www.softdownload.com.ar

Curso de Java

Autor : Ricardo Amezua Martinez

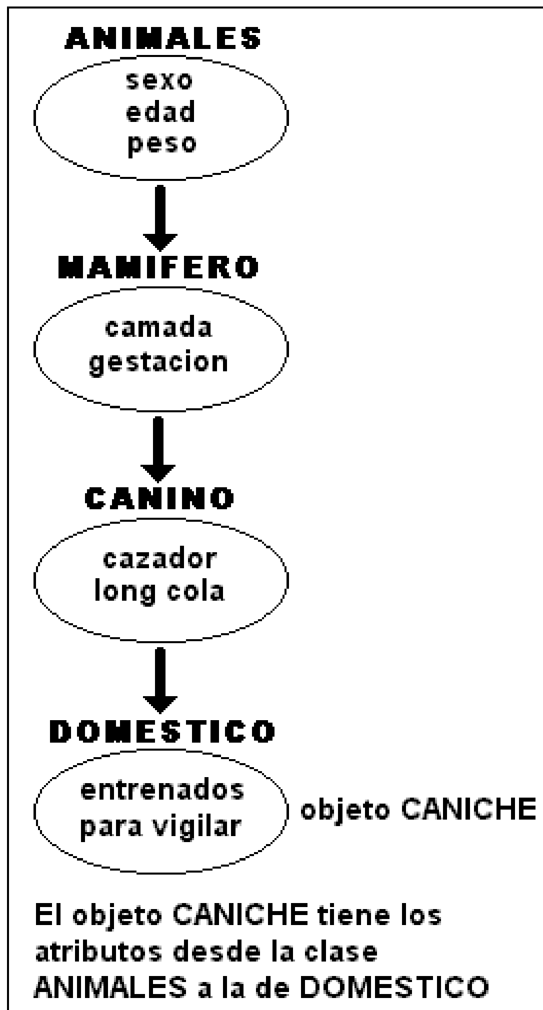
INTRODUCCIÓN

Los lenguajes estructurados se basan en estructuras de control bloques de código y subrutinas independientes que soportan recursividad y variables locales. La programación orientada a objetos coge las mejores ideas de la programación estructurada y los combina con nuevos conceptos de organización.

La programación orientada objetos permite descomponer un programa en grupos relacionados. Cada subgrupo pasa a ser un objeto autocontenido con sus propias instrucciones y datos. Tres características de los lenguajes orientado a objetos son, Encapsulación, Polimorfismo y la Herencia.

ENCAPSULAMIENTO: Como un envoltorio protector alrededor del código y los datos que se manipulan. El envoltorio define el comportamiento y protege el código y los datos para evitar que otro código acceda a ellos de manera arbitraria. El poder es que todo el mundo conoce como acceder a él y pueden utilizarlo de este modo independientemente de los detalles de implementación. En Java la base del encapsulamiento es la clase, (conjunto de objetos que comparten la misma estructura y comportamiento). Una clase se define mediante métodos que operan sobre esos datos. Métodos es un mensaje para realizar alguna acción en un objeto.

HERENCIA: Objetos que se relacionan entre ellos de una manera jerárquica. Es decir a partir de una clase donde están los atributos generales (superclase) se definen otras clases con atributos más específicos (subclase).



POLIMORFISMO: A los métodos que actúan sobre los objetos se les pasa información. Estos parámetros representan los valores de entrada a una función. El Polimorfismo significa que un método tenga múltiples implementaciones que se seleccionan en base a que tipo de objeto se le pasa.

1. PRIMER PROGRAMA

Los ficheros se almacenan con la extensión. *JAVA*, al compilar el código fuente, crea un uno con extensión *CLASS*. Java requiere que todo el código resida dentro de una clase con nombre. El nombre del fichero debe ser el mismo que el nombre de la clase donde esta la función MAIN.

```
public class holamundo{
    public static void main(String args[ ]){
            System.out.println("HOLA MUNDO");
    }
}
```

Para compilar necesitamos el compilador *JAVAC* y posteriormente para ejecutar la aplicación utilizamos el fichero *JAVA*.

```
C:\> javac holamundo.java

C:\> java holamundo
```

2. VARIABLES

Unidad básica de almacenamiento, la creación es la combinación de un identificador, un tipo y un ámbito. Tenemos 8 tipos de variables, la sintaxis para la creación.

tipo identificador = valor;

TIPO	*TAMAÑO*	*RANGO*
byte	8 bits	Valores numéricos de –128 a 127
short	16 bits	Valores numéricos de –32.768 a 32.767
int	32 bits	Valores numéricos de –2.147.483.648 a 2.147.483.647
long	64 bits	Valores numéricos sin límite.
float	32 bits	Valores numéricos hasta 38 cifras
double	64 bits	Valores numéricos hasta 308 cifras
char	16 bits	Valores alfanuméricos
String	Según long	Se utiliza para cadenas de caracteres
boolean	8 bits	Solo admite TRUE o FALSE
matriz[ ]	Según long	Agrupar variables del mismo tipo.

EJEMPLO: Programa utilizando variables.

```
class hola{
   public static void main(String args[ ]){
            int x,y;
            x=42;
            y=12;
            System.out.print("X= " +x);
            System.out.println("Y= " +y);
   }
}
```

3. MATRICES

Grupo de variables con el mismo nombre y tipo. La manera de referirse a cada uno de los elementos de una matriz es mediante su índice. Los tipos de la matrices son los mismos que el de las variables. Tenemos 2 tipos de array, unidimensional y bidimensional.

UNIDIMENSIONALES:

```
tipo nombre_array[]=new tipo[nº];

tipo nombre_array[]={valores};
```

BIDIMENSIONALES:

```
tipo nombre_array[][]=new tipo[nº][nº];

tipo nombre_array[][]={valores};
```

EJEMPLO: Se declara una matriz para guardar los días que tienen los meses del año. Luego mostramos los días del mes de abril.

```
class meses{
    public static void main(String args[]){
      int mesdias[]={31,28,31,30,31,30,31,31,30,31,30,31};
      System.out.pritnln("Abril " + mesdias[3] + "días");
    }
}
```

EJEMPLO: En este programa se utiliza un array bidimensional.

```
class multima{
  public static void main (String args[]){
          int m[][]=new int[3][3];
          m[0][0]=1;
          m[1][1]=1;
          m[2][2]=1;

          System.out.println(m[0][0]+" "+m[0][1]+" "+m[0][2]);
          System.out.println(m[1][0]+" "+m[1][1]+" "+m[1][2]);
          System.out.println(m[2][0]+" "+m[2][1]+" "+m[2][2]);
  }
}
```

4. OPERADORES

Aritméticos	*DESCRIPCION*
+	Suma
-	Resta
*	Multiplica
/	Divide
%	Devuelve el resto de una división
++	Incrementa en 1
--	Decremento en 1

*Solo se pueden utilizar para variables de tipo numérico.

Relacionales	*DESCRIPCION*
= =	Igual
!=	Distinto
>	Mayor que
<	Menor que
>=	Mayor o igual
<=	Menor o igual

*Para todo tipo de datos.

Lógicos	*DESCRIPCION*
&&	*condición* Y *condición*
| |	*condición* O *condición*
!	Negación de la condición

5. INTRODUCCIÓN DE DATOS

Para introducir valores en aplicaciones que trabajan bajo MS-DOS se utiliza el argumento (matriz de String) de la función main. Estos valores se introducción a la hora de ejecutar el programa, es decir, desde el prompt de MS-DOS. Hay que tener en cuenta que estos valores serán siempre de tipo String y si queremos realizar operaciones matemáticas deberemos transformar a valor numérico. En los applet esta manera de introducir datos quedará desfasada.

EJEMPLO:

```
class nombre{
        public static void main(String clientes[ ]){
                System.out.println("Hola " +clientes[0]);
                System.out.println("Hola " +clientes[1]);
        }
}
```

A la hora de ejecutar : c:\>java nombre **Pepe Antonio**

6. CONVERSIÓN DE DATOS

En Java hay una serie de funciones que realizan la conversión entre los distintos tipos de datos. Las conversiones pueden ser, de número a carácter, de número a cadena, entre números, de carácter a número, de carácter a cadena y de cadena a número. Este tipo de funciones se utilizará también en los applet.

DE NÚMERO A CARÁCTER

```
var_char=Character.forDigit(var_num,base);
```

DE NÚMERO A CADENA: El dato a convertir debe ser un objeto de una clase numérica.

```
clase_num objeto=new clase_num(valor);
var_String=objeto.toString();

var_String=String.valueOf(var_numerica);
```

ENTRE NUMEROS: El dato a convertir debe se un objeto de una clase numérica.

clase_num objeto=new clase_num(valor);
var_tipo=objeto.*tipo*Value();

```
Float F=new Float(3.1416);
int i=F.intValue();
long l=F.longValue();
float f=F.floatValue();
double d=F.doubleValue();
```

DE CARÁCTER A NÚMERO

var_num=Character.digit(var_char,base);

DE CARÁCTER A CADENA: El char a convertir debe ser un objeto de la clase Character.

Character objeto=new Character('letra');
var_String=objeto.toString();

DE CADENA A NÚMERO: El dato al que convertimos debe ser un objeto.

clase_num Objeto=new Clase_num(var_String);
var_num=objeto.*tipo*Value();

EJEMPLO:

```
    class conver2{
 public static void main(String numero[]){
    int n1=Character.digit('7',10);
    int n2=1;
    Character letra=new Character('z');
    double n3=150.50;
    String cad1="Numero";

    String cad=String.valueOf(n3);
    String cad2=letra.toString();

    System.out.println(cad1+cad+cad2);
    System.out.println(n1+n2);

    char nletra=Character.forDigit(n2,10);

    System.out.print(n1+" "+nletra);
}
}
```

EJEMPLO:

```
    class conver{
 public static void main(String numero[]){
   Integer entero=new Integer(numero[0]);
   double n1=entero.doubleValue();
   double n2=150.50;
   System.out.print(n1+n2);
}
```

```
}
```

7. SENTENCIAS DE CONTROL

Es la manera que tiene un lenguaje de programación de provocar que el flujo de la ejecución avance y se ramifique en función de los cambios de estado de los datos.

IF-ELSE: La ejecución atraviese un conjunto de estados boolean que determinan que se ejecuten distintos fragmentos de código. La cláusula else es opcional, la expresión puede ser de cualquier tipo y más de una (siempre que se unan mediante operadores lógicos).

```
if (expresion-booleana)
        Sentencia1;
else
        Sentencia2;
```

SWITCH: Realiza distintas operaciones en base al valor de una única variable o expresión. Es una sentencia muy similar a if-else, pero esta es mucho más cómoda y fácil de comprender.

```
switch (expresión){
        case valor1:
                sentencia;
                break;
        case valor2:
                sentencia;
                break;
        case valorN:
                sentencia;
                break;
        default:
}
```

El valor de la expresión se compara con cada uno de los literales de la sentencia case si coincide alguno se ejecuta el código que le sigue, si ninguno coincide se realiza la sentencia default (opcional), si no hay sentencia default no se ejecuta nada.

La sentencia break realiza la salida de un bloque de código. En el caso de sentencia switch realiza el código y cuando ejecuta break sale de este bloque y sigue con la ejecución del programa. En el caso que varias sentencias case realicen la misma ejecución se pueden agrupar, utilizando una sola sentencia break.

```
switch (expresión){
        case valor1:
        case valor2:
                sentencia;
                break;
        case valor3:
        case valor4:
```

```
        case valor0:
                sentencia;
                break;
        default:
}
```

WHILE: Ejecuta repetidamente el mismo bloque de código hasta que se cumpla una condición de terminación. Hay cuatro partes en cualquier bucle. *Inicialización, cuerpo, iteración* y *terminación*.

```
[inicialización;]
while(condición){
        cuerpo;
        iteración;
}
```

DO-WHILE: Es lo mismo que en el caso anterior pero aquí como mínimo siempre se ejecutara el cuerpo una vez, en el caso anterior es posible que no se ejecute ni una sola vez.

```
[inicialización;]
do{
        cuerpo;
        iteración;
}while(condición);
```

EJEMPLO:Este programa lo que hace es ir sumando n1 mientras que los valores no sean superiores a 100.

```
class fibo{
  public static void main(String args[]){
        int n1=0;
        do{
    n1++;
         System.out.println(n1+" ");
}while(n1<100);
  }
}
```

FOR: Realiza las mismas operaciones que en los casos anteriores pero la sintaxis es una forma compacta.

```
for (inicialización;condición;iteración)
{
        sentencia1;
        sentencia2;
}
```

EJEMPLO: Este programa muestra números del 1 al 100. Utilizando un bucle de tipo FOR.

```
class multi{
        public static void main(String args[]){
                int n1=0;
                for (n1=1;n1<=100;n1++)
                        System.out.print(n1+" ");
        }
}
```

La sentencia continue lo que hace es ignorar las sentencias que tiene el bucle y saltar directamente a la condición para ver si sigue siendo verdadera, si es así sigue dentro del bucle, en caso contrario saldría directamente de el.

8. CLASES Y OBJETOS

El elemento básico de la programación orientada a objetos es una clase. Una clase define la forma y comportamiento de un objeto. Un objeto es similar a una variable salvo que el objeto puede contener varias variables (variables de instancia) y métodos (rutinas que realizan operaciones).

Para la creación de un objeto se necesita el operador new, y que se declaren las variables de instancia dentro de una clase. Mediante una clase se pueden declarar varios objetos que tendrán los mismos atributos.

