Año 2022

Índice

Introducción a Java

Java es un lenguaje de programación orientado a objetos multiplataforma.

Es un lenguaje de tipo **compilado**, es decir, un lenguaje donde se escribe un código y, antes de poder ejecutarse, debe pasar por un compilador que traduce el código a otro lenguaje el cual, otro programa, lo convierte a lenguaje binario para que pueda ser comprendido por el ordenador.

Dicho compilador se trata de Java Development Kit (JDK), el cual debe ser descargado para comenzar a trabajar con Java.

Es necesario saber que el código de Java se lee de arriba hacia abajo y de derecha a izquierda.

Es importante recordar los principios básicos de la programación:

1. Analizar la situación
2. Resolver el problema
3. Programar

Definiciones de programación

Algoritmo

Serie ordenada, finita y precisa de acciones que resuelven un problema.

Siempre se debe obtener el mismo resultado si se tienen los mismos datos de entrada (como una receta de cocina).

Programa

Traducción de un algoritmo a un lenguaje de programación determinado capaz de ser ejecutado por una computadora.

Un programa es un algoritmo, pero no todo algoritmo es un programa.

- **Datos de entrada:** Toda información que llega al algoritmo.
- **Datos de salida:** Información que sale del algoritmo.

Pre-condiciones

Todas las condiciones que el algoritmo asume que cumplen sobre los datos de entrada.

Pos-condiciones

Todas las condiciones que va a cumplir el resultado del algoritmo y sólo se pueden cumplir si se cumplen las pre-condiciones.

Caja negra

Se observan sólo los datos de entrada y los datos de salida para comprobar el funcionamiento del algoritmo.

Se centra en qué hace el algoritmo y se preocupa en cumplir las pre-condiciones.

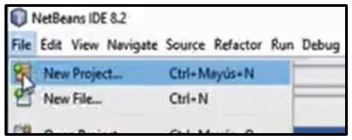
Caja blanca

Se observa el algoritmo desde dentro para ver todo su funcionamiento.

Se centra en cómo se hace el algoritmo y se preocupa por cumplir las pos-condiciones.

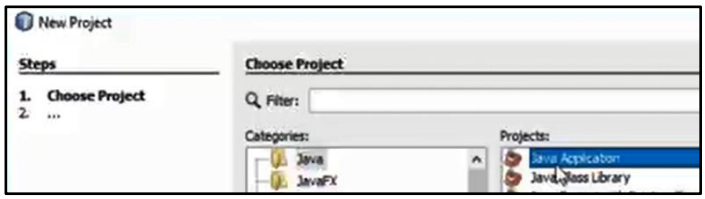
Interfaz de Desarrollo

Uno de los Entornos de Desarrollo Integrados (IDE son sus siglas en inglés), es **NETBEANS**.

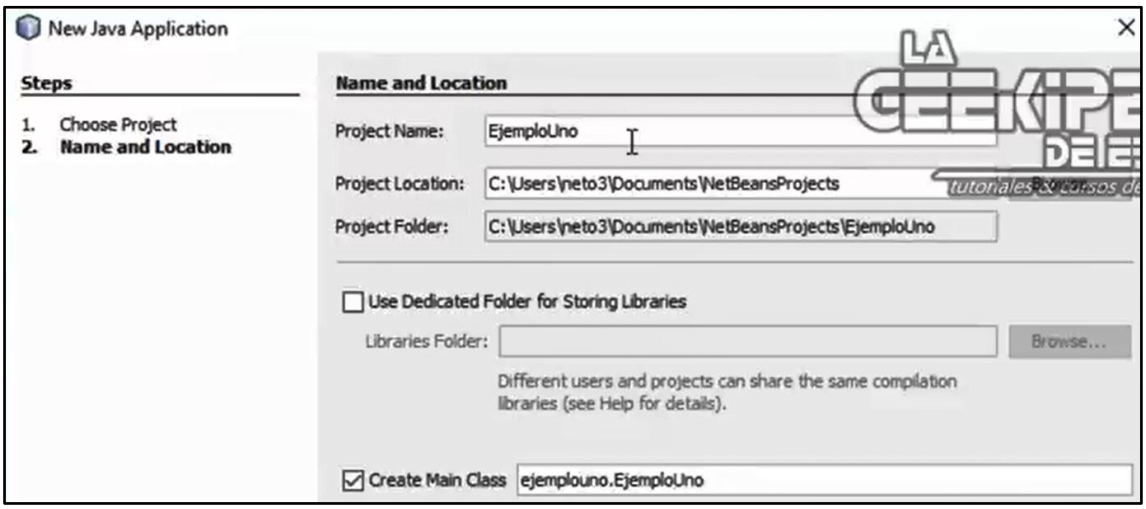


Para empezar a utilizar Netbeans, se debe empezar creando un nuevo proyecto.

