

INTRODUCCIÓN A LA PROGRAMACIÓN

Semana 7

ING. MARTIN POLIOTTO

Docente a cargo del módulo

Septiembre 2020



DIPLOMATURA EN

**NUEVAS
TECNOLOGÍAS**

SECRETARÍA DE
EXTENSIÓN
UNIVERSITARIA
1974 - 1982

SEU

UTN
Facultad Regional Córdoba

Ministerio de
PROMOCIÓN DEL EMPLEO
Y DE LA ECONOMÍA FAMILIAR

Ministerio de
CIENCIA Y
TECNOLOGÍA

CBA
ENTRE TODOS

Oficina de
Relaciones con la
Comunidad
y el Medio Ambiente

Comisión de
Oficina de Montevideo

Semana 07

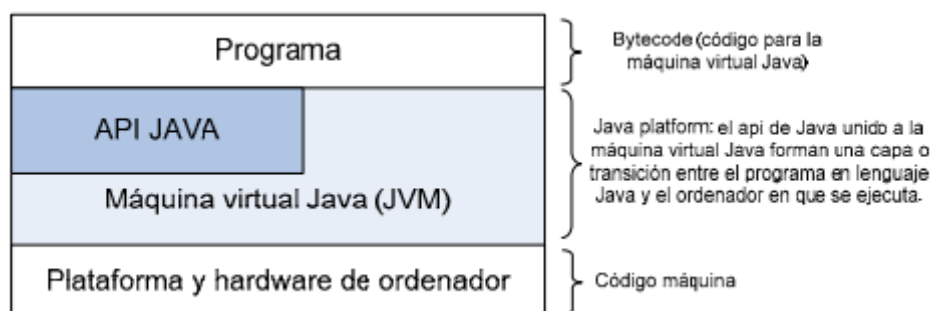
1. Api de Java

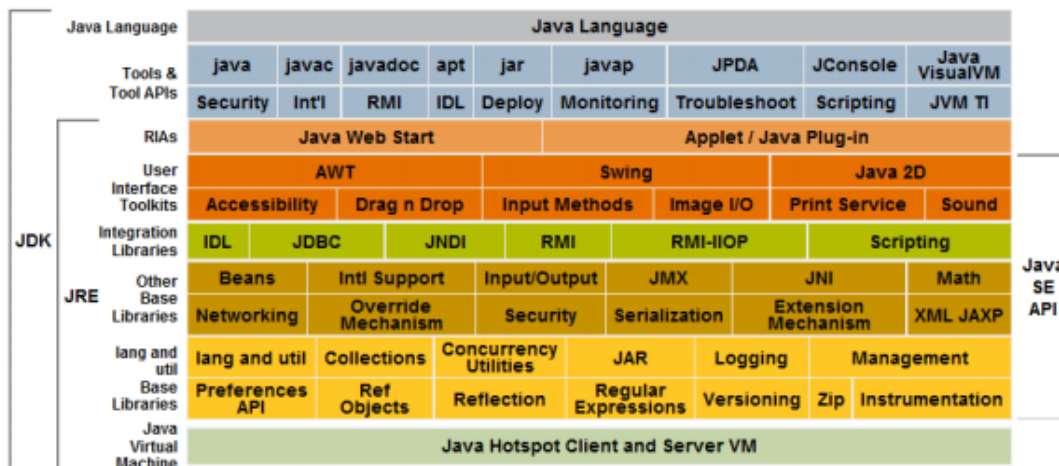
1.1 Qué es la Api de Java

Hasta ahora hemos visto ejemplos donde utilizábamos la clase System (por ejemplo en la invocación `System.out.println`) o la clase String (con la construimos *objetos cadenas*). Un pregunta interesante sería: ¿Dónde están dichas clases?

La respuesta está en que al instalar Java (el paquete JDK) en nuestro ordenador, además del compilador y la máquina virtual de Java se instalan bastantes más elementos. Entre ellos, una cantidad muy importante de clases que ofrece Java y que están a disposición de todos los programadores listas para ser usadas. Estas clases junto a otros elementos forman lo que se denomina API (Application Programming Interface) de Java.

Los siguientes esquemas, parte de la documentación de Java, nos dan una idea de cómo funciona el sistema Java y de qué se instala cuando instalamos Java (el paquete JDK):





Notar que cuando instalamos Java, instalamos múltiples herramientas, entre ellas una serie de **librerías (paquetes de clases)** a cuyo conjunto solemos referirnos como “biblioteca estándar de Java”. Las librerías contienen código Java listo para ser usado por nosotros.

