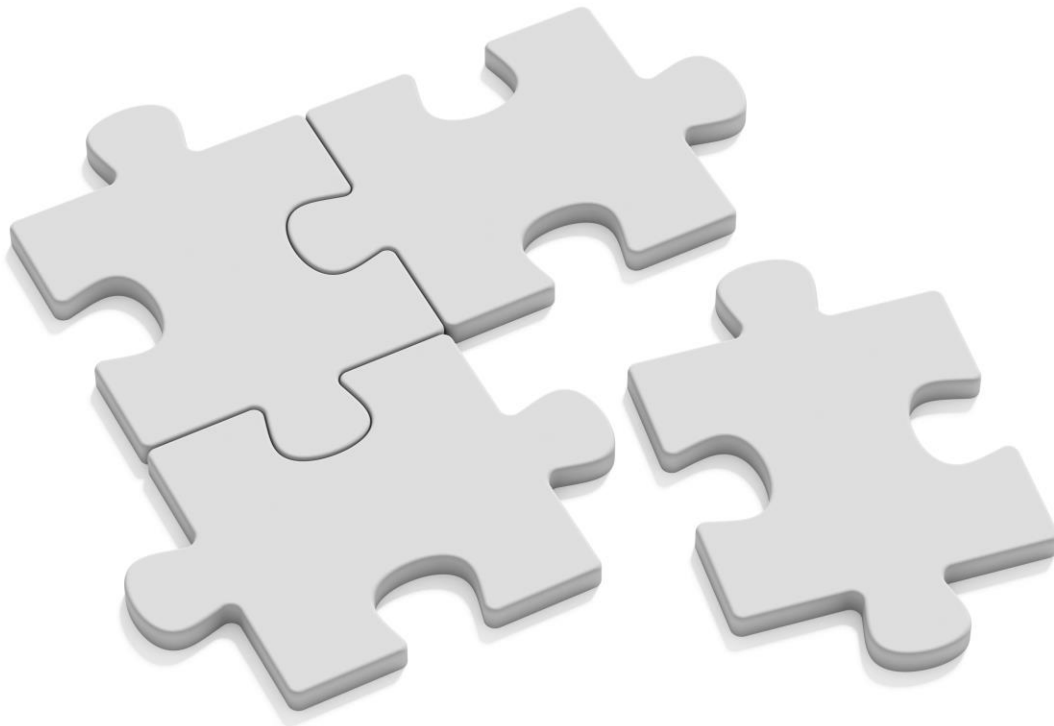


2017

CÁTEDRA DE LENGUAJE DE PROGRAMACIÓN JAVA

Ings. Mario Bressano & Miguel Iwanow

ENVÍO 01/2017

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 *Object*, *Thread*, *Math*...
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 incluídos 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 interfases, debe contener la implementación de todos los métodos (con las mismas listas de parámetros) de dichas interfases.

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 interfases pueden, asimismo, descender de otras interfases 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 interfases, 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 interface; 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:

Acceso desde:	**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 sobreescrito 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 atentaría 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 podriamos 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 construtor 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 promt o ventana de datos donde se puede pedir o desplegar información.

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

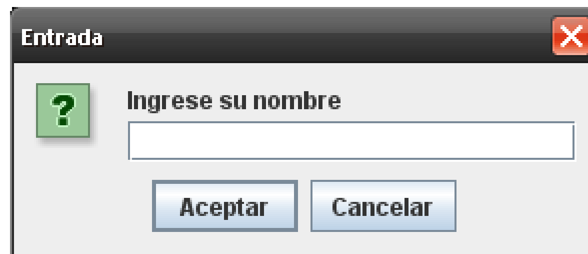
import javax.swing.JOptionPane;

———

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

Codigo:

