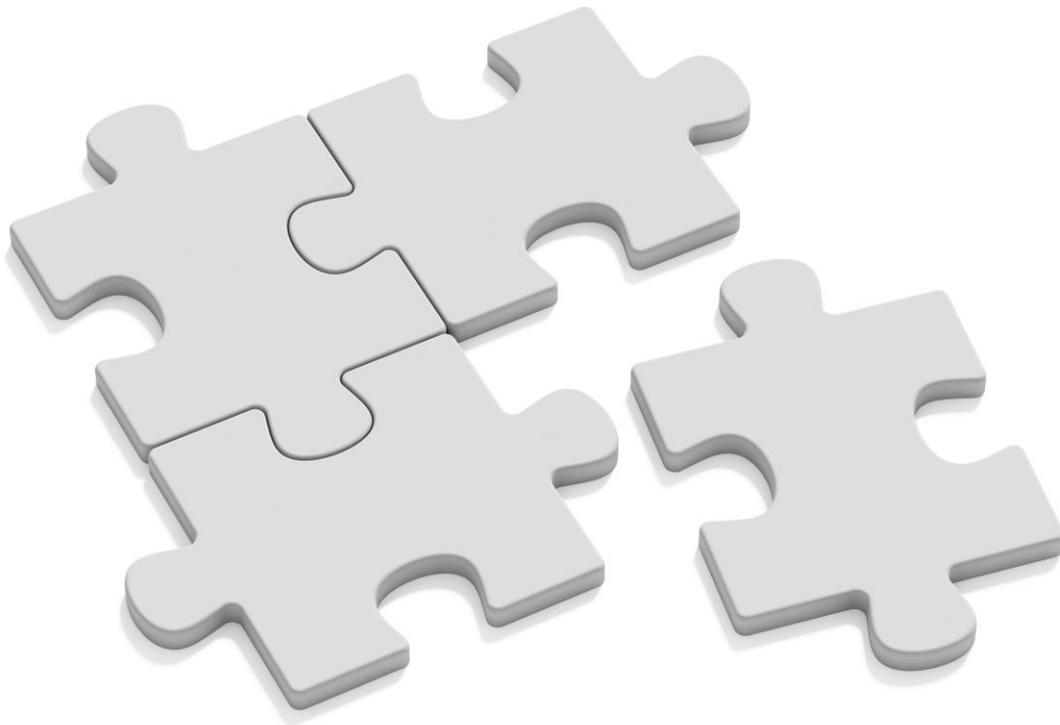




Universidad Tecnológica Nacional



2019

CÁTEDRA DE LENGUAJE DE PROGRAMACIÓN JAVA

Ings. Mario Bressano & Adrián Meca

ENVÍO 01/2019



Introducción al Lenguaje Java

Paquetes

Un **Paquete** en Java es un contenedor de clases que permite agrupar las distintas partes de un programa cuya funcionalidad tienen elementos comunes.

El uso de paquetes proporciona las siguientes ventajas:

- Agrupamiento de clases con características comunes.
- Reutilización de código.
- Mayor seguridad al existir niveles de acceso.

Contenidos de Paquetes

Un paquete puede contener:

- Clases
- Interfaces
- Tipos Enumerados
- Anotaciones

Uso de Paquetes

En el código de programación Java se usa la palabra reservada `package` para especificar a qué paquete pertenecen. Suele indicarse como primera sentencia:

```
package java.awt.event;
```

Para usar un paquete dentro del código se usa la declaración `import`. Si sólo se indica el nombre del paquete:

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

se importan todas las clases que contiene. Si además del nombre del paquete se especifica una clase, sólo se importa esa clase:

```
import java.awt.event.ActionEvent;
```

Después de añadir alguna de estas sentencias, se puede hacer referencia a la clase `ActionEvent` usando su nombre:

```
ActionEvent myEvent = new ActionEvent();
```



Si no se hubiera importado la clase o el paquete, cada vez que tuviéramos que usarla habría que especificarlo:

```
java.awt.event.ActionEvent myEvent = new java.awt.event.ActionEvent();
```

Paquetes importantes para Java

Estos son los paquetes más importantes de la API de Java:

Paquete	Descripción
java.applet	Contiene clases para la creación de applets.
java.awt	Contiene clases para crear interfaces de usuario con ventanas.
java.io	Contiene clases para manejar la entrada/salida.
java.lang	Contiene clases variadas pero imprescindibles para el lenguaje, como <i>Object</i> , <i>Thread</i> , <i>Math</i> ...
java.net	Contiene clases para soportar aplicaciones que acceden a redes TCP/IP.
java.util	Contiene clases que permiten el acceso a recursos del sistema, etc.
java.swing	Contiene clases para crear interfaces de usuario mejorando la AWT.

Declaración de la clase

La clase se declara mediante la línea `public class Ejercicio`. En el caso más general, la declaración de una clase puede contener los siguientes elementos:

```
[public] [final | abstract] class Clase [extends ClaseMadre] [implements Interfase1 [, Interfase2 ]...]
```

o bien, para interfaces:

```
[public] interface Interfase [extends InterfaseMadre1 [, InterfaseMadre2 ]...]
```

Como se ve, lo único obligatorio es **class** y el nombre de la clase.

Public, final o abstract

Definir una clase como pública (**public**) significa que puede ser usada por cualquier clase en cualquier paquete. Si no lo es, solamente puede ser utilizada por clases del mismo paquete (más sobre paquetes luego; básicamente, se trata de un grupo de clases e interfaces relacionadas, como los paquetes de biblioteca incluidos con Java).

Una clase final (**final**) es aquella que no puede tener clases que la hereden. Esto se utiliza básicamente por razones de seguridad (para que una clase no pueda ser reemplazada por otra que la herede), o por diseño de la aplicación.



Una clase abstracta (**abstract**) es una clase que puede tener herederas, pero no puede ser instanciada. Es, literalmente, abstracta (como la clase *Number* definida en `java.lang`). ¿Para qué sirve? Para modelar conceptos. Por ejemplo, la clase *Number* es una clase abstracta que representa cualquier tipo de números (y sus métodos no están implementados: son abstractos); las clases descendientes de ésta, como *Integer* o *Float*, sí implementan los métodos de la madre *Number*, y se pueden instanciar.

Por lo dicho, una clase no puede ser **final** y **abstract** a la vez (ya que la clase abstract requiere descendientes...)

Extends

La instrucción **extends** indica de qué clase desciende la nuestra. Si se omite, Java asume que desciende de la superclase **Object**.

Cuando una clase desciende de otra, esto significa que hereda sus atributos y sus métodos (es decir que, a menos que los redefinamos, sus métodos son los mismos que los de la clase madre y pueden utilizarse en forma transparente, a menos que sean *privados* en la clase madre o, para subclases de otros paquetes, protegidos o propios del paquete).

Implements

Una interfase (**interface**) es una clase que declara sus métodos pero no los implementa; cuando una clase implementa (**implements**) una o más interfaces, debe contener la implementación de todos los métodos (con las mismas listas de parámetros) de dichas interfaces.

Esto sirve para dar un ascendiente común a varias clases, obligándolas a implementar los mismos métodos y, por lo tanto, a comportarse de forma similar en cuanto a su interfase con otras clases y subclases.

Interface

Una interfase (**interface**), como se dijo, es una clase que no implementa sus métodos sino que deja a cargo la implementación a otras clases. Las interfaces pueden, asimismo, descender de otras interfaces pero no de otras clases.

Todos sus métodos son por definición abstractos y sus atributos son finales (aunque esto no se indica en el cuerpo de la interfase).

Son útiles para generar relaciones entre clases que de otro modo no están relacionadas (haciendo que implementen los mismos métodos), o para distribuir paquetes de clases indicando la estructura de la interfase pero no las clases individuales (objetos anónimos).

Si bien diferentes clases pueden implementar las mismas interfaces, y a la vez descender de otras clases, esto no es en realidad herencia múltiple ya que una clase no puede heredar atributos ni métodos de una interfase; y las clases que implementan una interfase pueden no estar ni siquiera relacionadas entre sí.

El cuerpo de la clase



El cuerpo de la clase, encerrado entre { y }, es la lista de atributos (variables) y métodos (funciones) que constituyen la clase.

No es obligatorio, pero en general se listan primero los atributos y luego los métodos.

Declaración de atributos

En Java no hay variables globales; todas las variables se declaran dentro del cuerpo de la clase o dentro de un método. Las variables declaradas dentro de un método son **locales** al método; las variables declaradas en el cuerpo de la clase se dice que son *miembros* de la clase y son accesibles por todos los métodos de la clase.

Por otra parte, además de los atributos de la propia clase se puede acceder a todos los atributos de la clase de la que desciende; por ejemplo, cualquier clase que descienda de la clase **Polygon** hereda los atributos *npoints*, *xpoints* e *ypoints*.

Finalmente, los atributos miembros de la clase pueden ser *atributos de clase* o *atributos de instancia*; se dice que son atributos *de clase* si se usa la palabra clave **static**: en ese caso la variable es única para todas las instancias (objetos) de la clase (ocupa un único lugar en memoria). Si no se usa *static*, el sistema crea un lugar nuevo para esa variable con cada instancia (o sea que es independiente para cada objeto).

La declaración sigue siempre el mismo esquema:

[private|protected|public] [static] [final] [transient] [volatile] Tipo NombreVariable [= Valor];

Private, protected o public

Java tiene 4 tipos de acceso diferente a las variables o métodos de una clase: privado, protegido, público o por paquete (si no se especifica nada).

De acuerdo a la forma en que se especifica un atributo, objetos de otras clases tienen distintas posibilidades de accederlos:

<i>Acceso desde:</i>	private	protected	public	(package)
la propia clase	S	S	S	S
subclase en el mismo paquete	N	S	S	S
otras clases en el mismo paquete	N	S	S	S
subclases en otros paquetes	N	X	S	N
otras clases en otros paquetes	N	N	S	N



S: puede acceder

N: no puede acceder

X: puede acceder al atributo en objetos que pertenezcan a la subclase, pero no en los que pertenecen a la clase madre. Es un caso especial ; más adelante veremos ejemplos de todo esto.

Static y Final

Como ya se vio, **static** sirve para definir un atributo como *de clase*, o sea único para todos los objetos de la clase.

En cuanto a **final**, como en las clases, determina que un atributo no pueda ser sobrescrito o redefinido. O sea: no se trata de una variable, sino de una *constante*

Los tipos de Java

Los tipos de variables disponibles son básicamente 3:

- tipos básicos (no son objetos)
- arreglos (arrays)
- clases e interfases

Con lo que vemos que cada vez que creamos una clase o interfase estamos definiendo un nuevo tipo.