1.2 Paquetes de clases

Los paquetes en Java (**packages**) son la forma en la que Java nos permite agrupar de alguna manera lógica los componentes de nuestra aplicación que estén relacionados entre sí.

Los paquetes permiten:

- poner en su interior casi cualquier cosa como: clases, interfaces, archivos de texto, entre otros. De este modo, los paquetes en Java ayudan a darle una buena organización a la aplicación ya que permiten modularizar o categorizar las diferentes estructuras que componen nuestro software

- brindar un nivel adicional de seguridad para nuestras clases, métodos o interfaces, pues permiten especificar si una clase o interfaz en particular es accesible por todos los componentes del software (sin importar el paquete) o si en realidad es solo accesible por las clases que estén en el mismo paquete que ésta.

En Java podemos crear paquetes mediante la sentencia **package**, tal como se muestra a continuación:

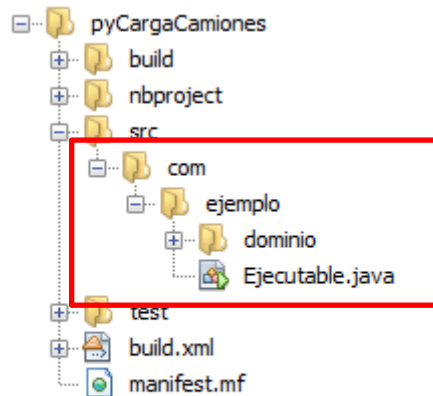
package ruta.del.paquete

Si bien la sintaxis es muy sencilla, podemos algunos detalles a tener en cuenta:

- La declaración del paquete debe estar al principio del archivo Java, es decir, es la primera línea que se debe ver en nuestro código o archivo .java
- Es posible definir una ruta compleja utilizando el operador punto (.). Tener presente que un paquete físicamente es el equivalente a una carpeta de nuestro sistema de directorios.

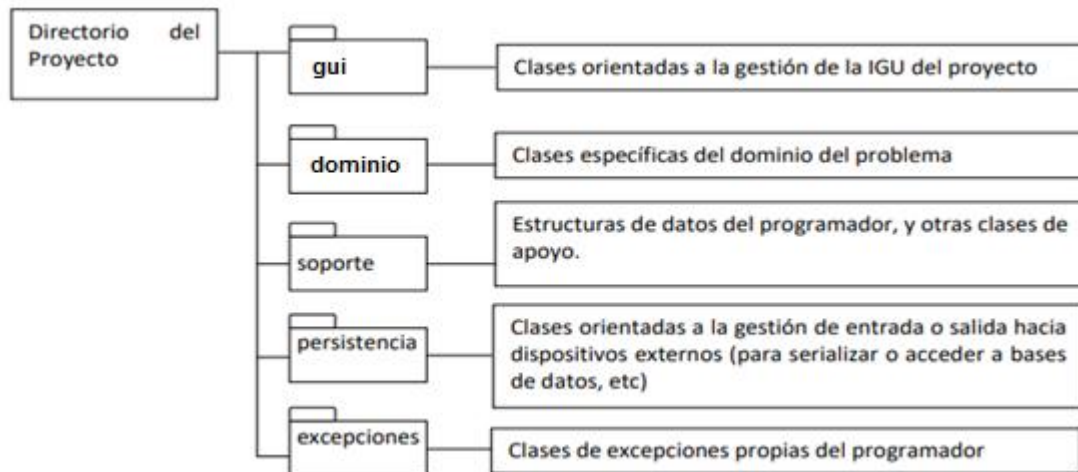
- Por ejemplo:

package *com.ejemplo.dominio*



- El nombre del paquete se define de manera inversa al dominio de la organización o grupo. Por ejemplo, dominioempresa.com puede ser usado como nombre de paquete así: [com.dominio_empresa.utilidades](#).
- El nombre del paquete debería definirse en minúscula. Si existen varias palabras en el nombre, se pueden separar con guion bajo (_).
- Si decidimos no declarar un paquete para nuestra clase, ésta quedará en un paquete que se conoce como paquete por defecto (**default package**).
- El uso de *packages* permite solucionar potenciales problemas en cuanto a nombres repetidos. Es posible tener dos clases con el mismo identificador pero en paquetes diferentes.
- Todas las clases nativas de Java vienen en *packages* predefinidos, y todos esos *packages* a su vez vienen comprimidos y distribuidos en un archivo llamado *rt.jar* (ubicado en la carpeta <jdk instalado>\jre\lib o similar, donde se instaló el JDK en su computadora).