CREACIÓN DEL OBJETO

```
nomb_clase nomb_objeto=new nomb_clase([valores]);
```

Cuando se hace referencia a un método este debe estar declarado y desarrollado al igual que el objeto. Para declarar y desarrollar un método debe estar dentro de una clase y se debe indicar el valor que devuelve, el nombre y los valores que se le pasan.

CREACIÓN Y DESARROLLO DEL MÉTODO

```
valor devuelto nombre_método([valores])
{
        cuerpo;
}
```

Si tenemos que hacer referencia a las variables de instancia y los métodos contenidos en un objeto se necesita el operador punto(.).

```
Objeto.nomb_método( );
Objeto.nomb_método(valores);
Objeto.variable;
```

EJEMPLO: Teniendo una clase con 2 variables y un método. Desde la otra clase se crea el objeto, se llama al método y se muestran los valores del objeto.

```
class punto{
        int x;
        int y;
        void inicio( ){x=10;y=20;}
}

class principal{
        public static void main(String args[]){
                punto p=new punto( );
                System.out.println (“objeto creado”);
                p.inicio( );
                System.out.println(p.x + “ “+p.y);
        }
}
```

EJEMPLO: En este programa se le pasan valores al método.

```
class punto{
        int x;
        int y;
        void inicio(int a, int b){x=a;y=b;}
}
```

```
class principal{
        public static void main(String args[]){
                int a=10;
                punto p=new punto( );
                System.out.println ("objeto creado");
                p.inicio(a,20);
                System.out.println(p.x + " "+p.y);
        }
}
```

CONSTRUCTORES: Es un método que inicializa un objeto inmediatamente después de su creación. Tienen el mismo nombre de la clase en la que residen. No devuelven ningún tipo de dato ni siquiera void. Al ser automático no necesita ser llamado.

EJEMPLO:

```
        class punto{
                int x , y;
                punto(int a, int b){x=a;y=b;}
        }

        class principal{
                public static void main(String args[]){
                        int a=10, b=20;
                        punto p=new punto(a,b);
                        System.out.println("objeto creado");
                        System.out.println(p.x+" "+p.y);
                }
        }
```

HERENCIA: Relaciona clases una encima de otra de una manera jerárquica. Esto permite que una clase herede todas las variables y métodos de sus ascendientes, además de crear los suyos propios. Para heredar una clase hay que utilizar la palabra clave extends.

```
class nombre_clase extends nombre_superclase{
        Cuerpo;
}
```

EJEMPLO: Partiendo de los ejemplos anteriores vamos a crear una jerarquía de clases.

```
class punto{
        int x , y;
        punto(int a, int b){x=a;y=b;}
}

class punto2 extends punto{
        int z;
        punto2(int a, int b, int c)
        {
                super(a,b);
                z=c;
        }
```

```
}

class principal{
        public static void main(String args[]){
                int a=10, b=20,c=30;
                punto2 p=new punto2(a,b,c);
                System.out.println("objeto creado");
                System.out.println(p.x+" "+p.y+" "+p.z);
        }
}
```

La sentencia super lo que hace es llamar al constructor de la superclase (clase de la que heredamos). En la función principal es donde se crea el objeto que llama al miembro constructor, al cual hay que pasar 3 valores (uno para la clase punto2 y dos para la clase punto).

SOBRECARGA: Se utiliza para crear más de un método con el mismo nombre, pero con distinto numero de parámetros y operaciones distintas. La sobrecarga se puede utilizar para miembros constructores o para la herencia.

EJEMPLO:

```
class punto{
        int x,y;
        punto(int a, int b){x=a; y=b;}
        punto(){x=-1;y=-1;}
}

class principal{
        public static void main(String args[]){
                punto p=new punto();
                punto p2=new punto(10,20);
                System.out.println(p.x+" "+p.y);
                System.out.println(p2.x+" "+p2.y);
        }
}
```

STATIC: El uso de este modificador nos permite utilizar el método y las variables sin necesidad de crear objetos. Los métodos estáticos solo pueden llamar a métodos estáticos. Hay que ser consciente que es equivalente a declararlas como globales.

EJEMPLO:

```
class estatica{
        static int a=3;
        static int b;
        static void metodo(int x)
        {
                System.out.println("x= " +x);
                System.out.println("a= " +a);
                System.out.println("b= " +b);
        }

        public static void main(String args[])
        {
                metodo(42);
        }
```

}

Si la llamada al método static esta en otra clase distinta, se tiene que poner el nombre de la clase en donde esta declarado el método static y el operador punto (.).

```
nomb_clase.nomb_metodo([parámetros]);
```

ABSTRACT: Se utiliza cuando en una clase se defina un método sin proporcionar una implementación completa de este método. Con ello conseguimos que en distintas clases tengamos métodos con el mismo nombre y desarrollar el método en una de las clases.

EJEMPLO:

```
abstract class abstracta{
    abstract funcion();
}

class subclase extends abstracta{
    int x;
    void funcion2(){System.out.println("Función Dos");}
    void funcion() {System.out.println("Func.Abstracta");}
}

class inicial{
    public static void main(String args[]){
      subclase var=new subclase();
    // abstracta var=new subclase();igual que la anterior.
      var.funcion();
    }
}
```

PROTECCIÓN DE ACCESO: Estos modificadores nos indican en que partes de un programa podemos utilizar los métodos y variables. Tenemos tres:

private: Solo tendrán acceso los de la misma clase donde estén definidos.
public: Se pude hacer referencia desde cualquier parte del programa.
protected: Se puede hacer referencia desde la misma clase y las subclases.

SENTENCIA FINAL: Se utiliza para definir valores y que estos no puedan ser modificados desde ningún lugar de nuestro programa. Esto implica que las referencias futuras a este elemento se basarán en la definición.

```
final tipo identificador = valor;
```

EJEMPLO:

```
final int NUEVO_ARCHIVO=1;
final int ABRIR_ARCHIVO=2;
final int GUARDAR_ARCHIVO=3;
final int CERRAR_ARCHIVO=4;
```

9. PAQUETES E INTERFACES

Los paquetes son recipientes con clases que se utilizan para mantener el espacio de clase dividido en compartimentos. Por ejemplo un paquete puede tener una clase llamada lista, que se puede almacenar en un paquete propio sin preocuparte por conflictos con otra clase llamada lista escrita por otra persona. El paquete tendrá que estar almacenado en un directorio con el mismo nombre que la clase creada.

package java.awt.imagen;

La sentencia import se utiliza para utilizar las clases o paquetes enteros sin tener que escribir esas clases. También se puede utilizar el asterisco (*) para buscar en el paquete completo sin tener que especificar cada una de las clases.

import paquete.paquete2.clase;
import paquete.paquete2.*;

Con Java vienen implementadas varios paquetes para realizar operaciones específicas, como operaciones matemáticas, gráficos, imágenes, etc ...

Todas las clases incorporadas en Java están almacenadas en directorios independientes. La clase **java.lang** es la única que se importa automáticamente. El resto, en caso de que sean necesarias las tendremos que importar.

10. GESTIÓN DE CADENAS

La manipulación de cadenas tiene una clase incorporada en el paquete java.lang. Esta clase llama a la clase String que es la representación de una matriz de caracteres que no se puede modificar. Existen otros métodos para crear y modificar cadenas, es el método StringBuffer.

CREACIÓN DE CADENAS:

String nombre="cadena";

LONGITUD DE CADENAS:

Objeto_cadena.length(); // devuelve un entero con la longitud.

CONCATENAR CADENAS:

Mediante el operador +.
Objeto_cadena1+Objeto_cadena2

EXTRACCIÓN DE CARACTERES:

Un solo carácter: Objeto_cadena.charAt(índice);
Varias letras: Obj_cad.getChars(inicio,fin,cadena,0);

COMPARAR CADENAS:

EQUALS: Devuelve true si las 2 cadenas son iguales. Hace distinción entre minúsculas y mayúsculas. Para ignorar esta diferencia se utiliza la segunda sentencia.

```
Objeto_cadena1.equals(Objeto_cadena2);
Objeto_cadena1.equalsIgnoreCase(Objeto_cadena2);
```

COMPARETO: Compara 2 cadenas indicando si son iguales, si la primera es menor a la segunda o si la primera es mayor a la segunda. Devuelve un valor numérico.

VALOR	***DESCRIPCIÓN***
0	Cadenas iguales.
< 0	Cadena1 menor a Cadena2.
> 0	Cadena1 mayor a Cadena2.

EJEMPLO:

```
class cadena{
	public static void main(String args[]){
		String cadena="Esto es una cadena";
		String cad1="1234";
		String cad2="1334";
		boolean res;
		int valor;

		char auxiliar[]=new char[20];

		System.out.println("Long: " +cadena.length());
		System.out.println("Concateno: "+cad1+cad2);
		System.out.println(cad1.charAt(1));
		cadena.getChars(0,5,auxiliar,0);
		System.out.println(auxiliar);

		res=cad1.equals(cad2);
		if(res==true)
			System.out.println("Son iguales");
		else
			System.out.println("Son distintas");

		valor=cad1.compareTo(cad2);
		if(valor==0)
			System.out.println("Son iguales");
		else
			if (valor<0)
				System.out.println("cad1 menor cad2");
			else
				System.out.println("cad1 mayor cad2");
	}
}
```

BUSQUEDAS: Método mediante el cual se puede buscar caracteres o cadenas, devuelve el índice donde se encuentra la primera o última coincidencia. Si la búsqueda no tiene éxito devuelve –1.

Objeto_cadena.indexOf('carácter');
Objeto_cadena.indexOf("cadena");

Objeto_cadena.lastIndexOf('carácter');
Objeto_cadena.lastIndexOf("cadena");

REMPLAZAR: Recibe 2 parámetros. Letra que reemplaza y la letra a reemplazar.

Obj_cad.replace('letra reemplaza','letr a reemplazar');

ELIMINAR ESPACIOS: Quita los espacios iniciales y finales de una cadena.

Objeto_cadena.trim();

CONVERSION: De minúsculas a mayúsculas y viceversa.

Objeto_cadena.toLowerCase(); // Lo convierte a minúsculas.
Objeto_cadena.toUpperCase(); // Lo convierte a mayúsculas.

EJEMPLO:

```
class cade{
	public static void main(String args[]){
		String cad="Ahora es otra cadena distinta";
		String cad1="    Ahora    ";
		String aux;

		System.out.println("Primer en: "+cad.indexOf('e'));
		System.out.println("Ultima: "+cad.lastIndexOf('e'));
		System.out.println("otra en = "+cad.indexOf("otra"));

		System.out.println("Reemplaz cadena por padena");
		aux="cadena".replace('c','p');
		System.out.println(aux);

		System.out.println(cad1.trim());

		System.out.println(cad1.toUpperCase());
		System.out.println(cad1.toLowerCase());
	}
}
```

Utilizando el método StringBuffer que también representa una secuencia de caracteres que se pueden ampliar y modificar directamente. Admite las mismas sentencias que String y añade nuevas sentencias.

CREACIÓN:

StringBuffer nombre=new StringBuffer("cadena");

CAPACIDAD: Reserva 16 espacios aunque ya tengamos algo escrito. Con la segunda línea lo que haces es indicar el numero de espacios que vas a reservar.

Objeto_cadena.capacity();
Objeto_cadena.ensureCapacity(nº);

MODIFICAR: Permite modificar un carácter en una determinada posición.

Objeto_cadena.SetcharAt(posición,nuevo carácter);

INSERTAR: Añade en una determinada posición un carácter o una cadena.

Objeto_cadena.insert(posicion,'carácter');
Objeto_cadena.insert(posicion,"cadena");

ESPACIO SI/NO: Para saber si el contenido de una variable es un espacio. Devuelve true en caso de ser un espacio.

Character.IsSpace(objeto_carácter);

NUMERO O CARÁCTER: Para saber si el contenido de una variable es un número o un carácter. Devuelve true cuando es un número.

Character.isDigit(variable_char);

11. FUNCIONES MATIMÁTICAS Y NUMERICAS

La clase **Math** contiene todas las funciones en coma flotante que se utilizan en geometría y trigonometría. Todas las funciones tienen que ir precedidas del objeto **Math** y deben llevar como cabecera el paquete java.util.*.

SENO: Devuelve un double, que es el seno de un ángulo(expresado en radianes).

Math.sin(variable_double);

COSENO: Devuelve un double, que es el coseno de un ángulo(expresado en radianes).

Math.cos(variable_double);

TANGENTE: Devuelve double, es la tangente de un ángulo(expresado en radianes).

Math.tan(variable_double);

POTENCIA: Devuelve el primer numero elevado al segundo. Deben ser double.

Math.pow(double y,double x);

RAÍZ CUADRADA: Devuelve un valor double, que es la raíz cuadrada del parámetro.

Math.sqrt(variable_double);

LOGARITMO: Devuelve el logaritmo natural del parámetro. Debe ser de tipo double.

Math.log(variable_double);

VALOR ABSOLUTO: Devuelve el valor positivo del parámetro. Cualquier tipo.

Math.abs(variable);

MAXIMO: Devuelve el valor máximo de 2 valores. Puede ser de cualquier tipo.

Math.max(variable1,variable2);

MINIMO: Devuelve el valor mínimo de 2 valores. Puede ser de cualquier tipo.

Math.min(variable1,variable2);

REDONDEO: Redondea un valor en coma flotante. Según el parámetro que se le pasa devuelve un tipo u otro distinto.

Math.round(variable_float); //Devuelve un entero.
Math.round(variable_double); // Devuelve un long.