Los **tipos básicos** son:

Tipo	Tamaño/Formato	Descripción
byte	8-bit complemento a 2	Entero de un byte
short	16-bit complemento a 2	Entero corto
int	32-bit complemento a 2	Entero
long	64-bit complemento a 2	Entero largo
float	32-bit IEEE 754	Punto flotante, precisión simple
double	64-bit IEEE 754	Punto flotante, precisión doble
char	16-bit caracter Unicode	Un caracter
boolean	true, false	Valor booleano (verdadero o falso)



Los **arrays** son arreglos de cualquier tipo (básico o no). Por ejemplo, existe una clase *Integer*, un arreglo de objetos de dicha clase se notaría:

```
Integer vector[ ];
```

Los arreglos siempre son dinámicos, por lo que **no es válido** poner algo como:

```
Integer cadena[5];
```

Aunque sí es válido inicializar un arreglo, como en:

```
int días[ ] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

```
char letras[ ] = { 'E', 'F', 'M', 'A', 'M', 'J', 'J', 'A', 'S', 'O', 'N', 'D' };
```

```
String nombres[ ] = new String[12];
```

Nota al margen: no confundir un String (*cadena* de caracteres) con un *arreglo* de caracteres! Son cosas bien distintas!

Ya hablaremos más adelante de las clases String y StringBuffer.

En Java, para todas las variables de tipo básico se accede al valor asignado a la misma directamente (no se conoce la dirección de memoria que ocupa). Para las demás (arrays, clases o interfases), se accede a través de un puntero a la variable. El valor del puntero no es accesible ni se puede modificar (como en C); Java no necesita esto y además eso atendería contra la robustez del lenguaje.

De hecho, en Java no existen los tipos **pointer**, **struct** o **union**. Un objeto es más que una estructura, y las uniones no se hacen necesarias con un método de programación adecuado (y además se evita la posibilidad de acceder a los datos incorrectamente).

Algo más respecto a los arreglos: ya que Java gestiona el manejo de memoria para los mismos, y lanza excepciones si se intenta violar el espacio asignado a una variable, se evitan problemas típicos de C como acceder a lugares de memoria prohibidos o fuera del lugar definido para la variable (como cuando se usa un subíndice más grande que lo previsto para un arreglo...).

Y los métodos...

Los métodos, como las clases, tienen una declaración y un cuerpo.

La declaración es del tipo:

```
[private|protected|public] [static] [abstract] [final] [native] [synchronized]
TipoDevuelto NombreMétodo ( [tipo1 nombre1[, tipo2 nombre2 ]...] ) [throws excepción1
[,excepción2]... ]
```

Básicamente, los métodos son como las funciones de C: implementan, a través de funciones, operaciones y estructuras de control, el cálculo de algún parámetro que es el que devuelven al objeto que los llama. Sólo pueden devolver **un** valor (del tipo *TipoDevuelto*), aunque pueden no



devolver ninguno (en ese caso *TipoDevuelto* es **void**). Como ya veremos, el valor de retorno se especifica con la instrucción **return**, dentro del método.

Los métodos pueden utilizar valores que les pasa el objeto que los llama (*parámetros*), indicados con *tipo1 nombre1, tipo2 nombre2...* en el esquema de la declaración.

Estos parámetros pueden ser de cualquiera de los tipos ya vistos. Si son tipos básicos, el método recibe el *valor* del parámetro; si son arrays, clases o interfases, recibe un puntero a los datos (*referencia*). Veamos un pequeño ejemplo:

```
public int AumentarCuenta(int cantidad) {
    cnt = cnt + cantidad;
    return cnt;
}
```

Este método, si lo agregamos a la clase **Contador**, le suma *cantidad* al acumulador **cnt**. En detalle:

- el método recibe un valor entero (*cantidad*)
- lo suma a la variable de instancia *cnt*
- devuelve la suma (*return cnt*)

¿Cómo hago si quiero devolver más de un valor? Por ejemplo, supongamos que queremos hacer un método dentro de una clase que devuelva la posición del mouse.

Lo siguiente no sirve:

```
void GetMousePos(int x, int y) {
    x = ....;           // esto no sirve!
    y = ....;           // esto tampoco!
}
```

porque el método no puede modificar los parámetros *x* e *y* (que han sido pasados por valor, o sea que el método recibe el valor numérico pero no sabe a dónde están las variables en memoria).

La solución es utilizar, en lugar de tipos básicos, una clase:

```
class MousePos { public int x, y; }
```

y luego utilizar esa clase en nuestro método:

```
void GetMousePos( MousePos m ) {
    m.x = .....;
    m.y = .....;
}
```



Un método **static** de la clase es un método que sólo accesa variables de la clase. Se definen usando el atributo `static`

El resto de la declaración

Public, **private** y **protected** actúan exactamente igual para los métodos que para los atributos, así que veamos el resto.

Los métodos estáticos (**static**), son, como los atributos, métodos *de clase*; si el método no es **static** es un método *de instancia*. El significado es el mismo que para los atributos: un método *static* es compartido por todas las instancias de la clase.

Ya hemos hablado de las clases abstractas; los métodos abstractos (**abstract**) son aquellos de los que se da la declaración pero no la implementación (o sea que consiste sólo del encabezamiento). Cualquier clase que contenga al menos un método abstracto (o cuya clase madre contenga al menos un método abstracto que no esté implementado en la hija) es una clase abstracta.

Es **final** un método que no puede ser redefinido por ningún descendiente de la clase.

Las clases **native** son aquellas que se implementan en otro lenguaje (por ejemplo C o C++) propio de la máquina. Sun aconseja utilizarlas bajo riesgo propio, ya que en realidad son ajenas al lenguaje. Pero la posibilidad de usar viejas bibliotecas que uno armó y no tiene ganas de reescribir existe!.

Las clases **synchronized** permiten sincronizar varios *threads* para el caso en que dos o más accedan concurrentemente a los mismos datos. De nuevo, más detalles habrá en el futuro, cuando hablemos de threads.

Finalmente, la cláusula **throws** sirve para indicar que la clase genera determinadas excepciones. También hablaremos de las excepciones más adelante.

InputStream: EL OBJETO System.in

Al igual que Java nos ofrece System.out para escribir en pantalla, tenemos System.in para leer de ella. System.in es un objeto de una clase de java que se llama InputStream.

Para java, un InputStream es cualquier cosa de la que se leen bytes. Puede ser el teclado, un fichero, un socket, o cualquier otro dispositivo de entrada. Esto, por un lado es una ventaja. Si todas esas cosas son InputStream, podemos hacer código que lea de ellas sin saber qué estamos leyendo.

Como un InputStream es para leer bytes, sólo tiene métodos para leer bytes. Nosotros queremos leer palabras o números del teclado, no bytes. Si escribimos en el teclado una A mayúscula y la leemos con System.in, obtendremos un entero de valor 65, que es el valor del byte correspondiente a la A.

Reader



Para Java, una clase Reader es una clase que lee caracteres. Esto se parece más a lo que queremos. Un Reader tiene métodos para leer caracteres. Con esta clase ya podríamos trabajar. La pena es que seguimos teniendo System.in, que es un InputStream y no un Reader.

¿Cómo convertimos el System.in en Reader?.

Hay una clase en java, la InputStreamReader, que nos hace esta conversión. Para obtener un Reader, únicamente tenemos que instanciar un InputStreamReader pasándole en el constructor un InputStream.

El código es el siguiente

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

Estamos declarando una variable "isr" de tipo InputStreamReader. Creamos un objeto de esta clase haciendo new InputStreamReader(...). Entre paréntesis le pasamos el InputStream que queremos convertir a Reader, en este caso, el System.in

Ya tenemos el Reader. ¿Cómo funciona exactamente?

- InputStreamReader es un Reader. Se comporta igual que in Reader y se puede poner en cualquier sitio que admita un Reader. Es decir, podemos leer de él caracteres.
- Al constuirlo le hemos pasado un InputStream, en concreto, System.in. InputStreamReader de alguna forma se lo guarda dentro.
- Cuando a InputStreamReader le pedimos caracteres, él le pide al InputStream que tiene guardado dentro los bytes, los convierte a caracteres y nos los devuelve.

LA CLASE BufferedReader

La clase InputStreamReader nos da los caracteres sueltos. Si estamos leyendo de teclado, el que usa el programa puede escribir 10 caracteres o 20 o 13. Si usamos InputStreamReader, como lee caracteres sueltos, Tenemos que decirle cuántos queremos (que no lo sabemos), o bien ir pidiendo de uno en uno hasta que no haya más.

Es problemático si sólo tuviéramos la clase InputStreamReader ya que deberíamos repetir el código por muchos lados. Para el caso concreto de leer de teclado, sería ideal si hubiese una clase en java que nos lea de golpe todo lo que ha escrito el usuario de nuestro programa y nos lo diera de un golpe.

Como la gente de Java son muy listos, esa clase existe en Java. Se llama BufferedReader. El mecanismo para obtener un BufferedReader a partir de otro Reader cualquiera (por ejemplo el InputStreamReader), es similar al que usamos antes. Lo instanciamos pasándole en el constructor el Reader.

El código es

```
BufferedReader br = new BufferedReader (isr);
```

El funcionamiento de esta clase es igual que el InputStreamReader. Cuando le pedmos una línea completa de caracteres (un String), ella se lo pide al Reader que tenga dentro, los convierte en String y nos lo devuelve.

Para pedirle un String, se usa el método readLine(). Este método lee todos los caracteres tecleados (recibidos si fuera otro dispositivo de entrada) hasta que encuentra la pulsación de la tecla <INTRO>, <RETURN>



String texto = br.readLine();

Esto lee del teclado un String completo y lo guarda en una variable "texto".



Ejercicio Básico 01: Imprimir en consola la frase "ESTO ES JAVA"

```
import java.*;

public class Ejercicio_basico_00 {

    public static void main(String args[])
    {
        System.out.println("\n ESTO ES JAVA");
    }
}
```

- La palabra void indica que el método main no retorna ningún valor.
- La forma (String args[]) es la definición de los argumentos que recibe el método main. En este caso se recibe un argumento. Los paréntesis [] indican que el argumentos es un arreglo y la palabra String es el tipo de los elementos del arreglo.

Por lo tanto main recibe como argumento un arreglo de strings que corresponden a los argumentos con que se invoca el programa.

Pero la actualidad nos indica que lo visual manda sobre el texto por eso veremos el trabajo que realizaremos con componentes modales del paquete SWING

JOptionPane

Es un componente al estilo Pop Up que sirve como un prompt o ventana de datos donde se puede pedir o desplegar información.

Siempre que pensemos usar la clase JOptionPane, debemos primero importarla.

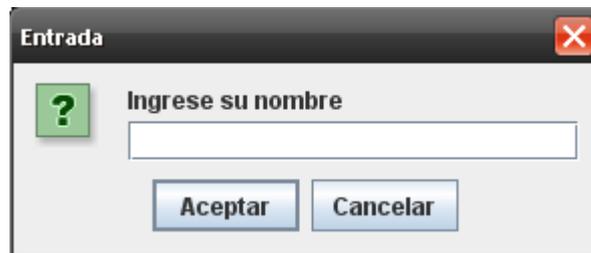
```
import javax.swing.JOptionPane;
```

Solicitar datos:

Al solicitar información, este se guarda en un String usando el método *showInputDialog*

Codigo:

```
String nombre;
nombre = JOptionPane.showInputDialog("Ingrese su nombre");
```

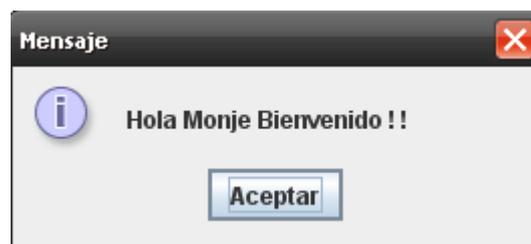


Mostrar mensaje:

Para mostrar un mensaje en una ventana solo debemos usar el método `showMessageDialog`

Código:

```
JOptionPane.showMessageDialog(null,"Hola"+nombre+" Bienvenido ! !");
```



Ejemplo Suma:

```
String numeroA;
String numeroB;

int numero1;
int numero2;
int resultado;

numeroA = JOptionPane.showInputDialog("Digite primer numero");
numeroB = JOptionPane.showInputDialog("Digite primer numero");

numero1=Integer.parseInt(numeroA);
numero2=Integer.parseInt(numeroB);
resultado=numero1+numero2;

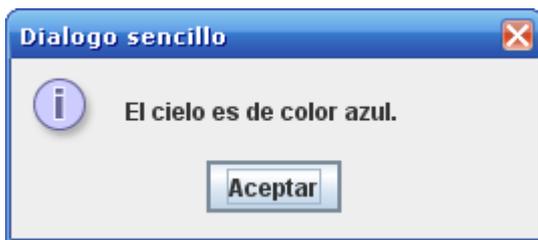
JOptionPane.showMessageDialog(null,"La suma es: "+resultado+" resultados");
```

showMessageDialog

Es un diálogo simple que presenta un botón de “Aceptar”. Se puede especificar fácilmente el mensaje, el icono, y el título que el diálogo exhibe. Aquí están algunos ejemplos del `showMessageDialog` que se usan:

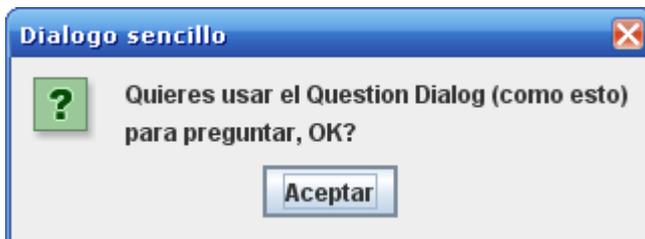
Código:

```
JOptionPane.showMessageDialog(ventana, “El cielo es de color azul.”, “Dialogo sencillo”,  
JOptionPane.INFORMATION_MESSAGE);
```



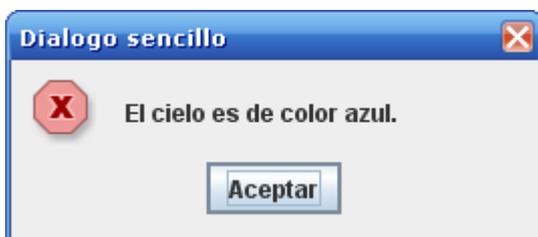
Código:

```
JOptionPane.showMessageDialog(ventana, “Quieres usar el Question Dialog ”+ “(como esto)\n”  
+ “para preguntar, OK?”, “Dialogo sencillo”, JOptionPane.QUESTION_MESSAGE);
```



Código:

```
JOptionPane.showMessageDialog(ventana, “El cielo es de color azul.”, “Dialogo sencillo”,  
JOptionPane.ERROR_MESSAGE);
```



Código:

```
JOptionPane.showMessageDialog(ventana, “El cielo es de color azul.”, “Dialogo sencillo”,  
JOptionPane.INFORMATION_MESSAGE, icono);
```



Código:

```
JOptionPane.showMessageDialog(ventana, "El cielo es de color azul.", "Dialogo sencillo",
JOptionPane.PLAIN_MESSAGE);
```

Los argumentos:

- **Component parentComponent:** Es el componente padre del diálogo, si el argumento pasa como *null* simplemente el diálogo se mostrará en centro de la pantalla y no tendrá el foco principal si hay JFrame o alguna otra ventana
- **Object message:** El mensaje a mostrar, es el texto principal
- **String title:** El título que lleva el diálogo en la barra de tareas
- **int messageType:** El tipo de diálogo, puede ser del tipo DEFAULT_OPTION, YES_NO_OPTION, YES_NO_CANCEL_OPTION, OK_CANCEL_OPTION
- **int messageType:** El tipo de icono que tendrá el diálogo, puede ser PLAIN_MESSAGE (sin icono), ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE
- **Icon icon:** El icono personalizado que tendrá el diálogo
- **Object[] options:** Estas son las opciones que tendrá el diálogo, son útiles cuando creamos nuestros propios diálogos

Más sobre JOptionPane (descargado de Wikipedia)

JOptionPane

JOptionPane tiene dos juegos repetidos de ventanas de aviso/confirmación. Una para ventanas normales y otra para *JInternalFrame*. Puesto que son lo mismo, vamos a ver aquí sólo los de ventanas normales. Las distintas posibilidades que tenemos de *JOptionPane* son:

JOptionPane.showOptionDialog()

Tenemos un método *JOptionPane.showOptionDialog()* que nos muestra la ventana más configurable de todas, en ella debemos definir todos los botones que lleva. De hecho, las demás ventanas disponibles con *JOptionPane* se construyen a partir de esta. Por ello, al método debemos pasarle muchos parámetros:

- **parentComponent:** A partir de este componente, se intentará determinar cual es la ventana que debe hacer de padre del *JOptionPane*. Se puede pasar *null*, pero conviene pasar, por ejemplo, el botón desde el cual se lanza la acción que provoca que se visualice el *JOptionPane*.

De esta manera, la ventana de aviso se visualizar  sobre el bot n y no se podr  ir detr s del mismo si hacemos click en otro sitio.

- **message:** El mensaje a mostrar, habitualmente un *String*, aunque vale cualquier *Object* cuyo m todo *toString()* devuelva algo con sentido.
- **title:** El t tulo para la ventana.
- **optionType:** Un entero indicando qu  opciones queremos que tenga la ventana. Los posibles valores son las constantes definidas en *JOptionPane*: *DEFAULT_OPTION*, *YES_NO_OPTION*, *YES_NO_CANCEL_OPTION*, o *OK_CANCEL_OPTION*.
- **messageType:** Un entero para indicar qu  tipo de mensaje estamos mostrando. Este tipo servir  para que se determine qu  icono mostrar. Los posibles valores son constantes definidas en *JOptionPane*: *ERROR_MESSAGE*, *INFORMATION_MESSAGE*, *WARNING_MESSAGE*, *QUESTION_MESSAGE*, o *PLAIN_MESSAGE*
- **icon:** Un icono para mostrar. Si ponemos *null*, saldr  el icono adecuado seg n el par metro *messageType*.
- **options:** Un array de *objects* que determinan las posibles opciones. Si los objetos son componentes visuales, aparecer n tal cual como opciones. Si son *String*, el *JOptionPane* pondr  tantos botones como *String*. Si son cualquier otra cosa, se les tratar  como *String* llamando al m todo *toString()*. Si se pasa *null*, saldr n los botones por defecto que se hayan indicado en *optionType*.
- **initialValue:** Selecci n por defecto. Debe ser uno de los *Object* que hayamos pasado en el par metro *options*. Se puede pasar *null*.

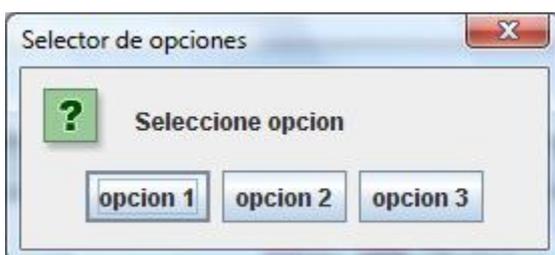
La llamada a *JOptionPane.showOptionDialog()* devuelve un entero que representa la opci n que ha seleccionado el usuario. La primera de las opciones del array es la posici n cero. Si se cierra la ventana con la cruz de la esquina superior derecha, el m todo devolver  -1.

Aqu  un ejemplo de c mo llamar a este m todo

```
int seleccion = JOptionPane.showOptionDialog(
    unComponentePadre,
    "Seleccione opcion",
    "Selector de opciones",
    JOptionPane.YES_NO_CANCEL_OPTION,
    JOptionPane.QUESTION_MESSAGE,
    unIcono, // null para icono por defecto.
    new Object[] { "opcion 1", "opcion 2", "opcion 3" }, // null para YES, NO y CANCEL
    "opcion 1");
```

```
if (seleccion != -1)
    System.out.println("seleccionada opcion " + (seleccion + 1));
```

y la ventana que se obtiene con el c digo anterior



JOptionPane.showInputDialog()

Tenemos varios métodos *JOptionPane.showInputDialog()* y la diferencia entre ellos es que tienen más o menos parámetros, según queramos aceptar o no las opciones por defecto. Los parámetros y sus significados son muy similares a los del método *showOptionDialog()*, pero hay una diferencia.

Si usamos los métodos que no tienen array de opciones, la ventana mostrará una caja de texto para que el usuario escriba la opción que desee (un texto libre). Si usamos un método que tenga un array de opciones, entonces aparecerá en la ventana un *JComboBox* en vez de una caja de texto, donde estarán las opciones que hemos pasado.

Aquí un par de trozos de código, el primero para conseguir una caja de texto,

```
// Con caja de texto
String seleccion = JOptionPane.showInputDialog(
    unComponentePadre,
    "Input dialog",
    JOptionPane.QUESTION_MESSAGE); // el icono será un interrogante
```

```
System.out.println("El usuario ha escrito "+selección);
```

y la imagen que obtenemos con este código

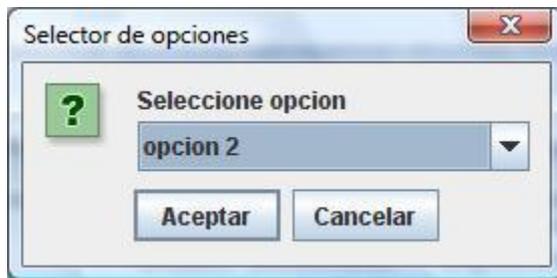


En este segundo ejemplo, damos todas las opciones que queremos, obteniendo un *JComboBox*