- Para que una clase pueda acceder a otras clases que se encuentran en un package distinto, se usa la instrucción **import** al comienzo del archivo de la clase.
- Una configuración de paquetes sugerida para nuestros proyectos es la que se muestra en el siguiente esquema:



1.3 Clases de Envoltorio

El lenguaje Java provee una serie de clases que permiten representar valores de tipos primitivos como objetos. Esas clases se implementan definiendo simplemente un atributo del tipo al que se desea representar, y dotando a la clase de métodos para manejar el valor almacenado. Dichas clases reciben el nombre de **wrappers o emboltorios**, y existe una por cada tipo primitivo:

Tipo primitivo	Wrapper class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

- Todas éstas clases están marcadas ellas mismas como **final** (no pueden ser derivadas) y sus atributos también son final (el valor de los atributos no puede modificarse una vez creado el objeto).

Los usos más comunes de estas clases suelen ser la convertir cadenas en tipos primitivos y viceversa. Por ejemplo:

- 1) Crear un objeto entero a partir de un número 10:

```
Integer entero = new Integer(10);
```

```
System.out.println("Entero: " + entero.intValue());
```

- La primer línea es equivalente a escribir directamente:

```
Integer entero = 10;
```

Esta conversión
implícita se llama **auto-
boxing**

- 2) Convertir una cadena con el valor 10 a un entero:

```
String valor = "10";
```

```
int val = Integer.parseInt(valor);
```

1.4 La clase String

Las cadenas de caracteres en Java se representan como valores de un tipo especial de datos llamado String. En principio, manejar un String es simple, y el procedimiento no difiere mucho de la forma de manejar primitivos: se declaran variables String, y se asignan valores en ellas en forma normal mediante el operador de asignación.

Algunas operaciones más frecuentes con objetos String:

1) **Comparar** cadenas:

El método **compareTo()** trabaja con dos Strings, los compara lexicográficamente (a nivel de valor ascii letra por letra), y retorna un valor de tipo int, que indica el resultado de la comparación. Supongamos que queremos comparar dos cadenas guardadas en dos variables. La forma de hacerlo sería:

```
String cad1 = "Hola Java";

String cad2 = "hola Java";

int rtdo = cad1.compareTo(cad2);

if(rtdo == 0){

    System.out.println("Cadenas iguales!");

}else if(rtdo > 0){

    System.out.println(cad1 + " es mayor que " + cad2 + "!");

}else{

    System.out.println(cad1 + " es menor que " + cad2 + "!");

}
```



```
}
```

El método **comparteTo()** devuelve entonces:

- ✓ 0 : cadenas **iguales**
- ✓ > 0 cad1 **mayor que** cad2
- ✓ < 0 cad1 **menor que** cad2

Otra método que también podemos usar para comparar cadenas es **equals()**. Este método es heredado de Object y redefinido en la propia clase String. A diferencia de compareTo() este método solo devuelve un valor boolean indicando si las cadenas son o no iguales.

2) **Concatenar** cadenas:

La clase **String** está marcada **final**, con lo cual no puede ser derivada. Pero un detalle interesante es que sus atributos principales también están marcados **final**, por lo cual son constantes: una vez que se asigna una cadena a un objeto String, ese valor no puede ser modificado (por eso decimos que los strings **son inmutables**).

El lenguaje Java hace esto por razones de eficiencia: gestionará ese String de forma de hacer más rápido su acceso y menos costoso su mantenimiento en memoria. Al momento de crear una cadena si ya existía con anterioridad, entonces devolverá la referencia a dicho objeto sin crear uno nuevo.

El problema de eficiencia generalmente se presenta *al concatenar* cadenas. Por ejemplo supongamos que necesitamos armar un listado con los objetos almacenados en un arreglo de tipo *Carga* (ejemplo utilizado en el material de la semana anterior)

```
String aux = "";  
  
for(int i = 0; i < arreglo.length; i++){  
  
    aux += arreglo[i].toString();  
  
}  
  
System.out.print("Listado de cargas:\n" + aux);
```

- ✓ Notar que cada vez que se ejecuta la línea `aux += arreglo[i].toString()` se está reasignando en la variable un objeto cadena diferente y queda un objeto anterior sin referenciar. Esto provoca que el recolector de basura (garbage collector) elimine el viejo objeto.