NUMERO ALEATORIO:

Variable_int=(int)(Math.random()*intervalo);

EJEMPLO:

```
import java.util.*;

class numero{
	public static void main(String args[]){
		double numero;
		double numero1;
		double numero2;

		numero=2;
		numero1=3;
		System.out.println(Math.pow(numero,numero1));

		numero=9;
		numero2=Math.sqrt(numero);
		System.out.println("Raiz de 9: "+numero2);

		System.out.println("Absoluto: "+Math.abs(-25));
		System.out.println(Math.max(numero,numero1));

		numero=20.7;
		System.out.println(Math.round(numero));
	}
}
```

EJEMPLO: Juego de la primitiva.

```
import java.util.*;

class primi{
    public static void main(String args[]){
        int ind1=0,ind2=0;
        int numero[]=new int[6];
                int temporal=0;

                numero[0]=(int)(Math.random()*48)+1;
        while(ind1<6)
                {
          temporal=(int)(Math.random()*48)+1;
                for(ind2=0;ind2<ind1;ind2++)
          {
              if(temporal==numero[ind2])
                              {
                  ind1--;
                              break;
                              }
              numero[ind1]=temporal;
          }
          ind1++;
        }
                for(ind1=0;ind1<=5;ind1++)
                        System.out.println(numero[ind1]);
    }
}
```

12. FUNCIONES DE FECHA Y HORA

Mediante la clase Date podemos representar una fecha y hora, tenemos 3 constructores para inicializar el objeto, que debe ser del tipo Date. Para trabajar con fechas y horas se necesita el paquete java.util.Date.

CONSTRUCTORES (Objetos de la clase Date):

```
Objeto=new Date();
Objeto=new Date(año,mes,día);
Objeto=new Date(año,mes,día,hora,minutos,segundos);
```

La primera línea nos muestra la fecha y hora actual. La segunda muestra la fecha que le indiquemos y la tercera muestra fecha y hora que le indiquemos. En el último constructor se admite también el formato, (“nombre_mes día año_4digitos hora:minutos PM/AM”).

OBTENER AÑO:

```
var_int=Objeto_date.getYear();
```

OBTENER MES:

var_int=Objeto_date.getMonth();

OBTENER DIA:

var_int=Objeto_date.getDay();

OBNETER HORA:

var_int=Objeto_date.getHours();

OBTENER MINUTOS:

var_int=Objeto_date.getMinutes();

OBTENER SEGUNDOS:

var_int=Objeto_date.getSeconds();

MODIFICAR FORMATO:

var_String=Objeto_date.toLocaleString();
El formato es **día/mes/año hora:minutos:segundos**

COMPARAR FECHAS:

new Date(año,mes,día).before(new Date(año,mes,día));

new Date(año,mes,día).after(new Date(año,mes,día));

new Date(año,mes,día).equals(new Date(año,mes,día));

En el primer caso devuelve true si la segunda fecha es menor a la primera.
En el segundo caso devuelve true si la segunda fecha es mayor a la primera.
En el tercer caso devuelve true si las fechas son iguales.

EJEMPLO:

```
import java.util.Date;

class creafecha{
```

```
public static void main( String args[]){
        int h,m,s;
        Date d4=new Date();

        h=d4.getHours();
        m=d4.getMinutes();
        s=d4.getSeconds();
        System.out.println(h+":"+m+":"+s);

        String cadena=d4.toLocaleString();
        System.out.println(cadena);

        Date d2=new Date(71,7,1);
        System.out.println("Fecha segunda: "+d2);

        Date d3=new Date("April 3 1993 3:24 PM");
        System.out.println("Fecha tercera: "+ d3);
    }
}
```

13. UTILIDADES

DICTIONARY

Dispositivo para almacenar valores a través de claves. La clave se utiliza para recuperar los valores más adelante. Mediante una tabla hash podemos añadir, borrar, etc.… Se necesitan los paquetes:

```
java.util.Dictionary;
Java.util.Hashtable;
```

CONSTRUCCIÓN:

```
Hashtable nombre_objeto=new Hashtable();
```

AÑADIR:

```
Objeto.put(“clave”,”valor”);
```

OBTENER VALOR:

```
Objeto.get(“clave”);
```

TAMAÑO DE LA TABLA:

```
Objeto.size();
```

BORRAR ELEMENTO:

```
Objeto.remove(“clave”);
```

ELEMENTOS SI/NO: Nos indica si la tabla tiene algún elemento. True si esta vacía.

```
Objeto.isEmpty();
```

EJEMPLO:

```
import java.util.Dictionary;
import java.util.Hashtable;

class utilidad{
	public static void main(String args[]){
		Hashtable ht=new Hashtable();
		ht.put("titulo","ms-dos");
		ht.put("autor","Ricardo");
		ht.put("editorial","SaraG");
		ht.remove("autor");
		mostrar(ht);
	}

	static void mostrar(Dictionary d){
		System.out.println("Titulo: "+d.get("titulo"));
		System.out.println("Autor: "+d.get("autor"));
		System.out.println("Edit: "+d.get("editorial"));
		System.out.println(d.size());
	}
}
```

VECTOR

Matriz ampliable de referencias a objeto. Los objetos se pueden almacenar al final de un vector o añadirlos en cualquier posición. Es una utilidad muy parecida a la que hemos visto en el punto anterior.

CONSTRUCTOR:

```
Vector nombre_objeto=new Vector();
```

AÑADIR ELEMENTOS:

```
Objeto.addElement("elemento");
```

BUSCAR ELEMENTOS: Devuelve el índice que ocupa el elemento buscado.

```
Objeto.indexOf("elemento");  // el primero que encuentra.
Objeto.lastIndexOf("elemento");  // el último que encuentra.
```

MOSTRAR UN ELEMENTO:

```
Objeto.elementAt(índice); //muestra el que se le indica.
Objeto.firstElement(); //muestra el primero.
Objeto.lastElement(); //muestra el último.
```

EJEMPLO:

```
import java.util.*;

class vectores{
	public static void main(String args[]){
		Vector datos=new Vector();
		datos.addElement("uno");
		datos.addElement("dos");
		datos.addElement("tres");
		System.out.println(datos.elementAt(1));
```

```
			System.out.println(datos.lastElement());
			System.out.println(datos.indexOf("uno"));
		}
}
```

14. EXCEPCIONES

Una excepción es una condición anormal que surge en una secuencia de código durante la ejecución. La gestión de excepciones lleva a la gestión de errores en tiempo de ejecución. Cuando surge una condición excepcional se crea un objeto Exception.

El trabajo con excepciones se realiza mediante las siguientes palabras clave:

try — Tiene el código de ejecución, si se produce un error lanza (throw) una excepción que será capturada.

catch — Captura la excepción lanzada por try que le precede. Puede capturar más de una excepción, es decir que se pueden añadir.

finally — La excepción es tratada en un gestor por omisión.

SINTAXIS DE EXCEPCIONES:

```
try{
	Bloque de código;
}
catch(TipoExcepcion1 e){
	gestión de la excepción;
}
catch(TipoExcepcion2  e){
	gestión de la excepción;
	throw(e);
}
finally{}
```

EJEMPLO: Se produce una excepción y por tanto mostrara la línea de catch, (en pantalla veremos solo el mensaje división por cero. Si todo va bien veríamos el resultado de la división.

```
class excep{
	public static void main(String args[]){
		try{
		  int a=42/0;
		  System.out.println(a);
		}

		catch(ArithmeticException e){
		  System.out.println("división por cero");
		}
	}
}
```

LISTA DE EXCEPCIONES:

ArithmeticException
ArrayIndexOutOfBoundsException
ArrayStoreException
ClassCastException
ClassNotFoundException
CloneNotSupportedException
IllegalAccessException
IllegalArgumentException
IllegalMonitorStateException
IllegalStateException
IllegalThreadStateException
IndexOutOfBoundsException
InstantiationException
InterruptedException
NegativeArraySizeException
NoSuchFieldException
NoSuchMethodException
NullPointerException
NumberFormatException
RuntimeException
SecurityException
StringIndexOutOfBoundsException
⋮

15. HILOS

Un programa multihilo es la programación por el cual se dividen los programas en dos o más procesos que se pueden ejecutar en paralelo. La clase Thread encapsula todo el control necesario sobre los hilos. Con el objeto Thread hay métodos que controlan si el hilo se esta ejecutando, esta durmiendo, en suspenso o detenido.

<u>Los métodos de la clase Thread:</u>

currentThread: Representa al hilo que esta ejecutándose en ese momento.

yield: Asegura que los hilos de menor prioridad no sufran inanición.

sleep: Pone a dormir al hilo en curso durante n milisegundos.

start: Crea un hilo de sistema y ejecuta. Luego llama al método run.

run: Es el cuerpo del hilo. Es llamado por el método start.

stop: Provoca la destrucción del hilo.

suspend: Detiene el hilo, pero no lo destruye. Puede ejecutarse de nuevo.

resume: Para revivir un hilo suspendido.

setName: Asigna un nombre al hilo en curso.

getName: Devuelve el nombre del hilo en ejecución.

setPriority() Establece la prioridad del hilo. De 1 a 10.

getPriority: Devuelve la prioridad del hilo en curso.

EJEMPLO: En el siguiente programa se genera un número aleatorio, que será el retardo de cada hilo. Por tanto según el número generado saldrá primero uno y luego el otro en orden aleatorio.

```
import java.util.*;

class hilos extends Thread{
	int tiempo;
	String nombre;

	public hilos(String nomb,int tempo){
		tiempo=tempo;
		nombre=nomb;
	}

	public void run(){
		try{sleep(tiempo);}
		catch(Exception e){
			System.out.println("Error");
		}
		System.out.println("Hilo: "+nombre+" Temp: "+tiempo);
	}

	public static void main(String args[]){
		hilos h1=new hilos("Uno",(int)(Math.random()*2000));
		hilos h2=new hilos("Dos",(int)(Math.random()*2000));

		h1.start();
		h2.start();
	}
}
```

EJEMPLO: Dos hilos se crean utilizando el método setPriority con dos niveles de prioridad distintos. Se inician ambos y se les permite la ejecución durante 10 segundos. Cada hilo gira en torno a un bucle que cuenta cuantas veces se ejecuto el mismo.

```
class pulso implements Runnable{
	int pulsa=0;
	private Thread hilo;
	private boolean corre=true;

	public pulso(int p){
		hilo=new Thread(this);
		hilo.setPriority(p);
	}

	public void run(){
		while(corre){
			pulsa++;
		}
	}
	public void stop(){
		corre=false;
	}
	public void start(){
		hilo.start();
	}
```

```
}

class hilos{

   public static void main(String args[]){

            Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
            pulso h1=new pulso(Thread.NORM_PRIORITY+2);
            pulso h2=new pulso(Thread.NORM_PRIORITY-2);

            h1.start();
            h2.start();

            try{Thread.sleep(10000);}
            catch(Exception e){}
            h1.stop();
            h2.stop();

            System.out.println(h1.pulsa+" contra "+h2.pulsa);
  }
}
```

16. APPLETS

Las applets son aplicaciones pequeñas a las que se accede a través de un servidor de Internet, se transmiten por la red, se instalan automáticamente y se ejecutan *in situ* como parte de un documento Web.

Todas las applets deben comenzar con dos líneas que importan todos los paquetes de java.awt.* y java.applet.* . Estos dos paquetes son obligatorios, pero existen otros paquetes de clases que nos servirán de utilidad. A parte de los paquetes también debemos heredar obligatoriamente la clase Applet para poder realizar dichos programas.

BIBLIOTECA	***DESCRIPCIÓN***
java.lang	Clases esenciales, números, signos, objetos, compilador, etc...
java.io	Clases que maneja entrada y salida. (Ficheros).
java.util	Clases manejo fechas, hora, string, matemáticas, etc...
java.net	Clases para redes. URL, TCP, UDP, IP, etc...
java.awt	Clases para manejo de interfaces gráfica, ventanas, etc...
java.awt.image	Clases para manejo de imágenes.
java.awt.peer	Clases de conexión gráfica y recursos dependiente de la plataforma.
java.applet	Clases para la creación de applet y recursos.

SINTAXIS GENERAL:

```
import java.awt.*;
```

```
import java.applet.*;

public class extends Applet{
        Cuerpo (metodos);
}
```

INICIALIZACIÓN DE LAS APPLETS

Cuando se escribe un applet se debe sobreescribir métodos de la clase Applet. El problema es, cual hay que sobreescribir y cual no. Para eso hay que conocer el orden de llamada y lo que hacen esos métodos.

A continuación se presenta el orden en que son llamados y una explicación de las operaciones y llamadas que realizan cada uno de los métodos.

init: Es el primero que se llama, es donde debería inicializar sus variables. Solo se le llama una vez, cuando se carga el applet, es decir, al inicio. Llama automáticamente a start.

start: Se llama después de init, mientras que a start se le llama cada vez que se visualiza en pantalla (cuando un usuario abandona una página y vuelve, la applet comienza la ejecución en start).

paint: Se llama cada vez que se necesita pintar el área de dibujo. Se puede utilizar para mostrar mensajes, objetos gráficos e imágenes. Métodos que realizan operaciones muy similares son repaint y redraw. Solo pinta en el applet, no refresca.

update: Se le llama cuando realmente se necesita actualizar la pantalla. La clase paint simplemente pinta el área. Asociado a este método están todas las ordenes de pintar. Es un método más importante de todos. Es donde se realizan casi todas las operaciones para animaciones junto con los hilos.

stop: Se llama cuando un visualizador de red abandona el documento html. Se debe utilizar para detener los hilos y dejarlos en espera. Se le puede llamar o se ejecuta automáticamente cuando se abandona la página.

