System.in.read())!='\n'){
        str.append(c);
    }
}catch(IOException ex){}
```

La clase *StringBuffer* es una clase que nos permite crear strings. Contiene métodos para añadir nuevos caracteres a un buffer y convertir el resultado final en un string. Usamos *append* para añadir un carácter al final de un objeto de la clase *StringBuffer*.

Para convertir un objeto *str* de la clase *StringBuffer* a *String* se usa la función miembro *toString*. Esta llamada se hace de forma implícita cuando dicho objeto se le pasa a *System.out.println*.

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

Finalmente, se ha de hacer notar, que la función *read* miembro de *InputStream* devuelve un **int** que es promocionado a **char**.

Veamos un ejemplo completo a fondo :

```

package teclado1;

import java.io.*;

public class TecladoApp1 {
    public static void main(String[] args) {

        StringBuffer str=new StringBuffer();//STRING
        char c;//dato char

        try{

            Reader entrada=new InputStreamReader(System.in);//introduccion de caracteres

            /**
             * la función read de InputStream devuelve un int que es
             * promocionado a char.
             */
            while ((c=(char)entrada.read())!='\n'){//mientras caracter no sea salto de linea hacer
                /**
                 * Las principales funciones de StringBuffer son insert y append.
                 * Usamos la función append para añadir un carácter al final de
                 * un objeto(caracter o porcion de string) de la clase StringBuffer .
                 */
                str.append(c);//fabricacion del texto
            }

            /**
             * el miembro out de la clase System es un objeto de la clase
             * PrintStream, que imprime texto en la pantalla (la salida estándar).
             * Para convertir un objeto str de la clase StringBuffer a String se usa
             * la función miembro toString. Esta llamada se hace de forma automatica
             * cuando dicho objeto se le pasa a System.out.println.
             */
            catch(IOException ex){}
            System.out.println(str);

            /**
             * La clase System tiene un miembro dato denominado in que es una
             * instancia de la clase InputStream que representa al teclado o flujo
             * de entrada estándar.
             * Para leer un carácter solamente tenemos que llamar a la función
             * read desde System.in.
             * el proceso de lectura ha de estar en un bloque try..catch.
             *
             * esta porcion de codigo es para detener la aplicacion pulsando retorno
             */
            try {System.in.read();}
            catch (Exception e) { }

        }
    }
}

```

## La clase *Reader*

Como hemos mostrado se puede conectar el objeto *System.in* con un objeto de la clase *InputStreamReader* para leer los caracteres tecleados por el usuario.

Esta conexión se realiza mediante la sentencia

```
Reader entrada=new InputStreamReader(System.in);
```

Para leer una sucesión de caracteres se emplea un código similar

```
StringBuffer str=new StringBuffer();
char c;
try{
    Reader entrada=new InputStreamReader(System.in);
    while ((c=(char)entrada.read())!='\n'){
        str.append(c);
    }
}catch(IOException ex){}
```

Para imprimir los caracteres leídos se escribe como en la sección anterior

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

Podemos usar la función *read* de otra manera para leer el conjunto de caracteres tecleados por el usuario.

```
char[] buffer=new char[255];
try{
    Reader entrada=new InputStreamReader(System.in);
    int numBytes=entrada.read(buffer);
    System.out.println("Número de bytes leídos "+numBytes);
}catch(IOException ex){ }
```

En esta segunda porción de código, se lee un conjunto de caracteres hasta que se pulsa la tecla RETORNO, los caracteres se guardan en el array *buffer*. La función *read* devuelve el número de caracteres leídos.

Para imprimir los caracteres leídos se crea un objeto *str* de la clase *String* a partir de un array de caracteres *buffer*, empleando uno de los constructores de dicha clase. A continuación, se imprime el string *str*.

```
String str=new String(buffer);
System.out.println(str);
```

## Ejemplo a fondo.