Para evitar los problemas de eficiencia que genera el estatus final de los atributos de la clase `String`, existe también la clase **StringBuilder** que permite definir objetos que representan cadenas cuyos contenidos pueden modificarse sin tener que cambiar las referencias, en base a métodos que acceden al contenido y pueden agregar caracteres, concatenar, etc. El tamaño de un `StringBuilder` se va ajustando a las necesidades de la cadena que se está almacenando. La clase `StringBuilder` implementa el método `toString()` de forma que al invocarlo, se retorna un objeto de la clase `String` con la cadena contenida en el `StringBuilder` original. El método `append()` de la clase `StringBuilder`, permite añadir al final de una cadena contenida en un objeto `StringBuilder`, otra cadena tomada como parámetro.

En el código anterior, lo reescribimos usando la clase `StringBuffer` en lugar de `String` para concatenar:

```
StringBuilder aux = new StringBuilder("");  
  
for(int i = 0; i < arreglo.length; i++){  
  
    aux.append(arreglo[i].toString());  
  
}  
  
System.out.print("Listado de cargas:\n" + aux.toString());
```

- ✓ Java posee también la clase StringBuffer idéntica a StringBuilder, pero que posee sus **métodos sincronizados**, por lo cual se la podemos usar de manera segura en un ambiente de **multihilos**, tema que no se tratará en este módulo.

1.5 Otras clases útiles

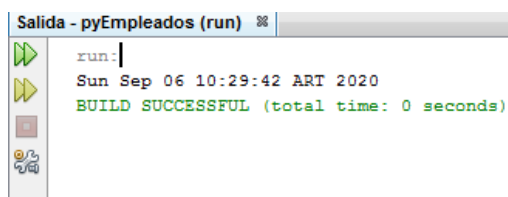
1.5.1 Manejo de Fechas

Los objetos de tipo **Date** en Java se utilizan para representar un instante específico en el tiempo (con precisión en milisegundos).

Para tomar la fecha/hora del sistema, simplemente escribimos:

```
System.out.println(new Date().toString());
```

Con la salida:



```
Salida - pyEmpleados (run) ✖  
run:  
Sun Sep 06 10:29:42 ART 2020  
BUILD SUCCESSFUL (total time: 0 seconds)
```

La clase `Date` (del paquete `java.util`) también permite convertir y parsear fechas. Sin embargo los métodos que proporcionaban esto ahora están **deprecated** (fuera de uso). En su lugar se ocupa la clase **Calendar** para conversiones y aritmética de fechas.

Esta clase también proporciona la habilidad representar fechas en distintos lenguajes o tipos de calendario específico.

Convertir objetos **Date** a **Calendar** es sencillo:

```
Date d = new Date(); //Crea el objeto Date
Calendar calendar = Calendar.getInstance(); //Obtiene una instancia de
//Calendar
calendar.setTime(date); //Asigna la fecha al Calendar
```

Una vez que tienes una instancia de `Calendar` puedes obtener información acerca de la fecha de la siguiente forma:

```
int year    = calendar.get(Calendar.YEAR);
int month   = calendar.get(Calendar.MONTH);
int weekOfMonth = calendar.get(Calendar.WEEK_OF_MONTH);
```

También podemos sumas o restar cantidades al objeto `Calendar`:

```
//Mostrar la fecha actual:

Date fechaHoy = new Date();

System.out.println(fechaHoy.toString());

//Asignar la fecha de hoy al objeto Calendar:

Calendar calendar = Calendar.getInstance();

calendar.setTime(fechaHoy);
```

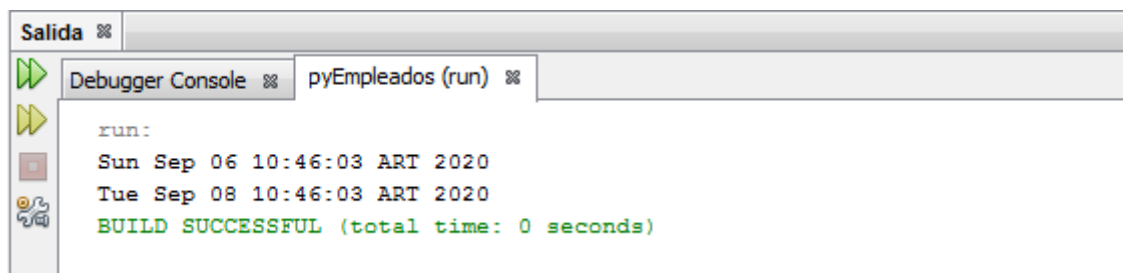
// sumar 2 días a la fecha actual y mostrar el calendar:

calendar.add(Calendar.DATE,2); // Constante Calendar.Date indica que se

//sumarán días

System.out.println(calendar.getTime());