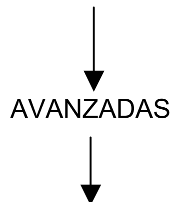
destroy: Cuando el entorno(navegador) determina que es necesario eliminar completamente la applet. En ese momento se liberan todos los recursos.

EJEMPLO:

```
import java.applet.*;
import java.awt.*;

public class hola extends Applet{
        public void paint(Graphics g){
                g.drawString("HOLA MUNDO",20,20);
        }
}
```

Una vez escrito el código fuente, se compila (como siempre) y para ver los resultados debemos insertarlo en un documento Web. Para asociar una aplicación al documento Web hay que seguir los siguientes de pasos (versión Front Page):

Menú INSERTAR



AVANZADAS

SUBPROGRAMA JAVA: Hay que asignar las propiedades de **alto**, **ancho** y **origen del subprograma** (nombre.class). El resto de propiedades son opcionales.

Existe otra posibilidad para ver los applet sin necesidad de navegador ni editor de páginas web. Si no tenemos editor crearíamos la pagina colocando las etiquetas directamente con un editor de texto normal. Una vez creada la página, compilar como siempre, y para ver el funcionamiento del applet simplemente tendríamos que ejecutar el visor *appletviewer*.

EJEMPLO: Página web con un editor de texto normal. La extensión debe ser htm.

```
<html>
        <body>
                <applet width=”tam” height=”tam” code=”nomb.class”> </applet>
        </body>
</html>
```

EJEMPLO: Utilizar el visor de java para ver el applet.

```
C:\>appletviewer pagina1.htm
```

TEXTO

Mediante el objeto de la clase Font asignaremos tipo de letra, estilo y tamaño. Luego utilizaremos el método setFont para establecer ese tipo de letra. Para mostrar esa cadena utilizaremos otros métodos dentro de paint. En resumen para trabajar con texto, primero le damos las características, luego las establecemos y por último lo mostramos.

<u>CREAR Y ESTABLECER LA FUENTE:</u>

```
Font nombre = new Font(“nombre_font”,estilo,tamaño);
Objeto_gráfico.setFont(objeto_font);

setFont(new Font("nombre_font",estilo,tamaño));
```

<u>MOSTRAR TEXTO:</u>

```
Objeto_gráfico.drawString(“mensaje”,x,y);

Objeto_grafico.drawChars(“mensaje”,pos_char,1,x,y);
```

Cuando mostramos la cadena, la posición donde se muestra se lo indicamos mediante los valores de x e y, que es el extremo izquierdo de la línea base de los caracteres. Los valores que puede tomar el parámetro estilo son:

ESTILO	***DESCRIPCIÓN***
Font.BOLD	Negrita.
Font.ITALIC	Cursiva.
Font.PLAIN	Normal.

EJEMPLO:

```
import java.awt.*;
import java.applet.*;

public class dib_letra extends Applet{
  public void paint(Graphics dibujo){
        dibujo.drawString("HOLA MUNDO",5,20);
        Font L1=new Font("Tahoma",Font.BOLD|Font.ITALIC,24);
        Font L2=new Font("Tahoma",Font.PLAIN,18);
        dibujo.setFont(L1);
        dibujo.drawString("HOLA MUNDO",5,50);
        dibujo.setFont(L2);
        dibujo.drawString("HOLA MUNDO",5,100);
  }
}
```

GRAFICOS SÍMPLES

Las coordenadas van de izquierda a derecha y de arriba abajo, es decir x_1, x_2 representan el ancho del objeto. y_1,y_2 representan la altura. Todos los métodos van precedidos del objeto de la clase Graphics que recibe la función paint.

drawRect(x_1,y_1,x_2,y_2);
fillRect(x_1,y_1,x_2,y_2);

drawRoundRect(x,y,ancho,alto,ancho_arco,alto_arco);
fillRoundRect(x,y,ancho,alto,ancho_arco,alto_arco);

drawOval(x_1,y_1,x_2,y_2);
fillOval(x_1,y_1,x_2,y_2);

drawLine(x_1,y_1,x_2,y_2); (*)

drawArc(x,y,anchura,altura,áng_comienzo,áng_final);
fillArc(x,y,anchura,altura,áng_comienzo,áng_final);

drawPolygon(array x, array y,nº lados);
fillPolygon(array x, array y,nº lados);

(*) La línea tiene un estilo y ancho fijo. Se dibuja desde las 2 primeras coordenadas hasta las 2 últimas coordenadas.

EJEMPLO: La applet debe tener 300 de alto y 400 de ancho.

```
import java.awt.*;
import java.applet.*;

public class dibujo extends Applet{
        public void paint(Graphics dibujo){
                int x[]={300,350,300,350,300};
                int y[]={30,30,5,5,30};

                dibujo.drawRect(10,30,30,40);
                dibujo.fillRect(10,10,15,15);
                dibujo.drawRoundRect(10,90,30,40,20,10);
                dibujo.fillRoundRect(50,90,70,40,20,10);
                dibujo.drawOval(100,10,120,30);
                dibujo.fillOval(130,50,150,150);
                dibujo.drawLine(5,3,40,3);
                dibujo.drawArc(5,200,100,80,0,90);
                dibujo.fillArc(5,200,40,60,45,180);

                dibujo.drawPolygon(x,y,5);
        }
}
```

COLORES

Se trabaja mediante el objeto Color. Puede asignarse a un objeto, al fondo o al texto de la applet o a los controles que tenga el applet. Este objeto trata independientemente el color sin tener en cuenta la capacidad del dispositivo, por tanto utilizara el color que mejor se adapte. Siempre se puede utilizar las variables estáticas de color.

```
Objeto_grafico.setColor(Color.variable_color);

setBackground(Color.variable_color);
```

Color.	black
	white
	red
	green
	blue
	yellow
	cyan
	orange
	gray
	darkGray

	lightGray

También se puede crear colores personalizados mediante el mismo objeto. Utilizando la escala RGB. Para crearlo asignamos valores al rojo, verde y azul. El rango para cada uno de los colores es de 0 a 255.

```
Color nomb_color=new Color(int rojo, int verde, int azul);

objeto_grafico.setColor(nomb_color);
```

EJEMPLO: Ejemplo de texto y color.

```
import java.awt.*;
import java.applet.*;

public class dib_letra extends Applet{
   public void init(){
        setBackground(Color.white);
        }

        public void paint(Graphics dibujo){
        dibujo.setColor(Color.blue);
        Font L1=new Font("Tahoma",Font.ITALIC,24);
        Font L2=new Font("Tahoma",Font.PLAIN,18);
        dibujo.setFont(L1);
        dibujo.drawString("HOLA MUNDO",5,50);
     dibujo.setColor(Color.yellow);
     dibujo.setFont(L2);
     dibujo.drawString("HOLA MUNDO",5,100);
        }
}
```

PINTAR EN EL APPLET

Se utiliza el método repaint, que obliga a volver a pintar la applet. A su vez repaint llama a update, con lo cual se actualiza la applet. Tenemos cuatro maneras de utilizar el método repaint.

repaint(); llama al método repaint.

repaint(tiempo); cada x tiempo se actualiza. Mantiene ritmo en las animaciones.

repaint(x_1,y_1,x_2,y_2); actualiza únicamente el área que delimitamos.

repaint(tiempo,x_1,y_1,x_2,y_2); es una mezcla de las 2 anteriores.

EJEMPLO: Texto danzando a lo loco.

```
import java.awt.*;
import java.applet.*;

public class text extends Applet implements Runnable {
   Thread parar;
        int x_coord = 0, y_coord = 0;

   public void start() {
      parar = new Thread(this);
      parar.start();
   }

        public void run() {
               for(;;){
                  try {Thread.sleep(100);}
                        catch (InterruptedException e){}
               repaint();
               }
   }

        public void paint(Graphics g) {
               x_coord = (int) (Math.random()*10+15*10);
      y_coord = (int) (Math.random()*10+36);
         g.drawString("H O L A",x_coord,y_coord);
   }
    }
```

EJEMPLO: Letras danzando.

```
      import java.awt.*;
      import java.applet.*;

      public class text extends Applet implements Runnable {

   char separado[];
   String s = "H o l a   B u e n a s";
   Thread parar = null;
   int i;
   int x_coord = 0, y_coord = 0;

      public void init() {
   separado =  new char [s.length()];
   s.getChars(0,s.length(),separado,0);
   resize((s.length()+1)*15, 50);
   setFont(new Font("TimesRoman",Font.BOLD,36));
      }

      public void start() {
        parar = new Thread(this);
   parar.start();
      }

      public void run() {
        for(;;)
        {
               try {Thread.sleep(100);}
               catch (InterruptedException e){}
      repaint();
        }
      }

      public void paint(Graphics g) {
```

(sigue)

```
    for(i=0;i<s.length();i++)
    {
       x_coord = (int) (Math.random()*10+15*i);
       y_coord = (int) (Math.random()*10+36);
          g.drawChars(separado, i,1,x_coord,y_coord);
    }
       }
} //fin de la clase.
```

EJEMPLO: Creación de un reloj.

```
import java.awt.*;
import java.applet.*;
import java.util.Date;

public class re extends Applet implements Runnable {
  Thread tiempo;

  public void paint(Graphics g) {
    String today;
    Date dat = new Date();
    Font F = new Font("Arial", Font.PLAIN, 24);
    g.setFont(F);
    today = dat.toLocaleString();
    g.drawString(today, 5,20);
  }

  public void start() {
 tiempo = new Thread(this);
     tiempo.start();
  }

  public void run() {
    for(;;)
    {
 repaint();
     pausa(1000);
    }
  }

  public void pausa(int tempo){
        try{Thread.sleep(tempo);}
        catch(InterruptedException e){}
  }
}//fin de la clase
```

TRATAMIENTO DE IMÁGENES

El trabajo con imágenes se realiza mediante la clase Image y un grupo de métodos asociados que los vamos a utilizar para cargar y visualizar. Los formatos deben ser *GIF* o *JPG*. Los métodos asociados a la clase son:

CREAR OBJETO:

```
Image obj_img;
```

LOCALIZAR IMAGEN: Para indicar la ruta donde esta el documento web. Es un parámetro del método getImage.

```
getDocumentBase()
```

CARGAR IMAGEN: Este método debe ir dentro de init para cargar la imagen. Los parámetros que recibe son el lugar donde se encuentra la imagen y el nombre.

```
obj_img=getImage(getDocumentBase(),”nombre_imagen”);
```

MOSTRAR IMAGEN: Insertamos una imagen en una posición determinada del applet.

```
objeto_grafico.drawImage(obj_img,x,y,this);
```

EJEMPLO: Muestra una imagen.

```
import java.awt.*;
import java.applet.*;

public class imagen extends Applet{

	Image img1;

	public void init(){
		img1=getImage(getDocumentBase(),"t2.gif");
	}

	public void paint(Graphics g){
		g.drawImage(img1,20,20,this);
	}
}
```

EJEMPLO: Animación de varias imágenes en la misma posición.

```
import java.awt.*;
import java.applet.*;

public class move extends Applet implements Runnable {
	Image tabla[]=new Image[3];
	Image img_pintar;
	Thread pasos;

	public void init(){
		setBackground(Color.white);
		tabla[0]=getImage(getDocumentBase(),"t3.gif");
		tabla[1]=getImage(getDocumentBase(),"t4.gif");
		tabla[2]=getImage(getDocumentBase(),"t5.gif");
	}

	public void start() {
	    pasos = new Thread(this);
	    pasos.start();
    }

    public void run() {
```

```
            for(;;){

                        //se puede hacer un bucle
                        //para evitar estas líneas
                        //de código.

                        img_pintar = tabla[0];
                        repaint();
                        pausa(500);

                        img_pintar = tabla[1];
                        repaint();
                        pausa(500);

                        img_pintar = tabla[2];
                        repaint();
                        pausa(500);
            }
      }

      public void paint(Graphics g){
            g.drawImage(img_pintar,10,10,this);
      }

      void pausa(int tempo) {
            try {Thread.sleep(tempo);}
          catch (InterruptedException e) {}
      }
}
```

EJEMPLO: Animación de varias imágenes desplazándose de izquierda a derecha.

```
import java.awt.*;
import java.applet.*;

public class move extends Applet implements Runnable {
      Image tabla[]=new Image[3];
      Image img_pintar;
      int x_coord=0,y_coord=0,ir=0;
      boolean vol=false;
      Thread pasos;

      public void init(){
            setBackground(Color.white);
            tabla[0]=getImage(getDocumentBase(),"t3.gif");
            tabla[1]=getImage(getDocumentBase(),"t4.gif");
            tabla[2]=getImage(getDocumentBase(),"t5.gif");
      }

      public void start() {
          pasos = new Thread(this);
          pasos.start();
  }

  public void run() {
            for(;;){
                        int ind=0;
                        for(ind=0;ind<=2;ind++){
                                    img_pintar = tabla[ind];
                                    repaint();
                                    pausa(500);
```

```
                }
            }
        }

        public void paint(Graphics g){

            if(ir<100 && vol==false)
                ir+=10;

            if(ir>=100 || vol==true)
            {
                ir-=10;
                vol=true;
                if(ir<=0)
                    vol=false;
            }
            g.drawImage(img_pintar,ir,10,this);
        }

        void pausa(int tempo) {
            try {Thread.sleep(tempo);}
          catch (InterruptedException e) {}
        }
}
```

CREAR IMÁGENES

Mediante la interfaz ImageProducer crea imágenes a través de sus objetos. Estos objetos proporcionan matrices de enteros o bytes que representan los datos de la imagen (pixeles y color).