```

package teclado2;
import java.io.*;

public class TecladoApp2 {
    public static void main(String[] args) {
        char[] buffer=new char[106];//array de caracteres buffer de lectura

        System.out.println("Introduce una línea de texto y pulsa intro ");

        try{
            Reader entrada=new InputStreamReader(System.in);//lectura de los datos se llena el buffer

            /**
             * se lee un conjunto de caracteres hasta que se pulsa la tecla RETORNO,
             * los caracteres se guardan en el array b uffer.
             * La función read devuelve el número de caracteres leídos que se
             * guardan en una variable entera que mostrara los bytes leidos
             *
             * toda esta relacion en (entrada.read(buffer))
             */
            int numBytes=entrada.read(buffer);//read devuelve el numero de caracteres leidos
            System.out.println("Número de bytes leídos "+numBytes);}

        catch(IOException ex){System.out.println("Error entrada/salida");}

        System.out.println("La línea de texto que has escrito es ");

        /**
         * Para imprimir los caracteres leídos se crea un objeto str de la clase
         * String a partir de un array de caracteres buffer, empleando uno de
         * los constructores de dicha clase.
         */
        String str=new String(buffer);
        System.out.println(str);

    }
}

```



# **ENTRADA/SALIDA A UN ARCHIVO EN DISCO**

## Lectura de un archivo

Creamos un objeto *entrada* de la clase *FileReader* en vez de *InputStreamReader*. El final del archivo viene dado cuando la función *read* devuelve -1. El resto del código es similar.

```
FileReader entrada=null;//creamos objeto sin iniciar
StringBuffer str=new StringBuffer();//objeto str

try {
    entrada=new FileReader("doc.txt");//objeto se inicia
    int c;

    /**
     * en esta bucle se va leyendo el texto del archivo de entrada
     * mediante la funcion read y despues se va formando el texto que se
     * va leyendo en el String str mediante la funcion append
     */
    while((c=entrada.read())!=-1){
        str.append((char)c);

        System.out.println(str);//imprime el contenido del objeto str de la clase StringBuffer.
        System.out.println("-----");
    }

    /**
     * lanza una excepcion en caso de no encontrarse el archivo
     * en el directorio especificado en el directorio del proyecto
     */
    catch (IOException ex) {
        System.out.println("mensaje de error");
    }
}
```

Una vez concluido el proceso de lectura, es conveniente cerrar el flujo de datos, esto se realiza en una cláusula **finally** que siempre se llama independientemente de que se produzcan o no errores en el proceso de lectura/escritura.

```
}finally{
    if(entrada!=null){
        try{
            entrada.close();
        }catch(IOException ex){}
    }
}
```

El código completo de este ejemplo es el siguiente:

```

package archivo1;
import java.io.*;

public class ArchivoApp1 {

    /**
     * Creamos un objeto entrada de la clase FileReader en vez de InputStreamReader.
     * El final del archivo viene dado cuando la función read devuelve -1.
     */
    public static void main(String[] args) {
        FileReader entrada=null;//creamos objeto sin iniciar
        StringBuffer str=new StringBuffer();//objeto str

        try {
            entrada=new FileReader("doc.txt");//objeto se inicia
            int c;

            /**
             * en esta bucle se va leyendo el texto del archivo de entrada
             * mediante la funcion read y despues se va formando el texto que se
             * va leyendo en el String str mediante la funcion append
             */
            while((c=entrada.read())!=-1){
                str.append((char)c);}

            System.out.println(str);//imprime el contenido del objeto str de la clase StringBuffe r.
            System.out.println("-----");
        }

        /**
         * lanza una excepcion en caso de no encontrarse el archivo
         * en el directorio especificado en el directorio del proyecto
         */
        catch (IOException ex) {
            System.out.println("el archivo no se encuentra en la carpeta del proyecto");
        }

        /**
         * Una vez concluido el proceso de lectura, es conveniente cerrar el
         * flujo de datos, esto se realiza en una cláusula finally que
         * siempre se llama independientemente de que se produzcan o no
         * errores en el proceso de lectura/escritura.
         *
         * la manera de cerrar el flujo de datos es siempre igual.
         */
        finally{
            //con este if se cierra el flujo de datos
            if(entrada!=null){
                try{entrada.close();}
                catch(IOException ex){}
            }
            System.out.println("el bloque finally siempre se ejecuta");
        }
    }
}

```

## Lectura/escritura

Los pasos para leer y escribir en disco son los siguientes:

1. Se crean dos objetos de las clases *FileReader* y *FileWriter*, llamando a los respectivos constructores a los que se les pasa los nombres de los archivos o bien, objetos de la clase *File*, respectivamente