```
// Con JComboBox
Object seleccion = JOptionPane.showInputDialog(
    unComponentePadre,
    "Seleccione opción",
    "Selector de opciones",
    JOptionPane.QUESTION_MESSAGE,
    unIcono, // null para icono defecto
    new Object[] { "opcion 1", "opcion 2", "opcion 3" },
    "opcion 1");
```

```
System.out.println("El usuario ha elegido "+selección);
```

y esta es la imagen que se obtiene.



JOptionPane.showMessageDialog()

Esta es la más sencilla de todas, sólo muestra una ventana de aviso al usuario. La ejecución se detiene hasta que el usuario cierra la ventana. Hay varios métodos con el mismo nombre y más o menos parámetros, en función de si aceptamos las opciones por defecto (icono, por ejemplo) o queremos cambiar alguna cosa. Un trozo de código para llamarlo

```
JOptionPane.showMessageDialog(
    componentePadre,
    "Un aviso");

System.out.println("ya estas avisado");
```

JOptionPane.showConfirmDialog()

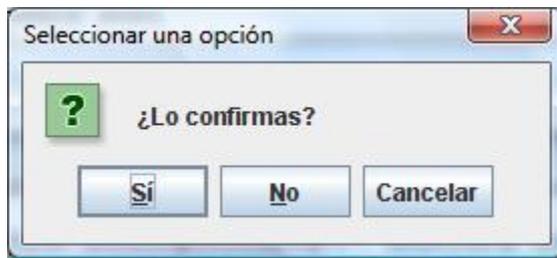
Este método muestra una ventana pidiendo una confirmación al usuario, estilo "*¿Seguro que lo quieres borrar todo?*" y da al usuario opción de aceptar o cancelar ese borrado masivo que está a punto de hacer. El método devuelve un entero indicando la respuesta del usuario. Los valores de ese entero puede ser alguna de las constantes definidas en *JOptionPane*: *YES_OPTION*, *NO_OPTION*, *CANCEL_OPTION*, *OK_OPTION*, *CLOSED_OPTION*. Por supuesto, hay métodos iguales con más o menos parámetros para configurar las cosas más o menos.

El siguiente ejemplo de código

```
int confirmado = JOptionPane.showConfirmDialog(
    componentePadre,
    "¿Lo confirmas?");

if (JOptionPane.OK_OPTION == confirmado)
    System.out.println("confirmado");
else
    System.out.println("vale... no borro nada...");
```

muestra la siguiente imagen



JOptionPane con timeout

La llamada a cualquiera de los métodos anteriores para la ejecución de nuestro programa hasta que el usuario cierre la ventana seleccionando alguna de las opciones. A veces no nos interesa quedarnos bloqueados indefinidamente esperando la respuesta del usuario, por ejemplo, podemos mostrarle un mensaje para informarle de algo con un `showMessageDialog()` y si en 30 segundos no lo ha cerrado, cerrarlo nosotros automáticamente desde nuestro código y continuar con lo que estábamos haciendo.

Desgraciadamente, `JOptionPane` no tiene ninguna opción con `timeout`, por lo que hacer esto no es inmediato. Debemos lanzar un `Timer` o un hilo para cerrar la ventana desde ese hilo.

Y desgraciadamente, los métodos que hemos visto de `JOptionPane` no nos devuelven una referencia a la ventana que acabamos de visualizar, por lo que tampoco podremos ocultarla desde código. La solución es que creamos nosotros la ventana y nos guardemos una referencia a ella. Afortunadamente, no debemos hacernos la ventana desde cero. Podemos hacer un `new` de `JOptionPane` para obtener el panel de dicha ventana y llamar al método `createDialog()` para obtener y visualizar el `JDialog` correspondiente. El código completo para esto puede ser como el del ejemplo

```
package com.chuidiang.ejemplos;

import java.awt.Component;

import javax.swing.JDialog;
import javax.swing.JOptionPane;

/**
 * Muestra un JOptionPane con un timeout para que se cierre automáticamente.
 *
 * @author chuidiang
 *
 */
public class JOptionPaneConTimeOut
{
    private static JOptionPane option = new JOptionPane("",
        JOptionPane.INFORMATION_MESSAGE);

    private static JDialog dialogo = null;

    /**
     * Solo hace caso a padre la primera vez que se llama a este método. La
     * llamada a este método se queda bloqueada hasta que el usuario cierra el
     * JOptionPane o pasa el timeout.
     *
     */
}
```



```

* @param padre
* @param texto
* @param titulo
* @param timeout
*     En mili segundos
*/
public static void visualizaDialogo( Component padre, String texto,
    String titulo, final long timeout)
{
    option.setMessage(texto);
    if ( null == dialogo )
    {
        dialogo = option.createDialog(padre, titulo);
    }
    else
    {
        dialogo.setTitle(titulo);
    }

    Thread hilo = new Thread()
    {
        public void run()
        {
            try
            {
                Thread.sleep(timeout);
                if ( dialogo.isVisible() )
                {
                    dialogo.setVisible(false);
                }
            }
            catch ( InterruptedException e )
            {
                e.printStackTrace();
            }
        }
    };
    hilo.start();

    dialogo.setVisible(true);
}
}

```

La clase del ejemplo tiene un método estático `visualizaDialogo()` al que podemos llamar para mostrar un aviso que se cierre automáticamente si el usuario no lo cierra antes. Esta clase/método es sólo de ejemplo y no permite configurar todo lo que se puede configurar en el `JOptionPane`, no queremos complicar la clase en exceso para esta explicación. Vamos viendo algunos detalles.

Creamos el `JOptionPane` de esta forma

```
private static JOptionPane option = new JOptionPane("",JOptionPane.INFORMATION_MESSAGE);
```



es decir, sin texto concreto "", y sólo para un mensaje de tipo información. Podríamos aquí usar cualquiera de los constructores que tiene `JOptionPane` y poner los parámetros que queramos. En sucesivas llamadas se pueden cambiar estos parámetros usando los métodos `set` al efecto

```
option.setMessage(texto);
```

Creamos el diálogo sólo la primera vez que se llama al método con

```
dialogo = option.createDialog(padre, titulo);
```

El título es el de la ventana (del `JDialog`) y podemos cambiarlo cuando queramos con

```
dialogo.setTitle(titulo);
```

Lo que no podemos cambiar es el padre, por eso el comentario que aparece de que sólo se hará caso al parámetro `parent` en la primera llamada.

Una vez configurada la ventana con el título y tal, creamos un hilo con una espera del tiempo indicado y pasado ese tiempo cerramos la ventana. Se comprueba antes si sigue visible, no sea que la haya cerrado el usuario antes de pasar el tiempo indicado. Recordamos arrancar el hilo con `hilo.start()` y nos aseguramos de hacer esto antes de hacer el diálogo visible, ya que al ser el diálogo modal, la llamada a `setVisible(true)` quedará bloqueada hasta que se cierre la ventana. Y después de arrancar el hilo, sólo queda hacer visible el diálogo

```
hilo.start();  
dialogo.setVisible(true);
```

Si el diálogo tuviera opciones que nos interesan, después de `dialogo.setVisible()`, podemos obtener la opción seleccionada con

```
Object valor = option.getValue();
```

Este valor será un `JOptionPane.UNINITIALIZED_VALUE` si el diálogo se ha cerrado por timeout, o bien el valor seleccionado `JOptionPane.YES_OPTION`, `JOptionPane.CANCEL_OPTION`, etc, o bien alguno de los `Object[]` que hayamos usado uno de los constructores que admite `Object[] options` como parámetro.

JDialog modal

A veces las ventanas de `JOptionPane` no son suficientes para lo que necesitamos. Por ejemplo, podemos querer pedir un dato más complejo que una simple selección de una opción. Y a veces mientras el usuario introduce ese dato complejo en la ventana y pulsa "Aceptar", queremos que nuestro código se pare en espera de la respuesta. Para estas situaciones tenemos los `JDialog` modales. En un `JDialog` podemos poner todos los componentes que queramos, haciendo la ventana todo lo compleja que queramos. Si ese `JDialog` es además modal, en el momento de hacerla visible llamando a `setVisible(true)`, el código se quedará parado en esa llamada hasta que la ventana se cierre. Podemos, por tanto, justo después del `setVisible(true)`, pedirle a la ventana los datos que ha introducido el operador.



Vamos a hacer aquí un ejemplo para ver todo esto. Como no quiero complicar demasiado el *JDialog* por claridad del ejemplo, el *JDialog* sólo tendrá un *TextField* dentro de él y al pulsar <intro> en ese *TextField*, después de haber escrito algo, el *JDialog* se cerrará. El código para conseguir esto es el siguiente:

```
import java.awt.Frame;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JDialog;
import javax.swing.JTextField;

/**
 * Dialogo modal que sirve de ventana de captura de datos.<br>
 * Contiene un JTextField en el que escribimos un texto y pulsando enter después
 * de escribir en el, la ventana se cierra.
 */
public class DialogoModal extends JDialog {
    private JTextField textField;

    /**
     * Constructor que pone titulo al dialogo, construye la ventana y la hace
     * modal.
     *
     * @param padre
     *      Frame que hace de padre de esta dialogo.
     */
    public DialogoModal(Frame padre) {

        // padre y modal
        super(padre, true);
        setTitle("Datos");
        textField = new JTextField(20);
        getContentPane().add(textField);

        // Se oculta la ventana al pulsar <enter> sobre el textfield
        textField.addActionListener(new ActionListener() {

            @Override
            public void actionPerformed(ActionEvent arg0) {
                setVisible(false);
            }
        });
    }

    /**
     * Devuelve el texto en el jtextfield
     *
     * @return el texto
     */
    public String getText() {
        return textField.getText();
    }
}
```



```
}  
}
```

Le hemos añadido además un método *getText()* que devuelve el texto del *TextField*. Será este al método al que llamemos para conseguir el texto introducido por el usuario después de que se cierre la ventana.

Para hacer la llamada a este diálogo y recoger los resultados, usaremos el siguiente código

```
DialogoModal dialogoModal = new DialogoModal((Frame) ventanaPadre);  
dialogoModal.pack(); // para darle tamaño automático a la ventana.  
dialogoModal.setVisible(true);
```