Obteniendo la salida:



Por último para poder mostrar una fecha en un formato específico utilizamos un objeto auxiliar de la clase SimpleDateFormat (), tal como se muestra a continuación:

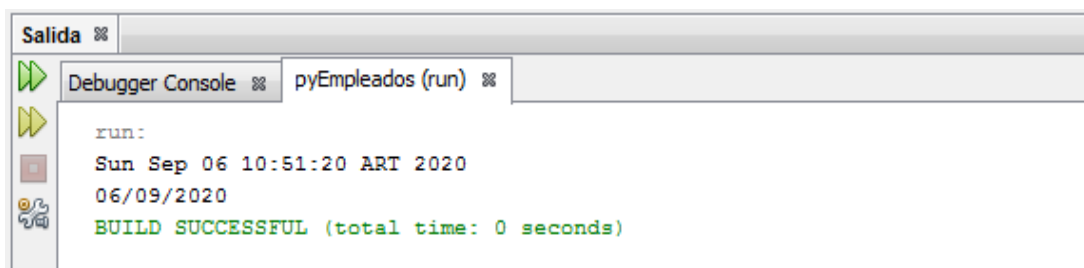
Date fechaHoy = new Date();

System.out.println(fechaHoy.toString());

SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");

System.out.println(sdf.format(fechaHoy));

Con la salida:



1.5.2 Números Aleatorios

Para generar números aleatorios en Java tenemos dos opciones. Por un lado podemos usar `Math.random()`, por otro la clase `java.util.Random`. La primera es de uso más sencillo y rápido. La segunda nos da más opciones.

Por ejemplo para generar un valor entero comprendido entre 0 y 6:

```
int rand = (int)(Math.random()*7);
```

`Math.random()` genera un valor de probabilidad, es decir: `[0, 1)`, lo que implica un valor entre 0 y casi 1 (sin incluir). Si multiplicamos por 7 el máximo será un entero igual a 6

Otra manera de generarlo:

```
Random rand = new Random();
```

```
rand.nextInt(7);
```

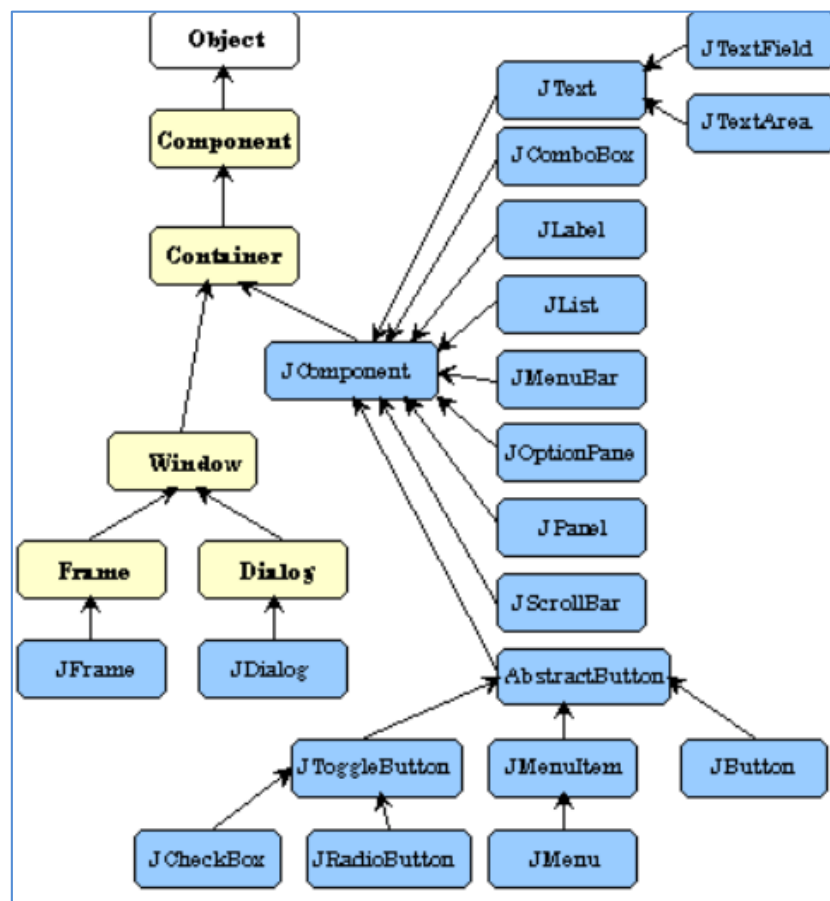
2. Swing y aplicaciones de ventanas

2.1 AWT (Abstract Windows Toolkit)

El lenguaje Java provee una amplia jerarquía de clases para desarrollo de interfaces de usuario de alto nivel basadas en ventanas. Esas clases conforman lo que se conoce como el Abstract Window Toolkit de Java (o AWT). La mayor parte de las clases del AWT vienen definidas en el paquete llamado **java.awt**.