```
entrada=new FileReader("doc2.txt");//objeto inicializado entrada
salida=new FileWriter("copia2.txt");//objeto inicializado salida
```

2. Se lee mediante *read* los caracteres del flujo de entrada, hasta llegar al final (la función *read* devuelve entonces -1), y se escribe dichos caracteres en el flujo de salida mediante *write*.

```
int c;
while((c=entrada.read())!=-1){//se lee la entrada del flujo de entrada
    salida.write(c);//se escribe la salida en el flujo de salida
    str.append((char)c);//se forma el texto de la entrada en un string
```

3. Finalmente, se cierran ambos flujos llamando a sus respectivas funciones *close* en bloques **try..catch**

```
finally{
    //cerrar los flujos de datos
    //cierre del flujo de entrada
    if(entrada!=null){
        try{entrada.close();}
        catch(IOException ex){}
    }
    //cierre del flujo de salida
    if(salida!=null){
        try{salida.close();}
        catch(IOException ex){}
    }
}
```

El código completo de este ejemplo que crea un archivo copia del original, es el siguiente

```
package archivo2;
import java.io.*;

/**
 *para leer y escribir en disco
 */
public class ArchivoApp2 {
    public static void main(String[] args) {
        FileReader entrada=null;//objeto sin iniciar
        FileWriter salida=null;//objeto sin iniciar
        StringBuffer str=new StringBuffer();//objeto de la clase StringBuffer

        /**
         * Se crean dos objetos de las clases FileReader y FileWriter,
         * llamando a los respectivos constructores a los que se les pasa
         * los nombres de los archivos o bien, objetos de la clase File,
         * respectivamente
         */
        try {
            entrada=new FileReader("doc2.txt");//objeto inicializado entrada
            salida=new FileWriter("copia2.txt");//objeto inicializado salida
```

```

/**
 * Se lee mediante read los caracteres del flujo de entrada,
 * hasta llegar al final (la función read devuelve entonces -1),
 * y se escribe dichos caracteres en el flujo de salida mediante
 * write.
 */
int c;
while((c=entrada.read())!=-1){//se lee la entrada del flujo de entrada
    salida.write(c);//se escribe la salida en el flujo de salida
    str.append((char)c);} //se forma el texto de la entrada en un string

System.out.println(str);//se imprime el contenido de entrada
System.out.println("-----");

}

/**
 * lanza una excepcion en caso de no encontrarse el archivo de
 * entrada en el directorio especificado en el CLASSPATH
 *
 * se puede especificar la frase de salida para la excepci on
 * ademas de incluir con +ex la declaracion de la excepcion
 */
catch (IOException ex) {
    System.out.println("el archivo de entrada no se encuentr a en la carpeta "+ex);}

/**
 * Finalmente, se cierran ambos flujos llamando a sus respectivas
 * funciones close en bloques try..catch
 */
finally{
//cerrar los flujos de datos
//cierre del flujo de entrada
if(entrada!=null){
    try{entrada.close();}
    catch(IOException ex){}
}
//cierre del flujo de salida
if(salida!=null){
    try{salida.close();}
    catch(IOException ex){}
}

System.out.println("el bloque finally siempre se ejecuta");
}

}
}

```

# **LEER Y ESCRIBIR DATOS PRIMITIVOS**

## Los flujos de datos *DataInputStream* y *DataOutputStream*

La clase **DataInputStream** es útil para leer datos del tipo primitivo. Esta clase tiene un sólo constructor que toma un objeto de la clase *InputStream* o sus derivadas como parámetro.

Se crea un objeto de la clase *DataInputStream* vinculándolo a un un objeto *FileInputStream* para leer desde un archivo en disco denominado pedido.txt..