```
// Al ser modal, la llamada a setVisible(true) se queda bloqueada hasta  
// que el dialogo modal se oculte. Por ello, a continuación tenemos  
// la seguridad de que el texto ya esta disponible.  
System.out.println("El usuario ha escrito "+dialogoModal.getText());
```



Uso de Swing PRÁCTICA PROPUESTA

Ejercicio 2: Sumar dos números enteros y mostrar el resultado. (Uso de sentencias de ingreso, salida y cálculo matemático).

Ejercicio 3: Indicar si el número entero ingresado es par o impar. (Uso de sentencias de ingreso, salida, cálculo matemático y decisión lógica).

Ejercicio 4: Sumar diez números enteros y mostrar el resultado. (Uso de sentencias de ingreso, salida y reiteración).

Ejercicio 5: Ingresar un número entero e indicar si es primo o compuesto. (Uso de sentencias de ingreso, salida, decisión lógica y reiteración).

Ejercicio 6: Ingresar un número entero e indicar si es primo o compuesto y mostrar sus divisores. (Uso de sentencias de ingreso, salida, decisión lógica y reiteración).

Ejercicio 7: Se desea realizar una aplicación que sume los cuadrados de los treinta primeros números naturales, mostrando el resultado en pantalla.

Ejercicio 8: Escribir una aplicación que lea un número entero desde teclado y realiza la suma de los 100 número siguientes, mostrando el resultado en pantalla.

Ejercicio 9: Escriba una aplicación que lea tres números enteros positivos, y que calcule e imprima en pantalla el menor y el mayor de todos ellos.

Ejercicio 10: Generalizar el ejercicio 9 para n números enteros.

Ejercicio 11: Escriba una aplicación que lea temperaturas expresadas en grados Fahrenheit y las convierta a grados Celsius mostrándola. El programa finalizará cuando lea un valor de temperatura igual a 999. La conversión de grados Fahrenheit (F) a Celsius (C) está dada por $C = 5/9(F - 32)$



Uso de JTextArea

```
import javax.swing.JTextArea; (importer la clase)

JTextArea areaTextoSalida = new JTextArea(); (Inicializarla)
.....
areaTextoSalida.setText("....."); (Inicializar la primera línea del TextArea)
.....
areaTextoSalida.append("....");(Carga de líneas del TextArea)
.....
JOptionPane.showMessageDialog(null, areaTextoSalida, "Resultados"
,JOptionPane.INFORMATION_MESSAGE); (Muestra el TextArea dentro de un JOptionPane)
```

Observación de interés

Clases envoltorio de las variables primitivas o wrapper classes

Estas clases **pertenecen al paquete java.lang** y son las siguientes: **Byte, Short, Integer, Long, Float, Double, Boolean y Character**. Uno de los usos más habituales de las clases envoltorio es la utilización de métodos estáticos que permiten obtener a partir de una String que almacena un número en forma de texto, el correspondiente valor numérico almacenado en la variable primitiva que interese.

Clase	Método estático
Byte	byte parseByte(String num)
Short	short parseShort(String num)
Integer	int parseInt(String num)
Long	long parseLong(String num)
Float	float parseFloat(String num)
Double	double parseDouble(String num)

Ejercicio 12: Escriba una aplicación que a partir de la expresión algebraica de la ecuación de 1º grado indique un conjunto de 10 pares de coordenadas x,y para trazar la recta que representa.

Ejercicio 13: Escriba una aplicación que lea los coeficientes a, b y c de una ecuación de segundo, y estudie si tiene o no solución. En caso positivo, las soluciones se calcularán e imprimirán en pantalla.



Ejercicio 14: Escriba una aplicación que permita conocer el costo de una compra. La selección puede ser hecha entre 5 artículos diferentes. (Uso de while y switch)

Ejercicio 15: Se desea cargar un lote de n votantes por teclado, se conoce su sexo (M= masculino y F= femenino) y su posibilidad de elección (1, 2, 3 y 4). Se desea conocer:

- Cantidad de votantes masculinos y femeninos.
- Cantidad de votos por cada elección.
- Porcentaje con respecto al total de votos por cada selección.
- Cantidad de votos en blanco y su porcentaje.

Ejercicio 16: En una escuela se desea realizar una aplicación que permita conocer el Nombre del alumno, su curso y división, cantidad de faltas (justificadas, injustificadas, totales – suma de las anteriores), que además nos notifique:

- ALUMNO REGULAR (menos de 15 faltas totales).
- ALUMNO REINCORPORADO (más de 15 faltas totales, pero no más de un 30% de faltas injustificadas sobre las totales).
- ALUMNO LIBRE (más de 15 faltas totales, pero más de un 30% de faltas injustificadas sobre las totales).
- Los datos son ingresados uno a uno por teclado

EJERCICIO 17: Realizar una aplicación en la cual ingresados los tres lados de un triángulo me informe si el triángulo es ISÓSCELES, ESCALENO o EQUILÁTERO.

EJERCICIO 18: En un instituto dedicado a dictar cursos de Inglés se desea que se confeccione una aplicación que calcule los promedios de las calificaciones de n alumnos. Se conoce el nombre del alumno, 5 notas con 2 decimales. Mostrar el promedio en dos decimales. Los valores son ingresados uno a uno por teclado.

Ejercicio 19: Uso de etiquetas para los break . Sumar números enteros, si el número ingresado es 0 parar.

Ejercicio 20: Ingresar el valor del radio de una circunferencia y calcular el valor de su área. Realizar el ejercicio usando otro método en la clase

Matrices, arrays o vectores en java.

Java posee la capacidad de definir un conjunto de variables del mismo tipo agrupadas todas ellas bajo un mismo nombre, y distinguiéndolas mediante un índice numérico.

Para definir un array en java es como definir una variable o atributo, pero al especificar el tipo lo que hacemos es colocar un par de corchetes [] para indicar que lo que estamos definiendo es un array. Por ejemplo:

```
public int [] losValores;
```

en la que definimos un array de enteros llamado losValores



- Declarar y crear el array
- Inicializar los elementos del array y usarlos

1- Declarar y crear un array

Para declarar un array se escribe

```
tipo_de_dato[] nombre_del_array;
```

Para declarar un array de enteros escribimos

```
int[] numeros;
```

Para crear un array de 4 número enteros escribimos

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

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

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

2- Inicializar y usar los elementos del array

Para inicializar el array de 4 enteros escribimos

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

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

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

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

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

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



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

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

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

Java verifica que el índice no sea mayor o igual que la dimensión del array, lo que facilita mucho el trabajo al programador.

PRÁCTICA ANEXA - Ejercicios propuestos – No resueltos:

Ejercicio 21: Crear un vector de 5 elementos {"Andrea","Juan","Pedro","Azul", "Marcelo"}; cargado por la aplicación.

Ejercicio 22: Crear un vector de 10 elementos cargado por teclado

Ejercicio 23: Crear un vector de n elementos cargado por la aplicación

Ejercicio 24: Crear un vector con los elementos {7,5,6,8,1,2,3,4} y calcular el mayor y el menor valor

Ejercicio 25: Crear un vector con los elementos {7,5,6,8,1,2,3,4} , calcular el mayor, el menor valor y ordenar el vector de mayor a menor

Ejercicio 26: Crear una matriz de 2 por 2 con los elementos {7,5,6,8} , calcular la cantidad de filas y la cantidad de columnas.

Ejercicio 27: Cargar por teclado una matriz de n filas y m columnas. Mostrar el resultado por pantalla.

Ejercicio 28: Cargar por teclado una matriz de cuadrada de n filas y columnas. Mostrar el valor de la diagonal principal y el de la diagonal secundaria



Ejercicio 29: Cargar por teclado una matriz de 4 filas y 4 columnas. Mostrar el valor del producto de la diagonal principal por el de la diagonal secundaria

RESOLVER CON OBJETOS

Ejercicio 30: Modificar el Ejercicio 12 de manera que los pares de valores estén acomodados en un array. Texto ejercicio: Escriba una aplicación que a partir de la expresión algebraica de la ecuación de 1º grado indique un conjunto de 10 pares de coordenadas x,y para trazar la recta que representa.

Ejercicio 31: Modificar el Ejercicio 14 de manera que los pares de valores de los artículos estén acomodados en un vector. Texto ejercicio: Escriba una aplicación que permita conocer el costo de una compra. La selección puede ser hecha entre 5 artículos diferentes. (Uso de while y switch).

Ejercicio 32: Dado un array A de 100 posiciones, inicializado con números aleatorios, escriba un programa que sume 4 a todos aquellos elementos del array menores que 10 y 5 a los restantes. Imprima el contenido del array en pantalla.

Ejercicio 33: Escribir un programa que rellene un array con los números comprendidos entre 25 y 35 dividido 3.

Ejercicio 34: Escribir un programa que solicite al usuario 5 números, los almacene en un array y luego calcule la media aritmética de esos números.

Ejercicio 35: Escribir un programa que tras asignar los números, 23, 45, 68, 99, 10, 15 y 4 a un array, determine las posiciones del array en las que se encuentran el máximo y el mínimo valor. Imprimir en pantalla estos valores.

Ejercicio 36: Escribir un programa que tras asignar los números, -2, 5, 8, -9, 10, 15 y -4 a un array, calcule, independientemente, la suma de los elementos positivos y negativos. Imprimir en pantalla estos valores.

Ejercicio 37: Escribir un programa que determine la posición de la siguiente matriz en la que se encuentra el valor máximo:

25 33 67



12 91 43

67 8 76

Ejercicio 38: Escribir un programa que sume independientemente los valores positivos y negativos de la siguiente matriz:

-18 13 98

12 -67 30

34 -20 31

Ejercicio 39: Dada la siguiente serie, confeccionar una aplicación donde ingresada la cantidad de términos calcule su valor.

$$1 + 1/2! + 1/3! + 1/4! + 1/5! + 1/6! + \dots + 1/n!$$

Ejercicio 40: Un triángulo recto puede tener lados cuyas longitudes sean valores enteros. El conjunto de tres valores enteros para los lados de un triángulo recto se conoce como TRIPLE DE PITÁGORAS. Las longitudes de los tres lados deben satisfacer la relación $a^2=b^2+c^2$. Escribir una aplicación que permita calcular todos los triples de Pitágoras hasta el valor 500.

Ejercicio 41: En una empresa se desea confeccionar una aplicación Java que calcule el jornal de sus empleados. Esta aplicación debe calcular y presentar el resultado empleado por empleado en forma detallada.

EMPLEADO: XXXXXXXX, XXXXX	
CANTIDAD HORAS TRABAJADAS: XX	SUELDO BÁSICO: XXX.XX\$
CANTIDAD HORAS SIMPLES: XX	DESCUENTOS :
XX.XX\$	
CANTIDAD HORAS EXCEDENTES: XX	PREMIO : XXX.XX\$
	A COBRAR : XXX.XX\$

De los mismos se conoce su nombre y apellido y cantidad de horas trabajadas. El jornal se paga por semana con el siguiente detalle: Valor pagado por hora 25\$ (hasta 40 hs semanales), pasadas las 40hs el valor por cada hora excedente es de 30\$. Los descuentos aplicados son del 19,5 % sobre el valor calculado sobre las primeras 40 horas. Además si trabaja al menos 40 hs percibe una bonificación de 100\$ sin descuentos.



IMPORTANTE: La mayoría de las clases son porciones de código

JFrame es un contenedor donde colocar componentes. La clase JFrame se encuentra en el paquete javax.swing. La mayoría de las ventanas son instancias de la clase JFrame o subclases de JFrame.

JFrame proporciona los atributos y comportamientos básicos de una ventana.