El método createImage genera un Image utilizable a través del objeto devuelto por MemoryImageSource objeto creado a partir de los datos que genera ImageProducer.

CREACION DEL OBJETO:

```
MemoryImageSource obj;
obj=new MemoryImageSource(ancho,alto,pixel,0,alinea);

obj_img=createImage(obj);
```

EJEMPLO:

```
import java.awt.*;
import java.applet.*;
import java.awt.image.*;

public class herra extends Applet{

Image img;

public void init(){
    generaImg();
}

public void generaImg(){
    int punto[]=new int[90000];
    int ind=0,x=0,y=0;
    int rojo,verde,azul;
    for(y=0;y<300;y++)
    {
      for(x=0;x<300;x++)
      {
```

```
            rojo=(x);
            verde=(x*2);
                                azul=(x*4);
                                  punto[ind++]=(255<<24)|(r<<16)|(v<<8)|a;
          }
        }
  img=createImage(new MemoryImageSource(300,300,punto,0,300));
        }

    public void paint(Graphics dib){
          dib.drawImage(img,0,0,this);
    }
       }
```

EVENTOS

Cualquier componente puede gestionar sucesos (eventos) sobreescribiendo el método handleEvent de la clase Event. Los eventos pueden ser de ventana, de teclado, de ratón y eventos de los controles como botones, listas, etc.. que se añadan al applet. Otra categoría son aquellos sucesos que ocurren cuando se realizan operaciones con ficheros. En este punto solo veremos los del ratón, del teclado y los comunes. El resto se irán viendo cuando sepamos crear los controles que los producen.

*EVENTOS DEL **RATÓN***

mouseEnter	Cuando el ratón entra en una applet.
mouseExit	Cuando el ratón sale de una applet.
mouseMove	Cuando el ratón se mueve sobre una applet.
mouseDown	Cuando se pulsa el botón izquierdo del ratón.
mouseUp	Cuando se suelta el botón del ratón.
mouseDrag	Cuando se arrastra el ratón sobre un applet.

SINTAXIS DEL MÉTODO:

```
public boolean evento(Event objeto,int x,int y)
{
        cuerpo;
        return true;
}
```

*EVENTOS DE **TECLADO***

keyDown	Cuando se pulsa una tecla.
keyUp	Cuando se suelta una tecla.

SINTAXIS DEL MÉTODO:

```
public boolean evento(Event objeto, int letra)
{
        cuerpo;
        return true;
}
```

*EVENTOS **GENERALES**:*

getFocus	Cuando el control recibe el foco.

lostFocus   Cuando el control pierde el foco.

SINTAXIS DEL MÉTODO:

```
public void evento()
{
        cuerpo;
        return true;
}
```

EJEMPLO: Eventos de teclado. Cuenta la pulsación de teclas.

```
import java.awt.*;
import java.applet.*;

public class tecla extends Applet{
      int contador=0;

      public boolean keyDown(Event evt, int veces){
            contador++;
            repaint();
            return true;
      }

      public void paint(Graphics dib){
            String cad =String.valueOf(contador);
            dib.drawString(cad,20,20);
      }
}
```

EJEMPLO: Hacemos que una imagen siga al ratón.

```
import java.awt.*;
import java.applet.*;

public class movedib extends Applet{
   Image imagen;
   int a;
   int b;

   public void init(){
                imagen=getImage(getDocumentBase(),"t3.gif");
}

   public boolean mouseMove(Event evt, int x, int y){
                a=x;
                b=y;
                repaint();
                return true;
   }

   public void paint(Graphics g){
                g.drawImage(imagen,a,b,this);
   }
}
```

EJEMPLO: Animación de imágenes siguiendo al ratón.

```
import java.awt.*;
import java.applet.*;

public class movedib extends Applet{

	Image imagen[]=new Image[3];
	Image pinta;
	int a=0,b=0;
	Thread pasos;

	public void init(){
		setLayout(null);
		imagen[0]=getImage(getDocumentBase(),"t1.gif");
		imagen[1]=getImage(getDocumentBase(),"t2.gif");
		imagen[2]=getImage(getDocumentBase(),"t3.gif");

	}

	public void run(){
		for(;;)
		{
			int ind=0;
			for(ind=0;ind<=2;ind++)
			{
				pinta=imagen[ind];
				repaint();
				pausa(500);
			}
		}
	}

	public boolean mouseMove(Event evt, int x, int y){
		a=x;
		b=y;
		return true;
	}

	public void paint(Graphics g){
		g.drawImage(pinta,a,b,this);
	}

	public pausa(int tiempo){
		try{Thread.sleep(tiempo);}
		catch(Exception e){}
	}
}
```

17. AÑADIR CONTROLES

La finalidad de añadir controles como botones, cajas de texto, etiquetas, listas, etc. es que el usuario interactue con el entorno y pueda realizar las operaciones que desee. Nosotros

seremos los encargados de añadir esos controles y asignarles las acciones que van a realizar. Muchos de estos controles comparten métodos y eventos.

ETIQUETAS

Se utilizan para dibujar una cadena de texto en una posición y con una alineación (dentro del componente). Realiza la misma función que drawString. Se utiliza la clase Label, lo primero que hacemos es crearla y luego la posicionamos dentro de la applet.

CREACION:

```
Label nombre_objeto=new Label("texto",alineción);
add(nombre_objeto);
```

ALINEACIÓN	***DESCRIPCIÓN***
Label.LEFT	Izquierda.
Label.RIGHT	Derecha.
Label.CENTER	Centrado.

POSICIONAMIENTO:

```
nombre_objeto.reshape(x,y,ancho,alto);
```

CAJAS DE TEXTO

TextField implementa un área de entrada de texto de una sola línea. Todo aquello que escribimos en una caja de texto es de tipo String por tanto si queremos realizar operaciones matemáticas deberemos transformarlo a un valor numérico.

CREACIÓN:

```
TextField nombre_objeto=new TextField(ancho);
add(nombre_objeto);
```

POSICIONAMIENTO:

```
nombre_objeto.reshape(x,y,ancho,alto);
```

GUARDAR TEXTO: valido para etiquetas.

```
variable=nombre_objeto.getText();
```

MOSTRAR TEXTO: valido para etiquetas.

```
nombre_objeto.setText(variable);
```

BOTONES

Son utilizados para ejecutar las acciones. La clase que se utiliza es Button. Una vez creado hay que asociarlo al método action para realizar la operación. El texto que aparece en los botones siempre va en el centro.

CREACIÓN:

```
Button nombre_objeto=new Button("texto");
add(nombre_objeto);
```

POSICIONAMIENTO:

```
nombre_objeto.reshape(x,y,ancho,alto);
```

ASOCIAR EL CONTROL A LA ACCIÓN

Para asociar los controles como botones, listas, casillas, etc... a un método debemos implementar el método action. En dicho método deberemos escribir las acciones que van a realizar cada uno de los controles que tengamos en el applet, teniendo en cuenta lo siguiente.

El método action recibe dos parámetros, el primero es un objeto de la clase Event que recoge el evento producido en el applet. El segundo es un objeto de la clase Object que recoge cual a sido el control que a producido la acción.

Con estos dos objetos lo primero que debemos escribir en el método action son sentencias if para determinar el tipo de control que a producido la acción (***línea 1)***. Si hay más controles del mismo tipo, deberemos distinguir uno de otro con más sentencias if dentro de cada if que gestiona el tipo de control ***(línea 2)***. Por último, cada uno de estos if deben llevar una sentencia return true para indicar que todo ha ido bien ***(lineal)***. Al final del método se debe incluir la sentencia return false para no hacer nada en caso de que la acción realizada no tenga interés para nuestra aplicación ***(línea 4)***.

```
public boolean action(Event nombre_ev, Object nombre_obj)
{
        (1)if (nombre_ev.target instanceof Button)
          {
         (2) cuerpo con if para cada uno de ese tipo;
         (3) return true;
          }
        (4)return false;
}
```

EJEMPLO: Con una caja de texto, una etiqueta y un botón, vamos a escribir un nombre en la caja de texto y al pulsar el botón pasaremos el texto a la etiqueta poniendo además del contenido de la caja de texto un hola.

```
import java.awt.*;
import java.applet.*;

public class eje extends Applet{
  TextField caja_texto=new TextField(10);
  Label etiqueta=new Label(" ",Label.CENTER);
  String nombre;

  public void init(){
    setLayout(null);
    Button boton=new Button("Cambiar");
    add(caja_texto);
    add(etiqueta);
    add(boton);
    caja_texto.reshape(10,10,50,20);
    etiqueta.reshape(70,10,50,20);
    boton.reshape(10,50,100,30);

  }

  public boolean action(Event evt,Object obj){
    if (evt.target instanceof Button)
    {
        nombre=caja_texto.getText();
        caja_texto.setText("");
        etiqueta.setBackground(Color.blue);
        etiqueta.setText("hola "+nombre);
        return true;
    }
    return false;
  }
}//fin de la clase
```

EJEMPLO: En este programa pedimos grados y al pulsar el botón mostrar dibuja el arco según los grados. Si se pulsa el botón limpia la caja de texto y poner el dibujo completo.

```
import java.awt.*;
import java.applet.*;

public class ejemplo extends Applet{
	TextField grados=new TextField(3);
	int g_final=0;

	public void init(){
		setLayout(null);
		Label eti=new Label("grados",Label.CENTER);
		Button okboton=new Button("Mostrar");
		Button limpia=new Button("Limpiar");
		add(eti);
		add(okboton);
		add(limpia);
		add(grados);
		eti.reshape(0,0,50,50);
		grados.reshape(60,15,90,20);
		okboton.reshape(10,60,50,40);
		limpia.reshape(10,120,50,40);
	}
```

(sigue)

```
        public boolean action(Event ev, Object arg){
                boolean res;
                String texto="";

                if(ev.target instanceof Button)
                {
                        res=arg.equals("Mostrar");
                        if(res==true)
                        {
                           g_final=Integer.parseInt(grados.getText());
                           redraw();
                        }
                        else
                        {
                          grados.setText(texto);
                          g_final=0;
                          redraw();
                        }
                        return true;
                }
                return false;
        }

        public void paint(Graphics g){
                g.drawArc(150,150,200,200,0,g_final);
                g.setColor(Color.blue);
                g.drawLine(125,250,375,250);
                g.drawLine(250,125,250,375);
        }

        public void redraw(){
                repaint();
        }
}
```

CAJAS DE TEXTO MULTILINEA

A veces no es suficiente una entrada de texto de una única línea. TextArea es una caja de texto multilínea, es decir, admite varias líneas. Como valores en la construcción toma un String inicial, seguido del número de columnas y filas que se desean visualizar. Además del método constructor existen otros de gran importancia.

CREACIÓN:

```
TextArea nomb=new TextArea(String,int colum,int fil);
add(nomb);
```

POSICIONAMIENTO:

```
nomb.reshape(x,y,ancho,alto);
```

INSERTAR TEXTO:

AL FINAL: nomb.append(texto);

POS ALEATORIA: nomb.insertText(texto,posición);

REENPLAZAR: nomb.replaceText(texto,p_ini,p_final);

EJEMPLO:

```
    import java.awt.*;
    import java.applet.*;

    public class eje5 extends Applet{
TextArea caja=new TextArea("hola a todos",50,10);
String nombre;

public void init(){
        setLayout(null);
        Button boton=new Button("Insertar");
        add(caja);
        add(boton);
        caja.reshape(10,10,200,100);
        boton.reshape(220,10,50,30);

}

public boolean action(Event evt,Object obj){
        if (evt.target instanceof Button)
        {
                String cad="hola a todos";
                caja.appendText(cad);
                //caja.insertText("adios",5);
                //caja.replaceText("tarde",0,5);
                return true;
        }
        return false;
}
    }
```

SIMULANDO EL PORTAPAPELES

Simular el portapapeles de windows es una manera rápida y sencilla de intercambiar información entre distintos controles del applet o frame. Va asociado a las cajas de texto multilínea. Debemos utilizar el una variable para contener la información y los métodos que ya están implementados en java para realizar las operaciones con el portapapeles. Los métodos a utilizar se ocupan de seleccionar , copiar, insertar texto, etc... y son los siguientes:

SELECCIONAR TODO EL TEXTO:

```
objeto_TextArea.selectAll();
```

COPIAR TEXTO SELECCIONADO:

```
var_String=objeto_TextArea.getSelectedText();
```

CORTAR TEXTO SELECCIONADO:

```
var_String=objeto_TextArea.getSelectedText();
obj.replaceText(“”,obj.getSelectionStart(),
```

```
obj.getSelectionEnd());
```

SELECCIONAR PARTE DEL TEXTO: Con el ratón directamente.