```
FileInputStream fileIn=new FileInputStream("pedido.txt");  
DataInputStream entrada=new DataInputStream(fileIn);
```

o en una sola línea

```
DataInputStream entrada=new DataInputStream(new FileInputStream("pedido.txt"));
```

La clase *DataInputStream* define diversos métodos *read* que son variaciones del método *read* de la clase base para leer datos de tipo primitivo

```
boolean readBoolean();  
byte readByte();  
int readUnsignedByte();  
short readShort();  
int readUnsignedShort();  
char readChar();  
int readInt();  
String readLine();  
long readLong();  
float readFloat();  
double readDouble();
```

La clase **DataOutputStream** es útil para escribir datos del tipo primitivo. Esta clase tiene un sólo constructor que toma un objeto de la clase *OutputStream* o sus derivadas como parámetro.

Se crea un objeto de la clase *DataOutputStream* vinculándolo a un un objeto *FileOutputStream* para escribir en un archivo en disco denominado pedido.txt..

```
FileOutputStream fileOut=new FileOutputStream("pedido.txt");  
DataOutputStream salida=new DataOutputStream(fileOut);
```

o en una sola línea

```
DataOutputStream salida=new DataOutputStream(new FileOutputStream("pedido.txt"));
```

La clase *DataOutputStream* define diversos métodos *write* que son variaciones del método *write* de la clase base para escribir datos de tipo primitivo

```
void writeBoolean(boolean v);  
void writeByte(int v);  
void writeBytes(String s);  
void writeShort(int v);  
void writeChars(String s);  
void writeChar(int v);  
void writeInt(int v);  
void writeLong(long v);  
void writeFloat(float v);  
void writeDouble(double v);
```

## EJEMPLO A FONDO

```

package archivo;
import java.io.*;

/**
 *En este ejemplo, se escriben datos a un archivo y se leen del mismo, que corresponden
 *a un pedido
 */
public class ArchivoApp {
    public static void main(String[] args) throws IOException {

        //ESCRITURA

        DataOutputStream salida=new DataOutputStream(new FileOutputStream("pedido.txt"));
        //precio de cada item, un dato de tipo double.
        double[] precios={1350, 400, 890, 6200, 8730};
        //número de unidades, un dato del tipo primitivo int
        int[] unidades={5, 7, 12, 8, 30};
        //descripción del item, un objeto de la clase String
        String[] descripciones={"paquetes de papel", "lápices", "bolígrafos", "carteras", "mesas"};

        /**
         * La clase DataOutputStream define diversos métodos writeXXX
         * que son variaciones del método write
         *
         * Se escribe en el flujo de salida los distintos datos llamando a las
         * distintas versiones de la función writeXXX
         *
         * Para leer bien los datos, el string ha de separarse del siguiente dato con un
         * carácter nueva línea '\n'.
         * Esto no es necesario si el string se escribe en el último lugar, después de los números.
         * el carácter tabulador como separador no es estrictamente necesario.
         *
         * con writeString hemos notado que se imprimen las palabras separadas
         * las letras con espacios en blanco por eso hemos optado por ponerle
         * writeBytes
         */
        for (int i=0; i<precios.length; i++) { //flujo de salida
            salida.writeBytes(descripciones[i]); //escribe un string pero
            //en este caso le hemos puesto que escriba una cadena de Bytes
            salida.writeChar('\n'); //escribe un caracter nueva linea
            salida.writeInt(unidades[i]); //escribe un entero
            salida.writeChar('\t'); //escribe un caracter separador tabulador
            salida.writeDouble(precios[i]); //escribe un decimal

            salida.close(); //se cierra flujo de salida
        }
    }
}

```



```

//LECTURA

DataInputStream entrada= new DataInputStream(new FileInputStream("pedido.txt"));
double precio;//variable decimal
int unidad;//variable entera
String descripcion=null;//objeto string sin iniciar
double total=0.0;//contador decimal

try {
/**
 * La clase DataInputStream define diversos métodos read que son
 * variaciones del método read de la clase base para leer datos de tipo
 * primitivo
 *
 * cuando readLine toma el valor null (no hay más que leer) y se sale del bucle
 * while y Se imprime "Final de archivo"
 */
while ((descripcion=entrada.readLine())!=null) {
    unidad=entrada.readInt();//lectura array de enteros que se guarda en unidad
    entrada.readChar();//lee el carácter tabulador
    precio=entrada.readDouble();//lectura array de decimales que se guarda en precio
    //se van leyendo los datos del archivo y se imprimen
    System.out.println("has pedido "+unidad+" "+descripcion+" a "+precio+" pts.");
    //se calcula el precio total del pedido.
    total=total+unidad*precio;}