```
/**
 * Crea una ventana simple con JFrame.
 */
import javax.swing.*;
public class Ventana
{
    public static void main( String[] args )
    {
        JFrame miVentana = new JFrame( "Mi Ventana" );
        miVentana.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE);
        miVentana.setSize( 300, 200 );
        miVentana.setVisible( true );
    } //fin del método main
} //fin de la clase ventana
```

```
/**
 * Crea una ventana simple con JFrame y le agrega unos botones.
 */
import javax.swing.*;
import java.awt.*;
public class Ventana02
{
    public static void main( String[] args )
    {
        JFrame miVentana = new JFrame();
        miVentana.setTitle( "Mi Ventana 02" );
        JButton btnUno = new JButton( "Uno" );
        JButton btnDos = new JButton( "Dos" );
        miVentana.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE);
        miVentana.setLayout( new FlowLayout() );
        miVentana.add( btnUno );
        miVentana.add( btnDos );
        miVentana.pack();
        miVentana.setVisible( true );
    } //fin del método main
} //fin de la clase ventan
```



```
/**
 * Ilustra cómo crear una ventana heredando de JFrame.
 *
 */
import java.awt.*;
import javax.swing.*;
public class Ventana03 extends JFrame
{
    public Ventana03( String titulo )
    {
        setTitle( titulo );
        inicializarComponentes();
        mostrarVentana();
    } //fin del constructor Ventana03

    public void inicializarComponentes() {
        setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE);
        JButton btnUno = new JButton( "Uno" );
        JButton btnDos = new JButton( "Dos" );
        setLayout( new FlowLayout() );
        add( btnUno );
        add( btnDos );
    } //fin del método inicializarComponentes

    public void mostrarVentana()
    {
        setSize( 300, 200 );
        setVisible( true );
    } //fin del método mostrarVentana
} //fin de la clase Ventana03

/**
 * Ilustra el funcionamiento de la clase Ventana03.
 */
public class TestVentana03
{
    public static void main( String[] args ) {
        Ventana03 uno = new Ventana03( "Ventana 01" );
        Ventana03 dos = new Ventana03( "Ventana 02" );
    } //fin del método void main
} //fin de la clase TestVentana03
```



Ejemplo de BorderLayout, BoxLayout y FlowLayout

Los Layout son las clases java que se encargan de decidir, dentro de un formulario, panel, ventana o lo que sea cómo van colocados los botones, áreas de texto y demás dentro de ellos. El Layout decide si los botones se pegan a la izquierda, a la derecha, si se deben hacer o no grandes al estirar la ventana, etc.

En este tutorial vamos a construir una ventana usando el BorderLayout. Para algunos de los paneles que meteremos en ella usaremos el BoxLayout y el FlowLayout.

La ventana será una ventana típica de cualquier aplicación, pero un poco exagerada. Es decir, una área central grande, en nuestro caso un JTextArea con scroll. Pondremos botones por los alrededores, arriba, abajo, a la izquierda y a la derecha. Por eso digo que es algo exagerado. Lo normal es ponerlos arriba y a un lado, no en todos los sitios a la vez.

Cada uno de los paneles que va en los laterales los coloreamos de un color distinto, de forma que podamos ver exactamente hasta donde llegan.

FlowLayout

Los botones que vamos a poner arriba y abajo del componente central van alineados en fila. Para hacer una fila horizontal de botones o cualesquiera otros componentes, el Layout ideal es el FlowLayout. Usarlo es bastante sencillo, basta decirle al panel que debe tener un FlowLayout y empezar a añadirle botones al panel.

Si el panel superior es una fila de cuatro botones, el código sería este

```
panelSuperior = new JPanel(new FlowLayout(FlowLayout.LEFT));  
panelSuperior.setBackground(Color.cyan);  
panelSuperior.add(new JButton("Sup.1"));  
panelSuperior.add(new JButton("Sup.2"));  
panelSuperior.add(new JButton("Sup.3"));  
panelSuperior.add(new JButton("Sup.4"));
```

Si en el constructor de FlowLayout no ponemos nada, los botones irán en fila horizontal centrados en el panel. Como los queremos pegados a la izquierda, en el constructor del



FlowLayout hemos puesto como parámetro FlowLayout.LEFT. Por supuesto, tenemos también FlowLayout.RIGHT. Si investigamos en la [API de FlowLayout](#), veremos también que hay constructores que nos permiten indicar qué márgenes queremos arriba, abajo, izquierda y derecha e incluso de separación entre botones.

El panel inferior lo haremos más variado, con un JLabel, un JTextField y un JButton. El código es similar

```
panelInferior = new JPanel(new FlowLayout());  
panelInferior.setBackground(Color.yellow);  
panelInferior.add(new JLabel("Pon texto"));  
panelInferior.add(new JTextField(25));  
panelInferior.add(new JButton("Listo"));
```

Esta vez hemos dejado el FlowLayout sin parámetros en el constructor, para que queden centrados.

BoxLayout

Para los componentes que van a los lados necesitamos algo similar al FlowLayout, pero que lo haga en vertical. El BoxLayout es algo más elaborado que el FlowLayout en el sentido de que permite tanto orientación horizontal como vertical. Sin embargo, aunque también se puede hacer, no es tan evidente el hacer que los componentes se peguen arriba, abajo, a la izquierda o a la derecha.

El código para nuestra barra de botones vertical en el lado izquierdo quedaría así

```
panelIzquierdo = new JPanel();  
panelIzquierdo.setLayout(new BoxLayout(panelIzquierdo, BoxLayout.Y_AXIS));  
panelIzquierdo.setBackground(Color.red);  
panelIzquierdo.add(new JButton("1"));  
panelIzquierdo.add(new JButton("2"));  
panelIzquierdo.add(new JButton("3"));
```

El constructor del BoxLayout es más complejo (y en mi opinión tonto) que el del FlowLayout. Debemos pasarle el contenedor al que lo estamos añadiendo, es decir, el parámetro panelIzquierdo. También debemos pasarle si queremos orientación vertical



BoxLayout.Y_AXIS u orientación horizontal BoxLayout.X_AXIS. Hay otras opciones, pero de momento las ignoramos. En la [API de BoxLayout](#) puedes ver más detalles sobre este Layout.

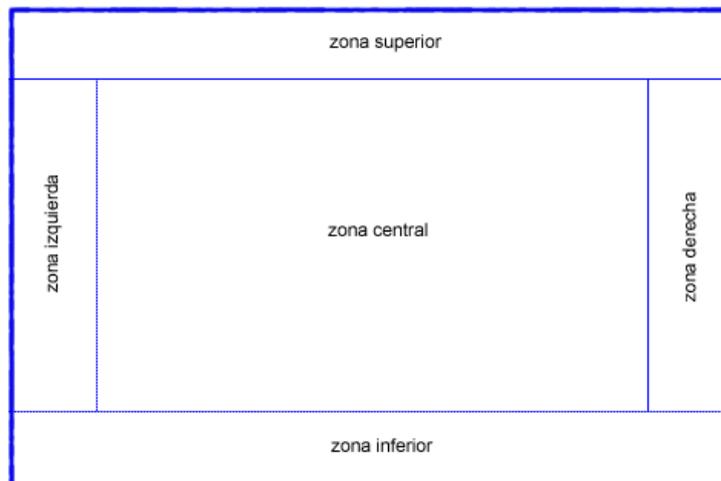
Para el panel derecho haremos lo mismo

```
panelDerecho = new JPanel();
panelDerecho.setBackground(Color.green);
panelDerecho.setLayout(new BoxLayout(panelDerecho, BoxLayout.Y_AXIS));
panelDerecho.add(new JButton("A"));
panelDerecho.add(new JButton("B"));
panelDerecho.add(new JButton("C"));
panelDerecho.add(new JButton("D"));
```

BorderLayout

Finalmente el Layout por defecto de todas las ventanas es el BorderLayout. Este es un Layout que divide el panel en cinco zonas: una central, una arriba, otra abajo, en la izquierda y en la derecha. El tamaño de estas zonas depende de los componentes que van dentro, pero más o menos es esto:

- ✓ La zona superior e inferior se estiran de izquierda a derecha hasta ocupar toda la ventana. En vertical se deja el tamaño justo para que quepan los componentes que hay dentro.
- ✓ Las zonas izquierda y derecha se estiran de arriba a abajo hasta ocupar todo el espacio disponible entre las zonas de arriba y abajo. El ancho para las zonas izquierda y derecha es el justo para que se vean los componentes que hay dentro.
- ✓ La zona central se estira en ambas direcciones hasta tocar a las otras zonas.



Por lo demás, este Layout es muy simple. Podemos añadir hasta cinco componentes, cada uno en su sitio.