Este paquete contiene una gran cantidad de clases para la gestión de las interfaces de usuario. Muchas de esas clases conforman una jerarquía que resulta muy valioso tener en claro.

Gráficamente:



La base de la jerarquía fundamental del AWT es la clase **Component**. En general, la clase Component representa cualquier objeto que tenga representación gráfica desplegable en pantalla, y que posea la capacidad de interactuar con el usuario. Esto incluye a objetos como botones, listas desplegables, etiquetas de texto, y a contenedores gráficos tales como cuadros de diálogo, ventanas, paneles, etc.

Si bien el paquete awt proporciona todas las clases necesarias para poder generar aplicaciones basadas en ventanas, nos centraremos en el uso de un paquete llamado **swing**.

Swing viene incluido con el entorno de desarrollo de Java (JDK) y extiende las funcionalidades de AWT. Esto implica que todas las clases de swing son especializaciones de clases pertenecientes a awt, pero con mejoras de rendimiento y renderización. Todas las clase están incluidas en el paquete **javax.swing.***;

2.2 Ventanas

La manera más sencilla de crear una ventana es declarar una clase propia que extienda de **JFrame**.

Por ejemplo:

```
public Ventana extends JFrame{  
  
    public Ventana(){  
  
        super("Primer ventana"); // El constructor de la clase JFrame permite  
        //definir el título de la ventana.  
  
        //Definir el tamaño de la ventana:  
  
        setSize(400, 300);
```


//Establecer la operación por defecto al cerrar la ventana:

```
setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
```

```
}
```

```
}
```

//Ejecutable:

```
public class Ejecutable {
```

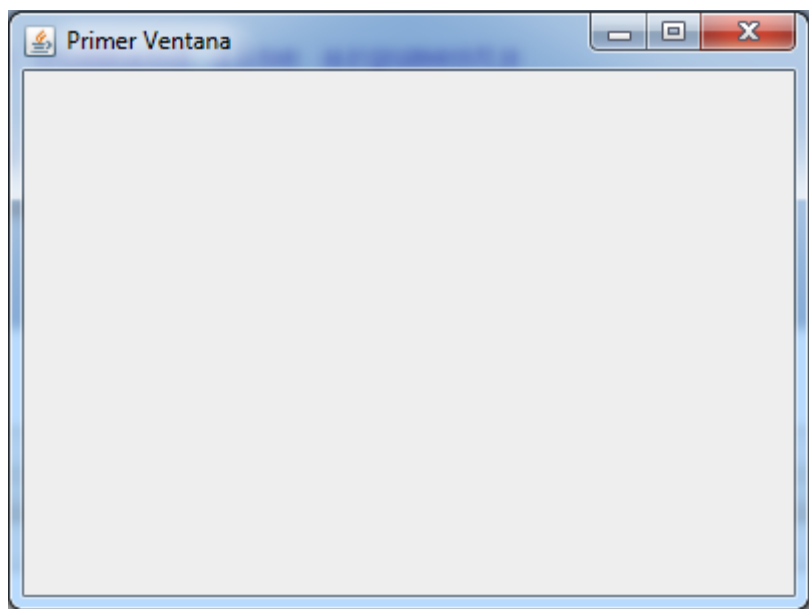
```
    public static void main(String[] args) {
```

```
        new Ventana().setVisible(true);
```

```
    }
```

```
}
```

Con la siguiente salida:



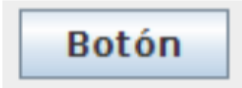
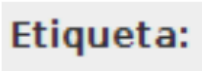
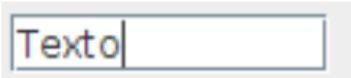

Para poder crear una clase de este tipo con Netbeans solo necesitamos hacer click derecho sobre un paquete y seleccionar **Nuevo >> Formulario JFrame**. Automáticamente Netbeans nos

habilitará una vista de diseño que nos permite agregar y ubicar componentes en el formulario mediante la opción Herramientas>>Paleta.

Las clases generadas de esta manera nos permitirán de manera visual manipular los componentes gráficos sin necesidad de escribir las líneas de código subyacentes. Todo el código Java se irá generando dinámicamente en un método llamado **initComponents()** que se invoca en el constructor de la clase creada. Todo el código autogenerado por Netbeans, por razones de seguridad, no será factible de modificar por el desarrollador.