    System.out.println("Final del archivo");
}

/**
 * el bloque catch no se ejecuta ya que hay una sentencia que hace que cuando
 * readLine tome el valor de null se salga del bucle
 * si no hubiera una sentencia para salir del bucle en un momento dado pues
 * se seguiria leyendo hasta el final del archivo y cuando llegase al final
 * automaticamente se ejecutaria el bloque catch
 */
catch (EOFException e) {
    System.out.println("Excepción cuando se alcanza el final del archivo");}

//se imprime el total precio
System.out.println("por un TOTAL de: "+total+" pts.");

entrada.close();//se cierra el flujo de entrada
}
}

```

# **LEER Y ESCRIBIR OBJETOS**

Java ha añadido una interesante faceta al lenguaje denominada serialización de objetos que permite convertir cualquier objeto cuya clase implemente el interface *Serializable* en una secuencia de bytes que pueden ser posteriormente leídos para restaurar el objeto original.

## El interface *Serializable*

Un objeto se puede serializar si implementa el interface *Serializable*. Este interface no declara ninguna función miembro, se trata de un interface vacío.

```
import java.io.*;
public interface Serializable{
}
```

Para hacer una clase serializable simplemente ha de implementar el interface *Serializable*.

```
package a_serializable_basico;

public class Lista implements java.io.Serializable{
    private int[] x;    //array de datos
    private int n;     //dimensión array

    public Lista(int[] x) { //constructor
        this.x=x; //pasando datos del array a dato miembro de la clase
        n=x.length; //longitud del array
        ordenar(); //llamada a la funcion ordenar
    }

    private void ordenar(){ //funcion ordenar ordena datos del array
        int aux; //contenedor auxiliar
        for(int i=0; i<n-1; i++){
            for(int j=i+1; j<n; j++){
                if(x[j]<x[i]){ //de menor a mayor
                    aux=x[j];
                    x[j]=x[i];
                    x[i]=aux;}}
        }

    public double valorMedio(){
        int suma=0; //sumatorio inicializado a 0
        for(int i=0; i<n; i++){ //sumatorio de elementos del array
            suma+=x[i];}
        return (double)suma/n; //retorno de la media
    }

    public String toString(){ //conversion del array en texto
        String texto=""; //texto sin iniciar
        for(int i=0; i<n; i++){ //inicializando texto del array
            texto+="\t"+x[i];}
        return texto; //retorno del texto
    }
}
```

## Lectura/escritura

Dos flujos de datos *ObjectInputStream* y *ObjectOutputStream* están especializados en la lectura y escritura de objetos. El comportamiento de estos dos flujos es similar a sus correspondientes que procesan flujos de datos primitivos *DataInputStream* y *DataOutputStream*, que hemos visto.

**Escribir objetos** al flujo de salida *ObjectOutputStream* requiere los siguientes pasos:

1. Creamos un objeto de la clase *Lista*  

```
Lista lista1= new Lista(new int[]{12, 15, 11, 4, 32});
```
2. Creamos un flujo de salida a disco, pasándole el nombre del archivo en disco o un objeto de la clase *File*.  

```
FileOutputStream fileOut=new FileOutputStream("media.obj");
```
3. El flujo de salida *ObjectOutputStream* es el que procesa los datos y se ha de vincular a un objeto *fileOut* de la clase *FileOutputStream*.  