LIMITES DE LA ZONA SELECCIONADA:

```
var_int=objeto_TextArea.getSelectionStart();

var_int=objeto_TextArea.getSelectionEnd();
```

PEGAR LO COPIADO O CORTADO: Ambas líneas son necesarias, por si hay un texto marcado y pegamos. Así elimina lo seleccionado y deja solo lo que se pega.

```
obj.replaceText("",obj.getSelectionStart(),
                obj.getSelectionEnd());

obj.insertText(var_String,obj.getSelectionStart());
```

EJEMPLO:

```
import java.awt.*;
import java.applet.*;

public class porta extends Applet{

String temp="";
TextArea caja=new TextArea("");
Button copia=new Button("Copiar");
Button corta=new Button("Cortar");
Button pega=new Button("Pegar");

public void init(){

   setLayout(null);
   add(caja);
   add(copia);
   add(corta);
   add(pega);
   caja.reshape(10,10,200,100);
   copia.reshape(10,110,60,40);
   corta.reshape(80,110,60,40);
   pega.reshape(150,110,60,40);
}

public boolean action(Event evt,Object obj){

   if(evt.target instanceof Button)
   {
```

```
    if(obj.equals("Copiar"))
      temp=caja.getSelectedText();

    if(obj.equals("Cortar"))
    {
      temp=caja.getSelectedText();
            caja.replaceText("",caja.getSelectionStart(),
                                          caja.getSelectionEnd());
    }

    if(obj.equals("Pegar"))
    {
          caja.replaceText("",caja.getSelectionStart(),
                                          caja.getSelectionEnd());
          caja.insertText(temp,caja.getSelectionStart());
    }
    return true;
  }
  return false;

 }//cierra el método action
}//cierra la clase
```

CASILLAS DE VERIFICACIÓN

Las casillas de verificación son botones de opciones, los cuales pueden estar varios seleccionados a la vez. El texto esta situado a la derecha. La clase Checkbox se utiliza para crear el control, y los métodos getState y setState para obtener y mostrar el estado, que admite dos valores true si esta seleccionado y false si no esta seleccionado.

CREAR: Tenemos 2 maneras para crearlos. El primero si se quiere establecer un valor inicial, con el segundo se crea sin indicar el estado, (por defecto están sin seleccionar);

```
Checkbox objeto=new Checkbox("texto",null, boolean);
Checkbox objeto=new Checkbox("texto");
add(objeto);
```

POSICIONAR:

```
objeto.reshape(x,y,ancho,alto);
```

ESTADOS:

```
objeto.getState();  Devuelve el estado del control.
objeto.setState()= boolean;  Establece el estado.
```

EJEMPLO:

```
import java.awt.*;
import java.applet.*;

public class chks extends Applet{

    boolean res=false;
    Checkbox chk=new Checkbox("Relleno");

    public void init(){
```

```
            setLayout(null);
            add(chk);
            chk.reshape(0,0,50,50);
      }

      public boolean action(Event ev, Object arg){
            if(ev.target instanceof Checkbox)
            {
                  if(chk.getState()==true)
                    res=true;
                  else
                    res=false;

                  redraw();
                  return true;
            }
            return false;
      }
```
(sigue)
```
      public void paint(Graphics g){
            if(res==true)
              g.fillArc(150,150,200,200,0,360);
            else
              g.drawArc(150,150,200,200,0,360);

            g.setColor(Color.blue);
            g.drawLine(125,250,375,250);
            g.drawLine(250,125,250,375);
      }

      public void redraw(){
            repaint();
      }
}
```

RADIO BOTONES

Son prácticamente igual a las casillas de verificación, salvo que este control solo permite que uno de ellos este seleccionado, también cambia el aspecto. Para crear este tipo de controles primero hacemos un grupo, y luego los controles. Los métodos asociados son los mismos que en las casillas de verificación (setState y getState).

CREACIÓN DEL GRUPO:

```
CheckboxGroup nombre_grupo=new CheckboxGroup();
```

CREACIÓIN DE LOS ELEMENTOS:

```
Checkbox obj=new Checkbox("text",nomb_grupo,boolean);
Checkbox obj1=new Checkbox("text",nom_grupo,boolean);
add(objeto);
add(objeto1);
```

POSICIONAMIENTO:

```
objeto.reshape(x,y,ancho,alto);
```

LISTAS DESPLEGABLES

Mediante la clase Choice crearemos este tipo de listas. Este tipo de lista solo admite una selección, el orden de sus elementos es según han sido añadidos pero se nos permite indicar cual es el valor que esta seleccionado por defecto. Las lista tienen dos eventos cuando se seleccionan elementos de la lista.

CREAR:

```
Choice nombre_objeto=new Choice();
add(nombre_objeto);
```

POSICIONAMIENTO:

```
nombre_objeto.reshape(x,y,ancho,alto);
```

AÑADIR ELEMENTOS:

```
nombre_objeto.addItem(“texto”);
```

VALOR INICIAL:

```
nombre_objeto.Select(“texto”);
```

SELECCIONAR ELEMENTO:

```
variable=objeto.getSelectedItem();
```

ÍNDICE DEL ELEMENTO:

```
variable=objeto.getSelectedIndex();
```

CONTAR ELEMENTOS:

```
variable=objeto.countItems();
```

EVENTOS:

```
LIST_SELECT
LIST_DESELECT
```

LISTAS

Mediante la clase List crearemos el objeto. Este tipo de lista puede admite más de una selección, el orden de sus elementos es según han sido añadidos. Los miembros son los mismos que en el control anterior solo cambia el modo de construirlos. En este control aparecen unas barras de desplazamiento verticales automáticamente.

CREAR:

```
List nombre_objeto=new List(0,true); múltiple selección.
List nombre_objeto=new List(0,false); selección simple.
```

EJEMPLO:

```
import java.awt.*;
import java.applet.*;

public class lista extends Applet{

Label eti1=new Label("",Label.LEFT);
Label eti2=new Label("",Label.LEFT);
Button mira=new Button("Mostrar");
List lista=new List(0,true);

        public void init(){
  setLayout(null);

  add(eti1);
  add(eti2);
  add(mira);
  add(lista);

  eti1.reshape(120,10,40,50);
  eti2.reshape(160,10,40,50);
  mira.reshape(10,90,75,40);
  lista.reshape(10,10,100,75);

  lista.addItem("Uno");
  lista.addItem("Dos");
  lista.addItem("Tres");
  lista.addItem("Cuatro");
  lista.addItem("Cinco");
  lista.addItem("Seis");
        }

public boolean action(Event evento,Object obj)
{
  if (evento.target instanceof Button)
  {
    int num=lista.getSelectedIndex();
        String cad1=lista.getSelectedItem();
    String cad2=String.valueOf(num);
    eti1.setText(cad2);
    eti2.setText(cad1);
    return true;

    // int num1=lista.countItems();
    // String cad2=String.valueOf(num1);
    // eti1.setText(cad2);

  }
  return false;
}
}
```

BARRAS DE DESPLAZAMIENTO

Se utiliza para seleccionar valores continuos entre un mínimo y un máximo especificado. Las barras pueden tener una orientación vertical u horizontal. Los métodos asociados sirven para obtener y establecer la posición de la barra y su valor mínimo y máximo. Para trabajar con las barras se utilizan los eventos que van asociadas a ellas junto con el método handleEvent.

CREACIÓN:

Scrollbar obj=new Scrollbar(orient, pos_ini,tam_cuadro,ini,fin);
add(obj);

CONSTANTE	***DESCRIPCIÓN***
Scrollbar.VERTICAL	Orientación vertical
Scrollbar.HORIZONTAL	Orientación horizontal

POSICIONAMIENTO:

obj.reshape(x,y,ancho,alto);

VALORES MÁXIMO Y MÍNIMO:

obj.getMaximum(valor);

obj.getMinimum(valor);

ESTABLECER VALOR:

obj.setValue(valor);

OBTENER VALOR:

variable=obj.getValue();

EVENTOS:

EVENTOS	***DESCRIPCIÓN***
Event.SCROLL_LINE_UP	Pulsar flecha arriba
Event.SCROLL_LINE_DOWN	Pulsar flecha abajo
Event.SCROLL_PAGE_UP	Pulsar en la barra arriba
Event.SCROLL_PAGE_DOWN	Pulsar en la barra abajo
Event.SCROLL_ABSOLUTE	Arrastrar el descriptor.

SINTAXIS DEL MÉTODO:

```
public boolean handleEvent(Event objeto)
{
        cuerpo;
```

```
        return true;
    }
```

EJEMPLO:

```
import java.awt.*;
import java.applet.*;

public class barra3 extends Applet{
Scrollbar hor=new Scrollbar(Scrollbar.VERTICAL ,5,10,0,10);
Label eti1=new Label("5",Label.LEFT);
int contador=50;

public void init(){
  setLayout(null);
  add(hor);
  add(eti1);
  hor.reshape(10,10,20,100);
  eti1.reshape(30,10,20,30);
}

public boolean handleEvent(Event evt){
  if (evt.target instanceof Scrollbar)
  {
            String cad=String.valueOf(hor.getValue());
     eti1.setText(cad);
     return true;
  }
  return false;
}
}// fin de la clase
```

CANVAS

Además de los componentes estándar, hay un componente (Canvas), que nos permite implementar cualquier otro tipo de control en su interior y capturar sus eventos a través de la canva. Es muy útil cuando el control que implementamos no responde a eventos.

La clase Canvas no responde a los eventos directamente si no que es el programador quien debe definir una subclase de Canvas a la que se envían todos los eventos y, otra clase donde crearemos el objeto de la subclase e implementaremos todo el código del programa.

CREACIÓN DE LA SUBCLASE:

```
public class nomb_subclase extends Canvas{
        cuerpo;
}
```

CREACIÓN DEL OBJETO CANVAS: Debe ser una clase distinta a la anterior.

```
public class nombre extends Applet{
    cuerpo;
    nomb_subclase obj=new nomb_subclase([param]);
    add(“texto“,obj);
}
```

TAMAÑO:

```
obj.resize(ancho,alto);
```

Los métodos relacionados con el diseño de los controles que hemos ido viendo anteriormente también se pueden aplicar al objeto canvas. Como por ejemplo los métodos para establecer color, tipo de letra, etc...

Existen dos clases con una estrecha relación a las canvas. Estas son Dimension que nos permite modificar el ancho y alto de la canva y la clase Insets que nos servirá para colocar un borde o recuadro a la canva.

OBJETO DIMENSION:

```
Dimension objeto=size();
```

MÉTODOS ASOCIADOS:

```
objeto.width(valor);

objeto.heigth(valor);
```

UTILIZACIÓN DE INSETS: Para usar esta clase debemos sobreescribir su método. Este método devuelve un objeto de la clase Insets.

```
public Insets insets()
{
        return new Insets(x1,y1,x2,y2);
}
```

EJEMPLO:

```
import java.awt.*;
import java.applet.*;

class Micanva extends Canvas{

  String texto="";
  public Micanva(){ } //constructor

  public void paint(Graphics dib){
       dib.drawString(texto,25,25);
       dib.drawRect(0,0,90,90);
  }

  public boolean mouseEnter(Event evt, int x,int y){
            texto="Raton dentro";
            repaint();
            return true;
```

```
        }

        public boolean mouseExit(Event evt, int x,int y){
                texto="Raton fuera";
                repaint();
                return true;
        }

    }

    public class can5 extends Applet{

        public void init(){
            Micanva lienzo=new Micanva();
            setLayout(null);
            add(lienzo);
            lienzo.resize(100,100);
        }

        public Insets insets(){
            return new Insets(10,10,10,10);
        }
      }
```

MARCOS, PANELES Y ORGANIZAR (*LAS VENTANAS*)

Frame (marco), es una ventana independiente del applet. Desciende de la clase Window que nos permite crear ventanas sin bordes, menús, etc... y por este motivo se implementa la clase Frame para construir ventanas con bordes, menús, títulos, etc... Debemos tener en cuenta que esa ventana solo se podrá cerrar a través de un control del applet o al cerrar la ventana del navegador.

CREAR OBJETO:

```
Frame obj_frame=new Frame("titulo de ventana");
```

DIMENSIONAR:

```
obj_frame.resize(ancho,alto);
```

MOSTRAR:

```
obj_frame.show();
```

OCULTAR:

```
obj_frame.hide();
```

CERRAR:

```
obj_frame.dispose();
System.exit(0);
```

AÑADIR CONTROLES: Si es un panel o canva se quita la posición, solo va el objeto.

```
obj_frame.add("posición",objeto_control);
```

POSICIÓN	*DESCRIPCIÓN*
North	Parte superior de la ventana.
South	Parte inferior de la ventana.
East	Parte izquierda de la ventana.
West	Parte derecha de la ventana.
Center	Parte central de la ventana.

Panel es una clase derivada de Container la cual nos va a servir para introducir controles en su interior, de la misma forma que una canva. La principal utilidad que se le puede dar es para colocar controles a una ventana en un punto que a nosotros nos interese a través de un objeto Panel.

CREAR OBJETO:

```
Panel objeto_panel=new Panel();
```

TAMAÑO Y POSICIÓN:

```
objeto_panel.resize(ancho,alto);
```

MOVIMIENTO:

```
objeto_panel.move(x,y);
```

AÑADIR ELEMENTO:

```
objeto_panel.add("posicion",objeto_control);

objeto_panel.add(objeto_control);
```

EJEMPLO:

```
import java.awt.*;
import java.applet.*;

public class venta11 extends Applet{
  Frame ventana=new Frame("hola");
  Label eti1=new Label("SARA GARCIA",Label.CENTER);
  Button abrir=new Button("Mostrar");

  public void init(){
    Panel parte=new Panel();
    parte.resize(50,50);
    ventana.add(parte);
    parte.add(eti1);
    add(abrir);
    abrir.reshape(10,10,50,50);
  }

  public boolean action(Event ev,Object obj){
    if(ev.target instanceof Button)
    {
      ventana.resize(100,100);
      ventana.show();
      return true;
    }
```

```
        return false;
    }
}
```

<u>Organizar</u> los componentes hasta ahora, era "*a mano*", mediante paneles o canvas, pero el paquete AWT incluye un gestor de organización que se establece mediante el método setLayout, que a través de varias clases de LayoutManager nos permite organizar los controles que tengamos en el applet, panel, canva, ventana, etc.. Estas clases son:

FlowLayout: Estilo de organización muy similar a un editor de texto. Los componentes se colocan desde la esquina superior izquierda hacia la derecha y cuando no cabe en una línea pasan a la siguiente, dejando espacio entre todos los componentes. Cada línea puede estar alineada a la izquierda, derecha o centro, según la constante de su construcción.