2.3 Componentes

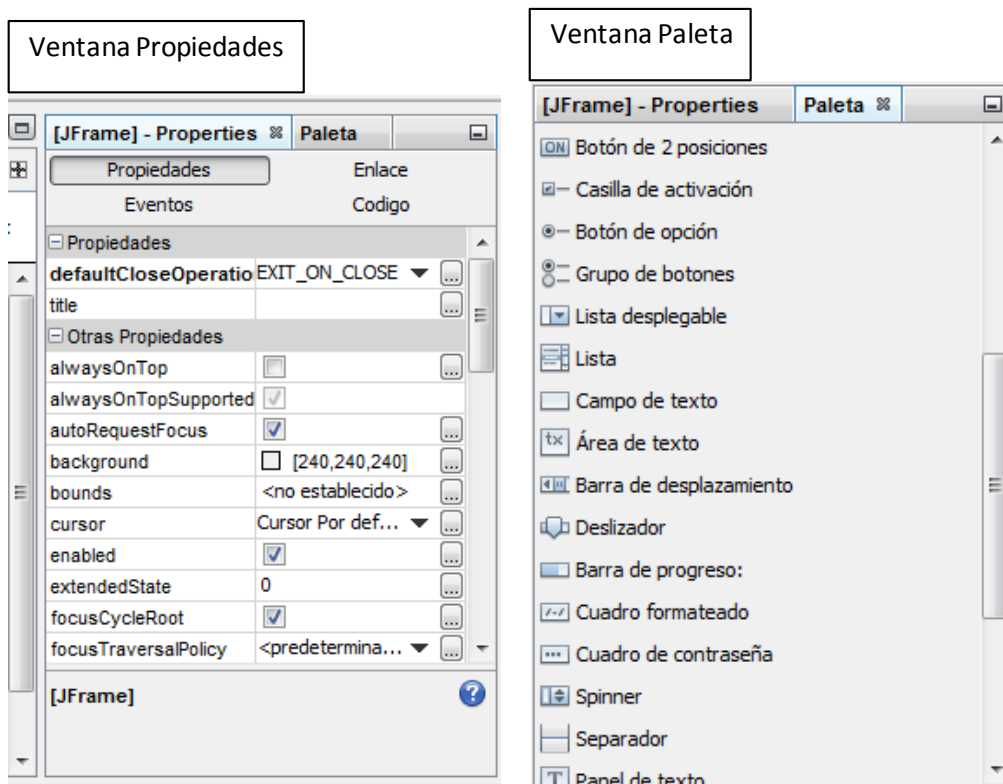
Algunos de los componentes usados con mayor frecuencia se muestran en el siguiente gráfico:

	JButton
	JLabel
	JTextField
	JCheckBox
	JRadioButton

Notar que todos nombres de los compontes de **swing** vienen prefijados con la letra **J**.

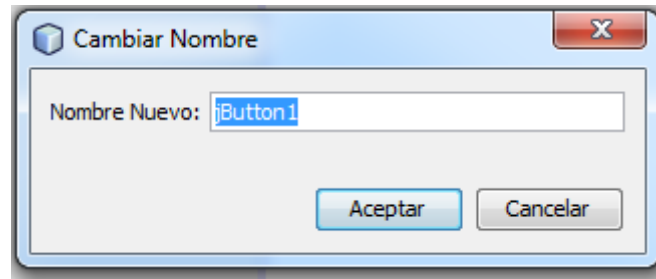
Para poder crear un componente simplemente debemos crear un objeto con el operador **new** y agregarlo al **contentPane** de la ventana correspondiente mediante el método `add`. Utilizando el IDE solamente arrastramos y soltamos el componente deseado en la vista de diseño (todo el código será generado en el método `initComponent()`).

Todo componente tiene ciertas propiedades como el color, el texto, el tipo de letra, tamaño, estado, etc. Para poder modificar estas propiedades utilizamos la ventana **Propiedades** de la opción **Ventana >> Herramientas IDE >> Propiedades**.



Por buena práctica lo primero a modificar en un componente luego de dibujarlo en una ventana es su nombre de la variable.