```
// Creamos el componente central, un JTextArea con scroll.
```

```
JScrollPane scroll=new JScrollPane(new JTextArea());
```

```
// Añadimos todos los componentes
```

```
v.getContentPane().setLayout(new BorderLayout());
```

```
v.getContentPane().add(scroll,BorderLayout.CENTER);
```

```
v.getContentPane().add(panelSuperior,BorderLayout.NORTH);
```

```
v.getContentPane().add(panelInferior,BorderLayout.SOUTH);
```

```
v.getContentPane().add(panelIzquierdo,BorderLayout.WEST);
```

```
v.getContentPane().add(panelDerecho,BorderLayout.EAST);
```

Listo. `v` es nuestra ventana (en el ejemplo un `JFrame`) y con `BorderLayout.CENTER`, `NORTH`, `SOUTH`, `WEST` y `EAST` estamos indicando dónde queremos cada cosa.

Puedes ver este ejemplo funcionando en el Applet de arriba. Cada zona se ha coloreado de un color para distinguirla mejor. Los botones no hacen nada, así que no te aburras pulsándolos. Por supuesto, te puedes bajar el [BorderLayout.jar](#) para ejecutarlo con

```
$ java -jar BorderLayout.jar
```

o bajarte el fuente [PruebaBorderLayout.java](#) para verlo, modificarlo y jugar con él.



De todas formas, si esto te parece sencillo, puedes intentarlo con el GridBagLayout. Te permite hacer lo mismo de una forma más complicada, pero sin usar tantos paneles.

Ejercicio 42

Ejercitación de Validación de campos

Las expresiones regulares permiten verificar los ingresos en los campos de texto de manera que las cadenas cumplan con las condiciones que se establezcan:

- a)- Comiencen con mayúsculas
- b)- No tengan espacios en blanco
- c)- Solo admitan un tipo de carácter

Usaremos JFrame. La mayoría de las aplicaciones Swing presentan su GUI principal dentro de un JFrame - un contenedor Swing de alto nivel que proporciona ventanas para applets y aplicaciones. Un frame tiene decoraciones como un borde, un título, y botones para cerrar y minimizar la ventana. Un programa típico simplemente crea un frame, añade componentes al panel de contenido, y quizás añade una barra de menú.

La clase String proporciona varios métodos:

La clase matcher es la más simple y devuelve expresiones booleanas indicando si hay concordancia o no.

//Validacion.java

// Validación de la información del usuario mediante expresiones regulares.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Validacion extends JFrame {
    private JTextField telefonoCampoTexto, codigoPostalCampoTexto, estadoCampoTexto,
        ciudadCampoTexto, direccionCampoTexto, nombreCampoTexto, apellidoCampoTexto;

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



```

{
    Validacion aplicacion = new Validacion();
    aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE ); //Cierra Frame
}

public Validacion()
{
    super( "Validación" ); //Nombre del Frame

    // crear los componentes de GUI
    JLabel telefonoEtiqueta = new JLabel( "Teléfono" );
    JLabel codigoPostalEtiqueta = new JLabel( "C.P." );
    JLabel estadoEtiqueta = new JLabel( "Estado" );
    JLabel ciudadEtiqueta = new JLabel( "Ciudad" );
    JLabel direccionEtiqueta = new JLabel( "Dirección" );
    JLabel nombreEtiqueta = new JLabel( "Primer nombre" );
    JLabel apellidoEtiqueta = new JLabel( "Apellido paterno" );

    JButton aceptarBoton = new JButton( "Aceptar" );
    aceptarBoton.addActionListener(

        new ActionListener() { // clase interna

            public void actionPerformed( ActionEvent evento ) {
                validarFecha();
            }

        } // fin de la clase interna

    ); // fin de la llamada a addActionListener

    telefonoCampoTexto = new JTextField( 15 );
    codigoPostalCampoTexto = new JTextField( 5 );
    estadoCampoTexto = new JTextField( 2 );
    ciudadCampoTexto = new JTextField( 12 );
    direccionCampoTexto = new JTextField( 20 );
    nombreCampoTexto = new JTextField( 20 );
    apellidoCampoTexto = new JTextField( 20 );

```



```
JPanel primerNombre = new JPanel();
primerNombre.add( nombreEtiqueta );
primerNombre.add( nombreCampoTexto );
```

```
JPanel apellidoPaterno = new JPanel();
apellidoPaterno.add( apellidoEtiqueta );
apellidoPaterno.add( apellidoCampoTexto );
```

```
JPanel direccion1 = new JPanel();
direccion1.add( direccionEtiqueta );
direccion1.add( direccionCampoTexto );
```

```
JPanel direccion2 = new JPanel();
direccion2.add( ciudadEtiqueta );
direccion2.add( ciudadCampoTexto );
direccion2.add( estadoEtiqueta );
direccion2.add( estadoCampoTexto );
direccion2.add( codigoPostalEtiqueta );
direccion2.add( codigoPostalCampoTexto );
```

```
JPanel telefono = new JPanel();
telefono.add( telefonoEtiqueta );
telefono.add( telefonoCampoTexto );
```

```
JPanel aceptar = new JPanel();
aceptar.add( aceptarBoton );
```

// agregar los componentes a la aplicación

```
Container contenedor = getContentPane();
contenedor.setLayout( new GridLayout( 6, 1 ) ); // 6 filas y 1 columna
```

```
contenedor.add( primerNombre );
contenedor.add( apellidoPaterno );
contenedor.add( direccion1 );
contenedor.add( direccion2 );
contenedor.add( telefono );
contenedor.add( aceptar );
```



```
setSize( 400, 225 ); //tamaño del Frame
setVisible( true );

} // fin del constructor de Validacion

// maneja el evento de acción de aceptarBoton
private void validarFecha()
{
    // asegurar que no haya cuadros de texto vacíos
    if ( apellidoCampoTexto.getText().equals( "" ) ||
        nombreCampoTexto.getText().equals( "" ) ||
        direccionCampoTexto.getText().equals( "" ) ||
        ciudadCampoTexto.getText().equals( "" ) ||
        estadoCampoTexto.getText().equals( "" ) ||
        codigoPostalCampoTexto.getText().equals( "" ) ||
        telefonoCampoTexto.getText().equals( "" ) ) // fin de la condición

        JOptionPane.showMessageDialog( this, "Por favor llene todos los campos" );

    // si el formato de primer nombre es inválido, mostrar mensaje
    else if ( !nombreCampoTexto.getText().matches( "[A-Z][a-zA-Z]*" ) )
        JOptionPane.showMessageDialog( this, "Primer nombre inválido" );

    // si el formato de apellido paterno es inválido, mostrar mensaje
    else if ( !apellidoCampoTexto.getText().matches( "[A-Z][a-zA-Z]*" ) )
        JOptionPane.showMessageDialog( this, "Apellido inválido" );

    // si el formato de dirección es inválido, mostrar mensaje
    else if ( !direccionCampoTexto.getText().matches(
        "[a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+\\s+\\d+" ) )
        JOptionPane.showMessageDialog( this, "Dirección inválida" );

    // si el formato de ciudad es inválido, mostrar mensaje
    else if ( !ciudadCampoTexto.getText().matches(
        "[a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+" ) )
```



```
JOptionPane.showMessageDialog( this, "Ciudad inválida" );

// si el formato de estado es inválido, mostrar mensaje
else if ( !estadoCampoTexto.getText().matches(
    "[a-zA-Z]+|[a-zA-Z]+\s[a-zA-Z]+" ) )
    JOptionPane.showMessageDialog( this, "Estado inválido" );

// si el formato de código postal es inválido, mostrar mensaje
else if ( !codigoPostalCampoTexto.getText().matches( "\\d{4}" ) )
    JOptionPane.showMessageDialog( this, "Código postal inválido" );

// si el formato de teléfono es inválido, mostrar mensaje
else if ( !telefonoCampoTexto.getText().matches(
    "[0-9]\\d{10}" ) )
    JOptionPane.showMessageDialog( this, "Número telefónico inválido" );

else // la información es válida, avisar al usuario
    JOptionPane.showMessageDialog( this, "Gracias" );

} // fin del método Validacion

} // fin de la clase Validacion
```

Ejercicio 43

En este ejercicio se verá la forma de ingresar datos en los campos de texto y se manejarán los eventos presionando Enter.

// TextFieldTest.java

// Ejemplo de demostración de la clase JTextField.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```



```
public class TextFieldTest extends JFrame {
    private JTextField textField1, textField2, textField3;
    private JPasswordField passwordField;

    public static void main( String args[] )
    {
        TextFieldTest application = new TextFieldTest();
        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    }
}
```

// Configurando la GUI

```
public TextFieldTest()
{
    super( "Funcionamiento de la clase JTextField " );

    Container container = getContentPane();
    container.setLayout( new FlowLayout() );

    // construir un Campo de texto con el tamaño por defecto
    textField1 = new JTextField( 10 );
    container.add( textField1 );

    // construir un Campo de texto con texto por defecto

    textField2 = new JTextField( "Entre un texto aquí" );
    container.add( textField2 );

    // construir un Campo de texto con texto por defecto con 20 elementos visibles no editables
    textField3 = new JTextField( "Texto visible, no editable", 20 );
    textField3.setEditable( false );
    container.add( textField3 );

    // construir un Campo de password con texto por defecto
    passwordField = new JPasswordField( "Texto no visible" );
    container.add( passwordField );

    // Registrar eventos a manipular
    TextFieldHandler handler = new TextFieldHandler();
}
```



```
textField1.addActionListener( handler );
textField2.addActionListener( handler );
textField3.addActionListener( handler );
passwordField.addActionListener( handler );

setSize( 325, 100 );
setVisible( true );

}

// Clase privada manejadora de eventos
private class TextFieldHandler implements ActionListener {

    // Procesamiento de eventos actionPerformed
    public void actionPerformed((ActionEvent event) )
    {
        String string = "";

        // El usuario presiona Enter en JTextField Campo de Texto 1
        if ( event.getSource() == textField1 )
            string = "Campo de texto 1: " + event.getActionCommand();

        // El usuario presiona Enter en JTextField Campo de Texto 2
        else if ( event.getSource() == textField2 )
            string = "Campo de texto 2: " + event.getActionCommand();

        // El usuario presiona Enter en JTextField Campo de Texto 3
        else if ( event.getSource() == textField3 )
            string = " Campo de texto 3: " + event.getActionCommand();

        // El usuario presiona Enter en Campo de Password
        else if ( event.getSource() == passwordField ) {
            string = "passwordField: " +
                new String( passwordField.getPassword() );
        }

        JOptionPane.showMessageDialog( null, string );
    }
}
```



```
    } // Fin del método actionPerformed  
  
}  
  
}
```

Ejercicio 44

En este ejercicio se verá la forma de colocar botones y su manejo. (se necesitan dos imágenes)

// ButtonTest.java

//Creación de JButtons.