<u>CREACIÓN</u>:

```
SetLayout(new FlowLayout(alinea,dist_hor,dist_ver));
```

<u>AÑADIR CONTROLES</u>:

```
add(objeto_control);
```

BorderLayout: Estilo de organización para ventanas de nivel superior, tiene cuatro componentes, estrechos y fijos en los extremos y una parte central amplia que contrae a los extremos.

<u>CREACIÓN</u>:

```
setLayout(new BorderLayout());
```

<u>AÑADIR ELEMENTOS</u>:

```
add("posicion",objeto_control);
```

POSICIÓN	***DECRIPCIÓN***
North	Parte superior de la ventana.
South	Parte inferior de la ventana.
East	Parte izquierda de la ventana.
West	Parte derecha de la ventana.
Center	Parte central de la ventana.

GridLayout: Crea una cuadrícula uniforme, definida a partir de un número de filas y columnas para cada uno de los controles. Las filas y columnas se le indican en el constructor.

<u>CREACIÓN</u>:

```
setLayout(new GridLayout(filas,columnas);
```

<u>AÑADIR ELEMENTOS</u>:

```
add(objeto_control);
```

MENUS

Cada ventana de nivel superior puede tener una barra de menús. Esta barra esta compuesta por menús popup, los cuales están compuestos de menús ítem. Los menús de tipo ítem también pueden comportarse como popup con lo que podremos anidar submenus creando una jerarquía.

Lo primero es crear la barra de menús mediante la clase MenuBar. Una vez creada, iremos creando los menús popup y cada uno de los ítem que contengan los popup. Y por último tendremos que ir añadiendo los ítem a los popup y los popup a la barra. También debemos añadir la barra de menú a la ventana.

CREAR BARRA DE MENU:

```
MenuBar obj_barra_menu=new MenuBar();
```

CREAR MENU POPUP:

```
Menu obj_menu=new Menu(“texto”);
```

CREAR MENU ITEM:

```
MenuItem obj_item=new MenuItem(“texto”);

CheckMenuItem obj_item=new CheckMenuItem(“text”);
```

AÑADIR MENU ITEM A MENU POPUP:

```
obj_menu.add(obj_item);
```

AÑADIR MENU POPUP A LA BARRA:

```
ojb_barra_menu.add(obj_menu);
```

AÑADIR BARRA A LA VENTANA:

```
objeto_frame.setMenuBar(obj_barra_menu);
```

EJEMPLO: Con este ejemplo solo visualizamos la ventana y el menú.

```
import java.awt.*;
import java.applet.*;

class menus extends Applet{

    public void init(){
        Frame ventana=new Frame("MENUS");
        ventana.resize(200,200);

        MenuBar barra=new MenuBar();
        Menu m_archivo=new Menu("Archivo");
        Menu m_edicion=new Menu("Edicion");
        Menu m_deshace=new Menu("Deshacer");
```

```
        MenuItem abrir=new MenuItem("Abrir");
        MenuItem guardar=new MenuItem("Guardar");
        MenuItem imprime=new MenuItem("Imprimir");
        MenuItem copiar=new MenuItem("Copiar");
        MenuItem cortar=new MenuItem("Cortar");
        MenuItem pegar=new MenuItem("Pegar");
        MenuItem todo=new MenuItem("Total");
        MenuItem parte=new MenuItem("Parte");
```

(*sigue*)

```
            m_archivo.add(abrir);
        m_archivo.add(guardar);
        m_archivo.add(new MenuItem("-"));
        m_archivo.add(imprime);

        m_edicion.add(copiar);
        m_edicion.add(cortar);
        m_edicion.add(pegar);

        m_deshace.add(todo);
        m_deshace.add(parte);

        m_edicion.add(m_deshace);

        barra.add(m_archivo);
        barra.add(m_edicion);

        ventana.setMenuBar(barra);
        ventana.show();
      }
   }
```

Una vez construido el menú y todos sus niveles lo que debemos hacer, es verificar en nuestro manejador de eventos (método action) el ítem que seleccionamos para realizar las acciones asociadas. Esta verificación la realizamos comparando el evento con los ítem que tenemos en nuestro menú.

En este tipo de aplicaciones debemos tener una clase que herede la clase Applet que solo tendrá la creación de la ventana. y otra distinta que herede Frame, la cual tendrá todos los métodos action, paint, etc... Todas las acciones y métodos de la clase que hereda Frame actúan sobre la ventana creada.

EJEMPLO:

```
import java.awt.*;
import java.applet.*;

public class menus extends Applet{
   public void init(){
      ventana v1=new ventana();
   }
}

class ventana extends Frame{
      MenuBar barra=new MenuBar();
      Menu m_colores=new Menu("Colores");
      MenuItem rojo=new MenuItem("Rojo");
      MenuItem azul=new MenuItem("Azul");
      MenuItem salir=new MenuItem("Salir");
```

(*sigue*)

```
    public ventana(){
        m_colores.add(rojo);
        m_colores.add(azul);
        m_colores.add(new MenuItem("-"));
        m_colores.add(salir);
        barra.add(m_colores);
        setMenuBar(barra);
        resize(200,200);
        show();
    }

    public boolean action(Event evt,Object obj){
       if(evt.target==rojo)
       {
          setBackground(Color.red);
          return true;
       }

       if(evt.target==azul)
       {
          setBackground(Color.blue);
          return true;
       }

       if(evt.target==salir)
       {
          dispose();
          return true;
       }
       return false;
    }
}// cierra la clase ventana
```

Otro de los puntos importantes en los menús es la posibilidad de activar y desactivar las opciones que tiene un menú. Ambas acciones se consiguen mediante dos métodos internos, que ya vienen implementados en java.

DESACTIVAR MENU:

```
objeto_item.disable();
```

ACTIVAR MENU:

```
objeto_item.enable();
```

CUADROS DE DIALOGO

Los cuadros de dialogo son ventanas estándar modales, es decir, que debemos cerrarlas para continuar la ejecución del applet. Se puede utilizar un frame como ventana pero con los cuadro de dialogo se ahorra mucho trabajo y tiempo. Se crean a partir de la clase FileDialog, por tanto debemos crear el objeto de esa clase para trabajar con sus métodos.

Los puntos a tener en cuenta de la ventana son los siguientes, cuando cerramos esta ventana, el objeto no se destruye, simplemente oculta la ventana. Al pulsar el botón ABRIR existe una función que guarda el nombre del fichero y la extensión en una cadena y otra que guarda la ruta completa de directorios con la unidad y la barra final de directorio. Si pulsamos el boton CANCELAR ambas funciones devuelven un nulo.

CREAR OBJETO:

FileDialog obj=new FileDialog(obj_frame,"Titulo",estilo);

ESTILO	***DESCRIPCIÓN***
FileDialog.LOAD	Cuadro para Abrir ficheros.
FileDialog.SAVE	Cuadro para guardar ficheros.

MOSTRAR DIALOGO:

obj.show();

GUARDAR NOMBRE FICHERO Y DIRECTORIO:

var_String=obj.getFile();

var_String=obj.getDirectory();

ESTABLECER NOMBRE EN DIALOGO:

obj.setFile(var_String);

EJEMPLO:

```
import java.awt.*;
import java.applet.*;

public class cuadros extends Applet{
    TextArea caja=new TextArea("");
    Button abre=new Button("Abrir");
```

```
        Frame c1=new Frame();
        FileDialog cuadro=new
        FileDialog(c1,"Abrir",FileDialog.LOAD);

    public void init(){
      setLayout(null);
      add(caja);
      add(abre);
      caja.reshape(10,10,200,100);
      abre.reshape(10,110,60,40);
    }

    public boolean action(Event evt,Object obj){
      String fichero=null;
      String directorio=null;

      if(evt.target instanceof Button)
      {
         if(obj.equals("Abrir"))
         {
           cuadro.show();
                   directorio=cuadro.getDirectory();
                   fichero=cuadro.getFile();

           if(fichero!=null)
                           caja.setText(directorio+fichero);
         }
         return true;
      }
      return false;
    }
} //cierra la clase
```

18. FICHEROS

La mayoría de los programas acceden a datos externos y estos datos se recuperan a partir de un origen de entrada y se guardan a través de un destino de salida. Java llama flujo a esta abstracción y la implementa con clases del paquete java.io. Este paquete debe ser incluido obligatoriamente en el código fuente.

El paquete java.io incluye una clase File para trabajar directamente con el sistema de archivos. No especifica como se recupera o almacena la información. Esta clase puede crear los objetos utilizando cualquiera de sus 3 constructores:

```
File objeto=new File("\");
File objeto=new File("\","nombre.extension");
File objeto=new File("ruta y nombre.extension");
```

Existen métodos para determinar las propiedades estándar de un objeto File. Estos métodos no se pueden utilizar para alterar sus valores. Todos pertenecen a la clase File y utilizan el paquete java.io.

MÉTODO	***DESCRIPCIÓN***
objeto_file.getName();	Devuelve el nombre del fichero.
objeto_file.getPath();	Devuelve la ruta del fichero.
objeto_file.exists();	Devuelve true si existe el fichero.
objeto_file.canWrite();	Devuelve true si se puede escribir.
objeto_file.canRead();	Devuelve true si se puede leer.

objeto_file.isDirectory();	Devuelve true si es un directorio.
objeto_file.isFile();	Devuelve true si es un fichero.
objeto_file.lastNodified();	Devuelve la fecha de la última modificación.
objeto_file.length();	Devuelve la longitud del fichero.

También existen métodos que permiten realizar alguna modificación sobre los ficheros, renombrar y borrar. La creación de ficheros es un punto que veremos a continuación en el apartado de los Streams.

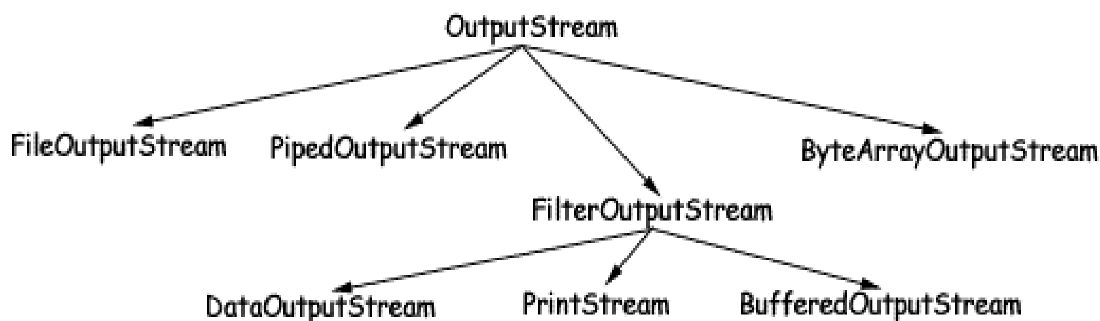
MÉTODO	***DESCRIPCIÓN***
objeto_file.renameTo(nombre);	Renombra el fichero con parámetro.
objeto_file.delete();	Borra el fichero del disco.

EJEMPLO:

```
import java.io.*;

public class fiche{
  public static void main(String args[]){
     File fichero=new File("c:\\java\\texto.txt");
     String cad1="";
        String cad2=fichero.getPath()+" Tam.:"+fichero.length();

     if (fichero.isFile()==true)
        cad1="Tipo: Fichero";
     else
        cad1="Tipo: Directorio";

     System.out.println(cad1);
     System.out.println(cad2);
  }
}
```

STREAMS DE SALIDA (*FILEOUTPUTSTREAM - ESCRITURA*)

Los *Streams* de salida es el modo de trabajo que tiene java de escribir (crear ficheros). Este proceso de escritura es realizado a partir de dos clases, FileOutputStream y DataOutputStream que se encuentran en el paquete java.io. Todas los métodos de escritura se complementan con otras clases que no son las anteriormente nombradas.



La clase FileOutputStream es útil para la escritura de ficheros de texto. Lo primero que debemos hacer es, abrir el fichero (creando objeto), luego escribiremos y por último lo cerraremos. Un inconveniente que tiene es que solo se pueden escribir datos de tipo byte.

APERTURA: Existen dos modos para abrir un fichero.

```
FileOutputStream objeto;
objeto=new FileOutputStream("ruta y nombre.ext");

File obj_file=new File("ruta y nombre.ext");
FileOutputStream objeto;
objeto=new FileOutputStream(obj_file);
```

ESCRITURA: Una vez abierto se escriben bytes utilizando alguna de las 3 posibilidades del método write.

```
objeto.write(int);
objeto.write(byte[]);
objeto.write(byte[],inicio,long_en_bytes);
```

Escribe un byte.
Escribe un array de bytes.
Escribe n bytes desde inicio hasta la longitud dada.

CERRAR: Se puede utilizar el método close o dejar que el sistema lo cierre cuando se destruya el objeto del fichero.

```
objeto.close();
```

La clase **DataOutputStream** puede escribir datos binarios, es decir, cualquier tipo. Es muy similar a la utilizada anteriormente. Necesitaremos un objeto de la clase FileOutputStream para enlazarlo como un *Stream* de salida y con un objeto de la clase BufferedOutputStream para crear el buffer de datos.