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

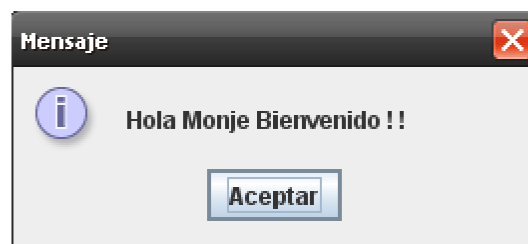


——————————————————————————————————————

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");
```

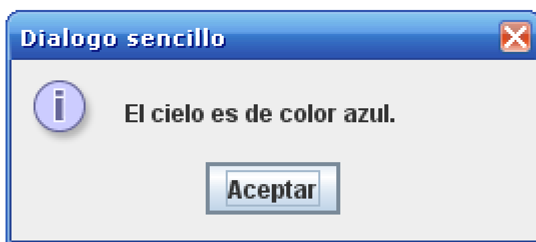
——————————————————————————————————————

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:

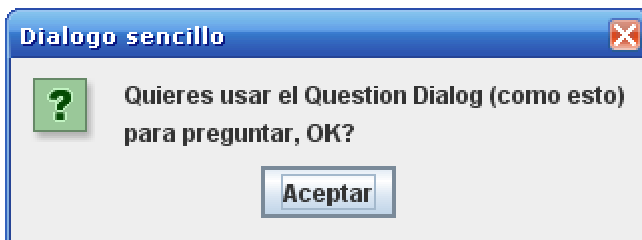
Codigo:

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



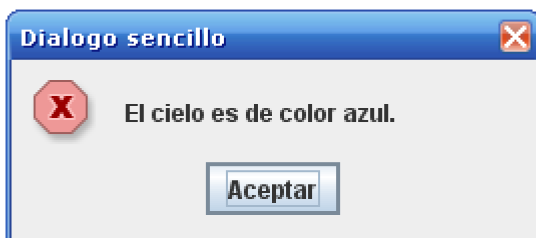
Codigo:

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



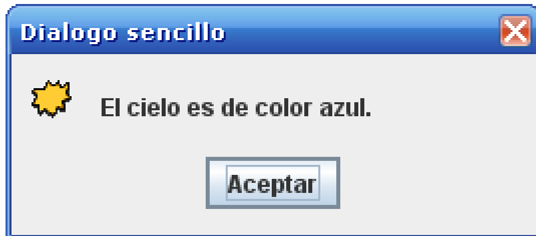
Código:

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 titulo que lleva el diálogo en la barra de tareas
- **int optionType**: 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 visualizará sobre el botón y no se podrá ir detrás del mismo si hacemos click en otro sitio.

- **message**: El mensaje a mostrar, habitualmente un *String*, aunque vale cualquier *Object* cuyo método *toString()* devuelva algo con sentido.
- **title**: El título para la ventana.
- **optionType**: Un entero indicando qué 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 qué tipo de mensaje estamos mostrando. Este tipo servirá para que se determine qué 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*, saldrá el icono adecuado según el parámetro *messageType*.
- **options**: Un array de *objects* que determinan las posibles opciones. Si los objetos son componentes visuales, aparecerán tal cual como opciones. Si son *String*, el *JOptionPane* pondrá tantos botones como *String*. Si son cualquier otra cosa, se les tratará como *String* llamando al método *toString()*. Si se pasa *null*, saldrán los botones por defecto que se hayan indicado en *optionType*.
- **initialValue**: Selección por defecto. Debe ser uno de los *Object* que hayamos pasado en el parámetro *options*. Se puede pasar *null*.

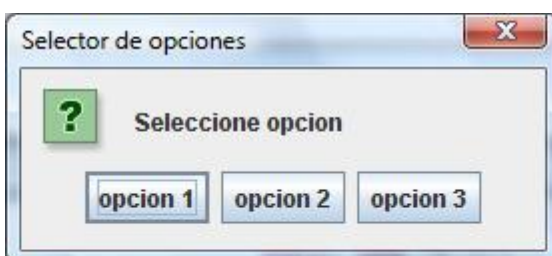
La llamada a *JOptionPane.showOptionDialog()* devuelve un entero que representa la opción que ha seleccionado el usuario. La primera de las opciones del array es la posición cero. Si se cierra la ventana con la cruz de la esquina superior derecha, el método devolverá -1.

Aquí un ejemplo de cómo llamar a este método

```
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 código 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.

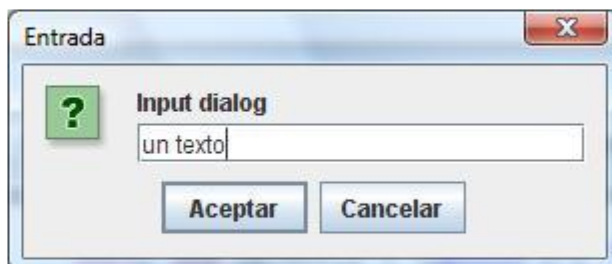
Si usamos los métodos que no tienen array de opciones, la ventana mostrará una caja de texto para que el usuario escriba la opción que desee (un texto libre). Si usamos un método que tenga un array de opciones, entonces aparecerá en la ventana un *JComboBox* en vez de una caja de texto, donde estarán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

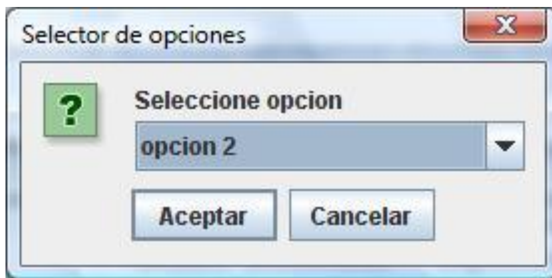


En este segundo ejemplo, damos todas las opciones que queremos, obteniendo un *JComboBox*

```
// 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 "+seleccion);
```

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()

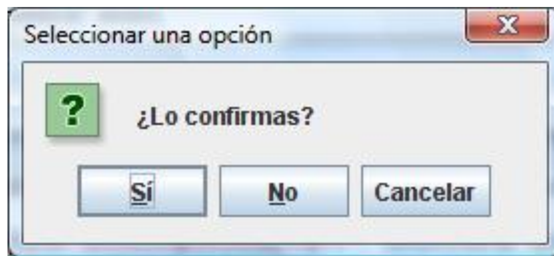
Este método muestra una ventana pidiendo una confirmación al usuario, estilo "*¿Seguro que lo quieres borrar todo?*" y da al usuario opción de aceptar o cancelar ese borrado masivo que está a punto de hacer. El método devuelve un entero indicando la respuesta del usuario. Los valores de ese entero puede 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 creemos 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 códígo 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, despues de dialogo.setVisible(), podemos obtener la opcion seleccionada con

Object valor = option.getValue();

Este valor será un JOption.UNINITIALIZED_VALUE si el dialogo 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 *JTextField* dentro de él y al pulsar *<intro>* en ese *JTextField*, después de haber escrito algo, el *JDialog* se cerrará. El código para conseguir esto es el siguiente:

```
pakage com.chuidiang.ejemplos.option_pane_dialog_modal;

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.
 *
 * @author Chuidiang
 *
 */
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("Mete un dato");
      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);
         }
      });
   }

   /**
    * Deveulve 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 *JTextField*. 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 Farenheit (F) a Celsius (C) está dada por C = 5/9(F – 32)

REVISIÓN MARZO 2017