```
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;  
  
public class ButtonTest extends JFrame {  
    private JButton plainButton, fancyButton;  
  
    public static void main( String args[] )  
    {  
        ButtonTest application = new ButtonTest();  
        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );  
    }  
}
```

// Configurar GUI

```
public ButtonTest()  
{  
    super( "Testeo de botones" );  
    Container container = getContentPane();  
    container.setLayout( new FlowLayout() );  
}
```

**// Crea botones**

```
plainButton = new JButton( "Botón Plano" );
container.add( plainButton );

Icon bug1 = new ImageIcon( "bug1.gif" );
Icon bug2 = new ImageIcon( "bug2.gif" );
fancyButton = new JButton( "Botón con Imagen", bug1 );
fancyButton.setRolloverIcon( bug2 );
container.add( fancyButton );
```

// Crear una instancia de la clase ButtonHandler para manejar los eventos de los botones

```
ButtonHandler handler = new ButtonHandler();
fancyButton.addActionListener( handler );
plainButton.addActionListener( handler );

setSize( 275, 100 );
setVisible( true );

}
```

// Clase privada para manejo de botones

```
private class ButtonHandler implements ActionListener {

    // manejo de eventos de botones
    public void actionPerformed((ActionEvent event)
    {
        JOptionPane.showMessageDialog( ButtonTest.this,
            "Usted presionó: " + event.getActionCommand() );
    }

}

}
```



Ejercicio 45

En este ejercicio se verá la forma de trabajar con botones de chequeo

```
// CheckBoxTest.java
```

```
// Creabotones JCheckBox
```

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

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

```
import javax.swing.*;
```

```
public class CheckBoxTest extends JFrame {
```

```
    private JTextField field;
```

```
    private JCheckBox bold, italic;
```

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

```
{
```

```
    CheckBoxTest application = new CheckBoxTest();
```

```
    application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
```

```
}
```

```
// configurar GUI
```

```
public CheckBoxTest()
```

```
{
```

```
    super( " Uso de JCheckBox" );
```

```
    Container container = getContentPane();
```

```
    container.setLayout( new FlowLayout() );
```

```
// Configurar JTextField y configurar fuentes
```

```
field = new JTextField( "Mirame como cambio", 20 );
```

```
field.setFont( new Font( "Serif", Font.PLAIN, 14 ) );
```

```
container.add( field );
```

```
// Crear objetos checkbox
```



```
bold = new JCheckBox( "Negrita" );  
container.add( bold );
```

```
italic = new JCheckBox( "Cursiva" );  
container.add( italic );
```

// Registrar listeners para JCheckBoxes

```
CheckBoxHandler handler = new CheckBoxHandler();  
bold.addItemListener( handler );  
italic.addItemListener( handler );
```

```
setSize( 275, 100 );  
setVisible( true );
```

```
}
```

// Clase privada para ItemListener manejo de eventos

```
private class CheckBoxHandler implements ItemListener {  
    private int valBold = Font.PLAIN;  
    private int valItalic = Font.PLAIN;
```

// Respuestas a los eventos registrados

```
public void itemStateChanged( ItemEvent event )  
{
```

// Evento chequeo negrita

```
    if ( event.getSource() == bold )  
        valBold = bold.isSelected() ? Font.BOLD : Font.PLAIN;
```

//Evento proceso cursiva

```
    if ( event.getSource() == italic )  
        valItalic = italic.isSelected() ? Font.ITALIC : Font.PLAIN;
```

// Configuración de la fuente del campo

```
    field.setFont( new Font( "Serif", valBold + valItalic, 14 ) );
```

```
}
```



```
}  
}
```

Ejercicio 46

En este ejercicio se verá la forma de trabajar con botones de radio.

//RadioButtonTest.java

// Crear botones de radio usando ButtonGroup y JRadioButton.

```
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;  
  
public class RadioButtonTest extends JFrame {  
    private JTextField field;  
    private Font plainFont, boldFont, italicFont, boldItalicFont;  
    private JRadioButton plainButton, boldButton, italicButton, boldItalicButton;  
    private ButtonGroup radioGroup;  
  
    public static void main( String args[] )  
    {  
        RadioButtonTest application = new RadioButtonTest();  
        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );  
    }  
  
    // Crear GUI y fuentes  
    public RadioButtonTest()  
    {  
        super( "RadioButton" );  
  
        Container container = getContentPane();  
        container.setLayout( new FlowLayout() );
```



// Configurar JTextField

```
field = new JTextField( "Mirá como cambia el estilo", 25 );  
container.add( field );
```

// Crear botones de radio

```
plainButton = new JRadioButton( "Normal", true );  
container.add( plainButton );
```

```
boldButton = new JRadioButton( "Negrita", false );  
container.add( boldButton );
```

```
italicButton = new JRadioButton( "Cursiva", false );  
container.add( italicButton );
```

```
boldItalicButton = new JRadioButton( "Negrita/Cursiva", false );  
container.add( boldItalicButton );
```

// Crear relaciones lógicas entre JRadioButtons

```
radioGroup = new ButtonGroup();  
radioGroup.add( plainButton );  
radioGroup.add( boldButton );  
radioGroup.add( italicButton );  
radioGroup.add( boldItalicButton );
```

// Crear objetos de fuentes

```
plainFont = new Font( "Serif", Font.PLAIN, 14 );  
boldFont = new Font( "Serif", Font.BOLD, 14 );  
italicFont = new Font( "Serif", Font.ITALIC, 14 );  
boldItalicFont = new Font( "Serif", Font.BOLD + Font.ITALIC, 14 );  
field.setFont( plainFont ); // set initial font
```

// Registra eventos para JRadioButtons

```
plainButton.addItemListener( new RadioButtonHandler( plainFont ) );  
boldButton.addItemListener( new RadioButtonHandler( boldFont ) );  
italicButton.addItemListener(  
    new RadioButtonHandler( italicFont ) );  
boldItalicButton.addItemListener(  
    new RadioButtonHandler( boldItalicFont ) );
```



```
setSize( 300, 100 );
setVisible( true );

}

// Clase privada para escucha de eventos
private class RadioButtonHandler implements ItemListener {
    private Font font;

    public RadioButtonHandler( Font f )
    {
        font = f;
    }

// Evento Botón
    public void itemStateChanged( ItemEvent event )
    {
        field.setFont( font );
    }
}
}
```

Ejercicio 47

Uso de Combo de texto con tres imágenes

```
// ComboBoxTest.java
// Usar un JComboBox para seleccionar una imagen para mostrar.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```



```

public class ComboBoxTest extends JFrame {
    private JComboBox imagesComboBox;
    private JLabel label;

    private String names[] =
        { "bug1.gif", "bug2.gif", "travelbug.gif", "buganim.gif" };
    private Icon icons[] = { new ImageIcon( names[ 0 ] ),
        new ImageIcon( names[ 1 ] ), new ImageIcon( names[ 2 ] ),
        new ImageIcon( names[ 3 ] ) };

    public static void main( String args[] )
    {
        ComboBoxTest application = new ComboBoxTest();
        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    }
// Configurar GUI
    public ComboBoxTest()
    {
        super( "JComboBox" );

        Container container = getContentPane();
        container.setLayout( new FlowLayout() );

// Configurar JComboBox y registrar sus eventos
        imagesComboBox = new JComboBox( names );
        imagesComboBox.setMaximumRowCount( 3 );
        imagesComboBox.addItemListener(

            new ItemListener() {

// Manejador de evento del JComboBox
                public void itemStateChanged( ItemEvent event )
                {
// Determinar cuál es la selección
                    if ( event.getStateChange() == ItemEvent.SELECTED )
                        label.setIcon( icons[
                            imagesComboBox.getSelectedIndex() ] );
                }
            }
        );
    }
}

```



```
    }  
  
    }  
);  
container.add( imagesComboBox );  
  
// Configuración JLabel para mostrar las imágenes  
label = new JLabel( icons[ 0 ] );  
container.add( label );  
  
setSize( 350, 100 );  
setVisible( true );  
  
}  
  
}
```

Ejercicio 48

Ejercicio que mediante una lista desplegable puedo seleccionar un ítem y realizar una acción.

//ListTest.java

// Seleccionar colores desde una JList.

```
import java.awt.*;  
import javax.swing.*;  
import javax.swing.event.*;  
  
public class ListTest extends JFrame {  
    private JList colorList;  
    private Container container;  
  
    private final String colorNames[] = { "Negro", "Azul", "Celeste",  
        "Gris Oscuro", "Gris", "Verde", "Gris Claro", "Violeta",
```



```
"Naranja", "Rosa", "Rojo", "Blanco", "Amarillo" );
```

```
private final Color colors[] = { Color.BLACK, Color.BLUE, Color.CYAN,  
    Color.DARK_GRAY, Color.GRAY, Color.GREEN, Color.LIGHT_GRAY,  
    Color.MAGENTA, Color.ORANGE, Color.PINK, Color.RED, Color.WHITE,  
    Color.YELLOW };
```

```
public static void main( String args[] )  
{  
    ListTest application = new ListTest();  
    application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );  
}
```

// Configurar GUI

```
public ListTest()  
{  
    super( "Lista de texto" );  
  
    container = getContentPane();  
    container.setLayout( new FlowLayout() );
```

// Crear un array con los valores de la lista de colores

```
colorList = new JList( colorNames );  
colorList.setVisibleRowCount( 5 );
```

// No permitir selecciones múltiples

```
colorList.setSelectionMode( ListSelectionModel.SINGLE_SELECTION );
```

// Adicionar un JScrollPane conteniendo JList

```
container.add( new JScrollPane( colorList ) );  
colorList.addListSelectionListener(
```

```
    new ListSelectionListener() {
```

// Selección de eventos

```
    public void valueChanged( ListSelectionEvent event )  
    {  
        container.setBackground(
```



```
        colors[ colorList.getSelectedIndex() ] );
    }

}

);
setSize( 350, 150 );
setVisible( true );

}

}
```

Ejercicio 49

Ejercicio que muestra el funcionamiento de un panel de selección múltiple

// MultipleSelectionTest.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MultipleSelectionTest extends JFrame {
    private JList colorList, copyList;
    private JButton copyButton;
    private final String colorNames[] = { "Negro", "Azul", "Celeste",
        "Gris Oscuro", "Gris", "Verde", "Gris Claro", "Violeta",
        "Naranja", "Rosa", "Rojo", "Blanco", "Amarillo" };
    public static void main( String args[] )
    {
        MultipleSelectionTest application = new MultipleSelectionTest();
        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    }
}
```

**// Configurar GUI**

```
public MultipleSelectionTest()
{
    super( "Lista de selección múltiple" );

    Container container = getContentPane();
    container.setLayout( new FlowLayout() );
```

// Configurar JList

```
colorList = new JList( colorNames );
colorList.setVisibleRowCount( 5 );
colorList.setSelectionMode(
    ListSelectionMode.MULTIPLE_INTERVAL_SELECTION );
container.add( new JScrollPane( colorList ) );
```

// Crear el botón copiar y registrar

```
copyButton = new JButton( "Copiar >>>" );
copyButton.addActionListener(

    new ActionListener() { // anonymous inner class
```

// Manejador de evento

```
    public void actionPerformed((ActionEvent event)
    {
        // Colocar valores seleccionados en copyList
        copyList.setListData( colorList.getSelectedValues() );
    }
    );
container.add( copyButton );
```

// Configurar JList

```
copyList = new JList( );
copyList.setVisibleRowCount( 5 );
copyList.setFixedCellWidth( 100 );
copyList.setFixedCellHeight( 15 );
copyList.setSelectionMode(
    ListSelectionMode.SINGLE_INTERVAL_SELECTION );
```



```
container.add( new JScrollPane( copyList ) );

setSize( 300, 130 );
setVisible( true );
}
}
```

Ejercicio 50 (Propuesto)

Utilizando las herramientas visualizadas en los ejercicios anteriores confeccionar una aplicación Java que permita ingresando un usuario y una contraseña:

- a- Si es correcta, una página de Bienvenida
- b- Si no es correcta, a una página de suscripción con datos personales. En ella deben usarse los objetos vistos.

Los datos no deben almacenarse. El usuario es JAVA y el Pass: ENTRAR