APERTURA:

```
DataOutputStream obj_data;
BufferOutputStream obj_buffer;
FileOutputStream obj_file;

obj_file=new FileOutputStream("ruta y nombre.ext");
obj_buffer=new BufferedOutputStream(obj_file);
obj_data=new DataOutputStream(obj_buffer);
```

ESCRITURA:

obj_data.write*tipo*(var_tipo);

tipo	***DESCRIPCIÓN***
writeBoolean	variable de tipo booleano

writeByte	variable de tipo byte
writeShort	variable de tipo short
writeChar	variable de tipo char
writeInt	variable de tipo int
writeFloat	variable de tipo float
writeDouble	variable de tipo double
writeBytes	array de bytes
writeChars	array de chars

CERRAR:

```
obj_file.close();
obj_buffer.close();
obj_data.close();
```

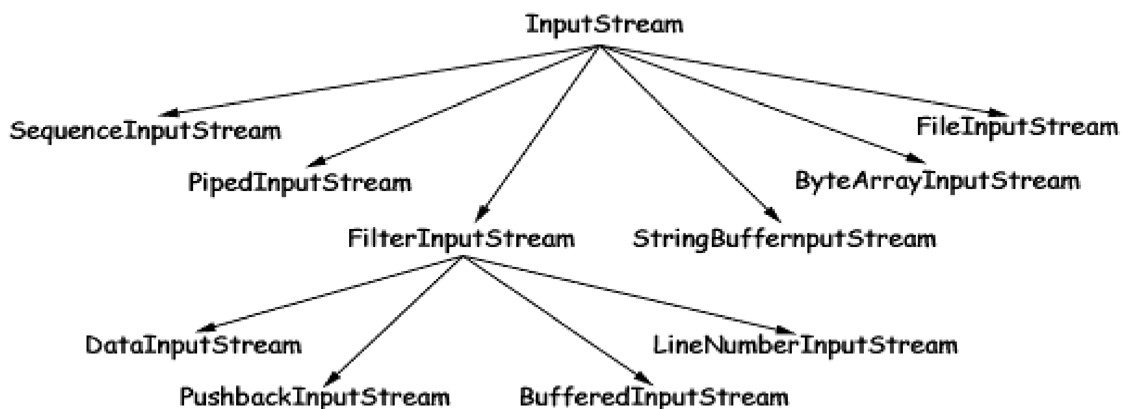
EJEMPLO: Métodos para abrir un fichero. Utilizamos cuadros de dialogo para elegir el fichero.

```
import java.io.*;
import java.awt.*;
import java.applet.*;

public class abre extends Applet{
   TextArea editor=new TextArea("",50,70);
   Button abre=new Button("Abrir");
   Frame c1=new Frame();

   public void init(){
      setLayout(new BorderLayout());
      add("Center",editor);
      add("South",abre);
      editor.reshape(10,10,300,200);
      abre.reshape(40,240,50,50);
   }

   public boolean action (Event evt, Object obj){
     String ruta="";

     if(evt.target instanceof Button)
     {
       FileDialog opn;
       opn=new FileDialog(c1,"Abrir",FileDialog.LOAD);
       opn.show();
       ruta=opn.getDirectory()+opn.getFile();
       abre_fichero(ruta); //llamamos a nuestra función
       return true;
```

```
    }
    return false;
  }

  public void abre_fichero(String ruta){
    String linea=null;
    try{
      FileInputStream canal;
      DataInputStream fichero;

      canal=new FileInputStream(ruta);
      fichero=new DataInputStream(canal);

      editor.setText("");
      do{
        linea=fichero.readLine();
        editor.appendText(linea+"\n");
      }while(linea!=null);

      canal.close();
      fichero.close();
    }
    catch(IOException e){}
  }
} //cierra la clase
```

STREAMS DE ENTRADA (*FILEINPUTSTREAM – LECTURA*)

La comparativa necesaria de la escritura de datos es la lectura. Este proceso se realiza a través de la clase FileInputStream que representa el fichero a leer. El modo en que se puede leer el fichero es completo, parcial (varios bytes) o acceder a un byte determinado. El inconveniente de utilizar solo esta clase es que, solo puede leer datos de tipo byte.



<u>*APERTURA*</u>: Dos modos de realizar la apertura.

```
FileInputStream obj;
obj=new FileInputStream("ruta y nombre.ext");

File obj_file=new File("ruta y nombre.ext");
FileInputStream obj;
obj=new FileInputStream(obj_file);
```

<u>*LECTURA*</u>: Una vez abierto el fichero se puede utilizar cualquiera de las 3 opciones del método read.

```
var_int=obj.read();
var_int=obj.read(byte[]);
```

var_int=obj.read(byte [], inicio,longitud);

Devuelve el byte leído o –1 si ha llegado al final del fichero.
Rellena el array con lo leído y devuelve el total de bytes o –1 si es final de fichero.
Llena el array desde inicio hasta longitud. Da los bytes leídos o –1 si es final de fichero.

CERRAR: Se puede utilizar el método close o dejar que el sistema lo cierre cuando se destruya el objeto del fichero.

objeto.close();

La clase **DataInputStream** se comporta de una manera muy parecida a FileInputStream. Pero los datos que lee pueden ser de cualquier tipo. Se suele utilizar con ficheros binarios y utiliza la clase FileInputStream como apoyo en las operaciones de lectura.

APERTURA:

DataInputStream obj_data;
FileInputStream obj_file;

obj_file=new FileInputStream("ruta y nombre.ext");
obj_data=new DataInputStream(obj_file);

LECTURA:

var tipo=obj_data.read*tipo*();

tipo	***DESCRIPCIÓN***
readByte	Lee tipo byte.
readShort	Lee tipo short.
readChar	Lee tipo char.
readInt	Lee tipo int.
readLong	Lee tipo long.
readFloat	Lee tipo float.
readDouble	Lee tipo double.
readLine	Lee Una cadena (líneas).

CERRAR:

obj_file.close();
obj_data.close();

EJEMPLO: Métodos para guardar ficheros. Utilizamos cuadros de dialogo para elegir el fichero.

```
import java.io.*;
import java.awt.*;
import java.applet.*;

public class salva extends Applet{

  TextArea editor=new TextArea("",50,70);
  Button guarda=new Button("Guardar");
  Frame c1=new Frame();

  public void init(){
      setLayout(new BorderLayout());
      add("Center",editor);
      add("South",guarda);
      editor.reshape(10,40,300,200);
      guarda.reshape(240,240,50,50);
  }

  public boolean action (Event evt, Object obj){
    String ruta=null;
    if(evt.target instanceof Button)
    {
      FileOutputStream canal;
      DataOutputStream fichero;

      FileDialog sal;
      sal=new FileDialog(c1,"Guardar",FileDialog.SAVE);
      sal.setFile(ruta);
      sal.show();
      ruta=sal.getFile();

      try{
        canal=new FileOutputStream(ruta);
        fichero=new DataOutputStream(canal);

                fichero.writeBytes(editor.getText());

        canal.close();
        fichero.close();
      }
      catch(IOException e){}
      return true;
    }
    return false;
  }
} // cierra la clase
```

19. SOCKETS

Mecanismos de comunicación entre programas a través de una red *TCP/IP*. Estos realizan la interfaz entre aplicación y protocolo. Dichos mecanismos pueden tener lugar dentro de la misma máquina o a través de una red, en forma cliente-servidor. Java proporciona para esto las clases ServerSocket, Socket,InetAddress, etc...

Para establecer la conexión a través de un *socket*, tenemos que programar por un lado el servidor y por otro los clientes. En el servidor utilizaremos la clase ServerSocket y en el cliente la clase Socket, utilizando el método accept() para esperar algún cliente. Una vez establecida la conexión podremos intercambiar datos utilizando los *Streams*.

SERVIDORES

Como se ha visto antes Java admite conectores de servidor. Los ServerSocket se utilizan para crear servidores en Internet (programas que están esperando a programas cliente locales). Estos conectores tienen un método adicional accept() que se bloquea y espera que un cliente inicie la comunicación para devolver un Socket y devolver la comunicación al cliente.

OBJETO SERVERSOCKET :

```
ServerSocket obj_server=new ServerSocket(int puerto);
```

OBJETO SOCKET : para poder recibir al cliente.

```
Socket obj_cliente;
```

RECIBIR LA COMUNICACIÓN (ACCEPT):

```
obj_cliente=obj_server.accept();
```

DEVOLVER LA COMUNICACIÓN(PRINTSTREAM): muy parecido a un println.

```
PrintStream obj;
obj=new PrintStream(obj_cliente.getOutputStream());
```

CLIENTES

Los conectores TCP/IP (*Socket*) se utilizan para implementar conexiones entre nodos de Internet . Estos conectores pueden examinar en cualquier momento la información de dirección y puerto asociado con ellos y acceder a los flujos E/S que se han visto anteriormente.

CREAR SOCKET: dos métodos para la creación.

```
Socket obj_clie=new Socket(direccion, int puerto);

Socket obj_clie=new Socket(String nodo, int puerto);
```

METODOS ASOCIADOS:

obj_clie.*metodo*();

metodo()	***DESCRIPCIÓN***
getInetAddress()	Devuelve la dirección a la que esta conectado.
getPort()	Devuelve el puerto remoto que esta conectado.
getLocalPort()	Devuelve el puerto local que esta conectado.
getInputStream()	Devuelve el flujo de entrada al cliente.
getOutputStream()	Devuelve el flujo de salida al cliente.
close()	Cierra los Streams.

La clase **InetAddress** se utiliza para obtener los nombres de los nodos y sus direcciones IP. Los métodos de esta clase están siempre asociados a *Socket*. La excepción que lanzan en caso de error es UnknownHostException. Nos van a servir de gran apoyo a la hora de utilizar los Socket. La clase InetAddress no tiene constructores visibles, son métodos estáticos.

BUSCAR NODO LOCAL:

InetAddress obj=InetAddress.getLocalHost();

BUSCAR NODO POR NOMBRE:

InetAddress obj=InetAddress.getByName("nombre");

BUSCAR TODOS LOS NODOS POR NOMBRE:

InetAddress obj[]=InetAddress.getAllByName("nombre");

OBTENER NOMBRE DE HOST:

String=obj_socket.getHostName();

OBTENER DIRECCION:

byte[]=obj_socket.getAddress();

OBTENER NOMBRE E IP DEL NODO:

String=obj_socket.toString();

EJEMPLO: En este ejemplo se crean dos códigos fuente, un para el servidor y otro para el cliente. Para ejecutarlos deberemos tener una ventana para el servidor y otra para el cliente. Primero ejecutamos el servidor que esta funcionando indefinidamente y

en la otra ventana iremos ejecutando repetidas veces el cliente. El resultado es un contador de visitas a ese servidor que nosotros hemos creado.

```
/*****PROGRAMA SERVIDOR  (SOC.JAVA)******/

import java.io.*;
import java.net.*;

public class soc{

  public static void main(String args[]){
    ServerSocket servidor;
    Socket cliente;
    int num_cliente=0;

    try{
      servidor=new ServerSocket(5000);
      do{
        num_cliente++;
        cliente=servidor.accept();
        System.out.println("Cliente: "+num_cliente);
        PrintStream ps;
         ps=new PrintStream(cliente.getOutputStream());
        ps.println("Es el cliente: "+num_cliente);
        cliente.close();
      }while(true);
    }
    catch(Exception e){}
  }
}
```

```
/**** PROGRAMA CLIENTE (CLIE.JAVA)*****/

import java.io.*;
import java.net.*;

public class clie{
  public static void main(String args[]){
    InetAddress direccion;
    Socket servidor;
    int num_cliente=0;

    try{
      direccion=InetAddress.getLocalHost();
      servidor=new Socket(direccion,5000);
      DataInputStream datos;
            datos=new DataInputStream(servidor.getInputStream());
      System.out.println(datos.readLine());
      servidor.close();
    }
    catch(Exception e){}
  }
}
```

20. TRABAJO EN RED

Java permite trabajar a través de la red con ficheros mediante el paquete java.net. La clase que contiene los métodos para estos procesos es la clase URL. En primer lugar tendremos que crear el objeto para especificar la dirección del recurso (fichero). A continuación debemos establecer la conexión de red mediante la clase URLConnection. Una vez hecho todo esto lo único que nos queda es leer y escribir datos mediante los *Streams*.

CREAR OBJETO: Todos los parámetros excepto el puerto que es un int son string.

```
URL objeto_url=new URL(protocolo,nodo,puerto,archivo);
URL objeto_url=new URL(protocolo,nodo,archivo);
```

CREAR CONEXIÓN:

```
URLConnection objeto=objeto_url.openConnection();
```

EJEMPLO:

```
import java.awt.*;
import java.applet.*;
import java.net.*;
import java.io.*;

public class red1 extends Applet{
    TextArea caja=new TextArea("",50,50);
      String texto;

    public void init(){
      setLayout(null);
      add(caja);
            caja.reshape(70,10,100,100);

            try{
         URL conx=new URL ("http://www.r.es/uno.htm");
      URLConnection fil=conx.openConnection();
              DataInputStream datos;
              datos=new DataInputStream(fil.getInputStream());

        do{
          texto=datos.readLine();
          caja.appendText(texto+"\n");
        }while(texto !=null);

        datos.close();
  }
      catch(IOException e){}
      }
  }
```

Autor : Ricardo Amezua Martinez (netamezua@mixmail.com)
Para **Softdownload Argentina**
www.softdownload.com.ar
info@softdownload.com.ar
Año 2001