```
ObjectOutputStream salida=new ObjectOutputStream(fileOut);
```

o en una sólo línea

```
ObjectOutputStream salida=new ObjectOutputStream(new FileOutputStream("media.obj"));
```

4. El método *writeObject* escribe los objetos al flujo de salida y los guarda en un archivo en disco. Por ejemplo, un string y un objeto de la clase *Lista*.  

```
salida.writeObject("guardar este string y un objeto \n");
salida.writeObject(lista1);
```
5. Finalmente, se cierra el flujo de salida.  

```
salida.close();
```

**El proceso de lectura** es paralelo al proceso de escritura, por lo que leer objetos del flujo de entrada *ObjectInputStream* requiere los siguientes pasos.

1. Creamos un flujo de entrada a disco, pasándole el nombre del archivo en disco o un objeto de la clase *File*.  

```
FileInputStream fileIn=new FileInputStream("media.obj");
```
2. El flujo de entrada *ObjectInputStream* es el que procesa los datos y se ha de vincular a un objeto *fileIn* de la clase *FileInputStream*.  

```
ObjectInputStream entrada=new ObjectInputStream(fileIn);
```

o en una sólo línea

```
ObjectInputStream entrada=new ObjectInputStream(new FileInputStream("media.obj"));
```

3. El método *readObject* lee los objetos del flujo de entrada, en el mismo orden en el que ha sido escritos. Primero un string y luego, un objeto de la clase *Lista*.  

```
String str=(String)entrada.readObject();
Lista obj1=(Lista)entrada.readObject();
```
4. Se realizan tareas con dichos objetos,  

```
System.out.println(" Valor medio "+obj1.valorMedio());
```
5. Finalmente, se cierra el flujo de entrada *entrada.close()*;

**Ejemplo**

```

package a_serializable_basico;

import java.io.*;

public class ArchivoApp {
    public static void main(String[] args) {
        //se pasa el objeto array al constructor de la clase Lista
        Lista lista1= new Lista(new int[]{12, 15, 11, 4, 32});

        try {
            /*
             * -----Escribir objetos al flujo de salida ObjectOutputStream ----
             *
             */
            ObjectOutputStream salida= new ObjectOutputStream(new
FileOutputStream("media.obj"));
            salida.writeObject("guardar este string y un objeto\n");//escribe un string
            salida.writeObject(lista1);//escribe un objeto array
            salida.close();//se cierra flujo de salida

            /*
             * -----lectura-----
             *
             */
            ObjectInputStream entrada= new ObjectInputStream(new FileInputStream("media.obj"));
            String str=(String)entrada.readObject();//lectura del string
            Lista obj1=(Lista)entrada.readObject();//lectura array
            System.out.println("Valor medio "+obj1.valorMedio());//mostrar en pantalla valor medio
            System.out.println("-----");
            System.out.println(str+obj1);//mostrar en pantalla string y array
            System.out.println("-----");
            entrada.close();//cierre de flujo de entrada

            //se puede fundir en una catch Exception
            catch (IOException ex) {//lanza una excepcion sino encontrase el archivo de salida el flujo de
entrada
                System.out.println(ex);}

            catch (ClassNotFoundException ex) {//lanza excepcion al no encontrar la clase
                System.out.println(ex);}

        }
    }
}

```

## El modificador *transient*

Cuando un miembro dato de una clase contiene información sensible, hay disponibles varias técnicas para protegerla. Incluso cuando dicha información es privada (el miembro dato tiene el modificador **private**) una vez que se ha enviado al flujo de salida alguien puede leerla en el archivo en disco o interceptarla en la red.

El modo más simple de proteger la información sensible, como una contraseña (password) es la de poner el modificador **transient** delante del miembro dato que la guarda.

La clase *Cliente* tiene dos miembros dato, el nombre del cliente y la contraseña o password.

Se redefine la función *toString* miembro de la clase base *Object*. Esta función devolverá el nombre del cliente y la contraseña. En el caso de que el miembro *password* guarde el valor **null** se imprimirá el texto (no disponible).

En el cuadro que sigue se muestra el código que define la clase *Cliente*.