Para ello nos posicionamos en el componente y seleccionamos **click derecho >> Cambiar nombre de variable.**



2.4 Manejo de control de eventos

La idea es la siguiente: cuando una ventana se despliega, el usuario puede comenzar a interactuar con ella usando el mouse, el teclado, etc. Cada acción realizada por el usuario sobre los elementos desplegados en la ventana o con la ventana misma, se designa genéricamente como un **evento**. Son eventos: hacer click de mouse sobre los controles naturales de la ventana (minimización, cierre, activación cuando la ventana está en segundo plano, etc.), hacer click sobre un botón disponible en el interior de la ventana, seleccionar una opción de un menú, etc. Por cada evento generado por el usuario, la máquina virtual Java (JVM) crea de manera automática un objeto que representa a ese evento, y almacena en los atributos de ese objeto la descripción del evento. Esos objetos descriptores de eventos, pertenecen a clases predefinidas diversas según haya sido el evento producido. Esas clases pertenecen al paquete `java.awt.event`, que fue uno de los que hemos importado en este modelo para la clase Ventana.

Las clases descriptoras de eventos más comunes son las siguientes:

- **WindowEvent:** acciones sobre los controles naturales de una **ventana** (cierre, minimización, activación, etc.)
- **ActionEvent:** operaciones sobre controles definidos por el usuario (típicamente botones o items de menú)
- **ItemEvent:** selección o des-selección de ciertos controles (listas, checkboxes, etc.)
- **KeyEvent:** presión de teclas en el teclado
- **MouseEvent:** acciones de mouse sobre componentes desplegados (click, arrastre, etc.)

Para responder a los eventos producidos por el usuario en la interfaz, el programador debe programar métodos de respuesta a esos eventos. Los métodos de respuesta que debe programar dependen del tipo de evento al que se quiere responder. Por ejemplo, si se desea responder a eventos de presión de botones del usuario (eventos representados por objetos de la clase `ActionEvent`), debe ser programado el método `actionPerformed()` con la respuesta a esos eventos. ¿Cómo sabe el programador cuáles son los métodos que debe programar en respuesta a cada clase de evento? Los obtiene de las llamadas interfaces de escucha.

Por cada clase descriptora de eventos, existe una clase de interface que contiene los encabezados de los métodos de respuesta que deben ser programados para responder a los eventos descriptos. Esas interfaces se conocen como interfaces de escucha, y están incluidas en el paquete `java.awt.event`. El nombre de una interface de escucha es simple de recordar: se llama igual que la clase

descriptor de eventos a la que responde, pero cambiando la terminación Event por Listener. Así, si la clase descriptor de eventos se llama WindowEvent, la interface de escucha se llamará WindowListener. Los métodos indicados por cada interfaz de escucha reciben automáticamente como parámetro a los objetos de las clases descriptoras que la JVM haya creado al producirse un evento.

En Netbeans simplemente nos posicionamos sobre el componente y seleccionamos la opción **Eventos** dentro de la ventana **Propiedades** analizada anteriormente, y seleccionamos el tipo de evento a gestionar. El IDE automáticamente nos creará un método privado en la clase Formulario y registrará dicho evento al componente en el código del método initComponents().

3. Apéndice

3.1 Palabras reservadas del lenguaje

<code>abstract</code>	<code>default</code>	<code>for</code>	<code>package</code>	<code>synchronized</code>
<code>assert</code>	<code>do</code>	<code>if</code>	<code>private</code>	<code>this</code>
<code>boolean</code>	<code>double</code>	<code>implements</code>	<code>protected</code>	<code>throw</code>
<code>break</code>	<code>else</code>	<code>import</code>	<code>public</code>	<code>throws</code>
<code>byte</code>	<code>enum</code>	<code>instanceof</code>	<code>return</code>	<code>transient</code>
<code>case</code>	<code>extends</code>	<code>int</code>	<code>short</code>	<code>true</code>
<code>catch</code>	<code>false</code>	<code>interface</code>	<code>static</code>	<code>try</code>
<code>char</code>	<code>final</code>	<code>long</code>	<code>strictfp</code>	<code>void</code>
<code>class</code>	<code>finally</code>	<code>native</code>	<code>super</code>	<code>volatile</code>
<code>continue</code>	<code>float</code>	<code>new</code>	<code>switch</code>	<code>while</code>

3.2 Caracteres de escape

A continuación hay una lista de secuencias de escape:

- `\n` ----> Nueva Línea.
- `\t` ----> Tabulador.
- `\r` ----> Retroceso de Carro.
- `\f` ----> Comienzo de Página.
- `\b` ----> Borrado a la Izquierda.
- `\\` ----> El carácter barra inversa (`\`).
- `\'` ----> El carácter prima simple (`'`).
- `\"` ----> El carácter prima doble o bi-prima (`"`).