```
package b_modificador_transient;

public class Cliente implements java.io.Serializable{
    private String nombre;//string nombre del cliente
    private transient String passWord;//string contraseña

    public Cliente(String nombre, String pw) { //constructor de la clase
        this.nombre=nombre;//apunta al miembro dato de la clase
        this.passWord=pw;//apunta al miembro dato de la clase
    }

    public String toString(){
        String texto="(contraseña no disponible)\nla información sensible guardada en el
miembro dato password que tiene por \nmodificador transient no ha sido guardada en el
archivo.\n";
        if(passWord==null){texto+=nombre;}
        return texto;
    }
}
```

En el cuadro siguiente se muestra los pasos para guardar un objeto de la clase *Cliente* en el archivo *cliente.obj*. Posteriormente, se lee el archivo para reconstruir el objeto *obj1* de dicha clase.

```

package b_modificador_transient;

import java.io.*; //se importa la libreria de java para flujos de entrada y salida

public class ArchivoApp{
    public static void main(String[] args) {
        /*
         * se crea un nuevo objeto en donde se pasan los parametros al unico
         * constructor de la clase ala que pertenece el objeto creado
         */
        Cliente cliente=new Cliente("Paco", "xyz");

        try{
            /*
             * -----ESCRITURA-----
             * -guardar un objeto de la clase Cliente en el archivo cliente.obj. -
             * Se crea un flujo de salida (objeto salida de la clase
             * ObjectOutputStream) y se asocia con un objeto de la clase
             * FileOutputStream para guardar la información en el archivo
             * cliente.obj.
             *
             * Se escribe el objeto cliente en el flujo de salida
             * mediante writeObject.
             * para strings extras o datos numericos tambien utilizaremos
             * writeObject ya se utiliza el objeto salida que pertenece a la
             * clase ObjectOutputStream
             */
            ObjectOutputStream salida=new ObjectOutputStream(new
FileOutputStream("cliente.obj"));
            salida.writeObject("Datos del cliente\n");//se escribe tambien un string antes
            salida.writeObject(cliente);//Se escribe el objeto cliente en el flujo de salida mediante
writeObject.
            salida.close();//cierre del flujo de salida

```

```

/*
 * -----LECTURA-----
 * --reconstruir el objeto obj1 de la clase Cliente --
 * Se crea un flujo de entrada (objeto entrada de la
 * clase ObjectInputStream) y se asocia con un objeto
 * de la clase FileInputStream para leer la información
 * que guarda el archivo cliente.obj.
 *
 * Se lee el objeto cliente en el flujo de salida mediante readObject.
 *
 * en la clase Cliente la información sensible guardada en el miembro
 * dato password que tiene por modificador transient no se podrá leer
 * ya que no habra sido guardada en el archivo antes creado ya que
 * esta protegido con el modificador transient.
 */
ObjectInputStream entrada= new ObjectInputStream(new FileInputStream("cliente.obj"));
String str=(String)entrada.readObject();//objeto string que guarda la linea de texto de
cliente.obj
Cliente obj1=(Cliente)entrada.readObject();//objeto que lee el objeto que guarda el
nombre y el password
System.out.println("-----");
//Se imprime en la pantalla dicho objeto llamando implícitamente a su
//función miembro toString.
System.out.println(str+obj1);
System.out.println("-----");
entrada.close();//se cierra flujo de entrada

//se puede fundir en una catch Exception
catch (IOException ex) {//lanza una excepcion sino encontrase el archivo de salida el flujo
de entrada
    System.out.println("Archivo para lectura no encontrado");}

catch (ClassNotFoundException ex) {//lanza excepcion al no encontrar la clase
    System.out.println("Clase Cliente no encontrada");}

}
}

```

La salida del programa es

```

-----
Datos del cliente
(contraseña no disponible)
la información sensible guardada en el miembro dato password que tiene por
modificador transient no ha sido guardada en el archivo.
Paco
-----

```

Lo que nos indica que la información sensible guardada en el miembro dato *password* que tiene por modificador **transient** no ha sido guardada en el archivo. En la reconstrucción del objeto *obj1* con la información guardada en el archivo el miembro dato *password* toma el valor **null**.