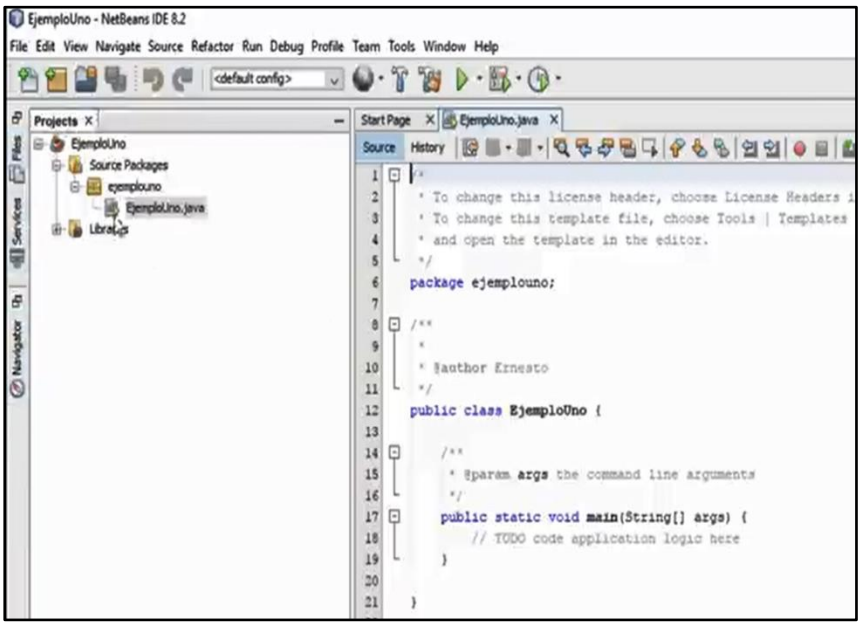
Se debe elegir la opción “Java Aplication”.



Luego se procede a escribir el nombre del proyecto, el cual debe estar escrito sin espacios.



Posteriormente, se creará la primera class del proyecto donde podremos empezar a trabajar.



JavaDocs

Mediante el uso de etiquetas específicas, Netbeans permite documentar el trabajo de forma “automática”.

@author establece el nombre del autor del proyecto.

@param indica qué datos de entrada fueron enviado por medio de argumentos a un método o función.

Primeros pasos en Java

Para empezar a programar en Java, se debe crear un archivo con extensión **.java**

La primera plantilla que se muestra (nuestra primer class) debe iniciar con el nombre de la class con la primer letra en mayúscula, el cual debe ser el mismo que el del archivo. Si hay más de una palabra, se escribe sin espacio y con mayúscula en cada letra inicial de cada palabra.

A continuación, y dentro de la class, se debe escribir el **método main**, que es donde iniciará la ejecución del código.

Introduccion.java

```
1   public class Introduccion {
2       public static void main(String args[]) {
3
4           System.out.println("Hola mundo!");
5
6       }
7   }
```

Para compilar en Java, se debe abrir el **Simbolo del sistema** y dirigirnos hasta el directorio donde se encuentra el archivo java que creamos.

Mediante el comando **dir** se pueden “ver” los directorios de la carpeta actual.

Con el comando **cd** seguido del nombre de un directorio, se puede acceder a él.

Una vez encontrado nuestro archivo java en el Símbolo del sistema, se debe utilizar el comando **javac** (brindado por JDK) seguido del nombre de nuestro archivo java junto con su extensión.

```
C:\Users\MiUsuario\Desktop> javac  Introduccion.java
```

De esta manera, se creará un archivo **.class**, el cual puede ser ejecutado con el Símbolo del sistema mediante el comando **java** seguido del nombre de nuestro archivo class (sin su extensión).

```
C:\Users\MiUsuario\Desktop> java  Introduccion
```

Empaquetado de un proyecto

Para que un programa creado con Java pueda ser ejecutado sin necesidad de utilizar el símbolo del sistema, es necesario empaquetarlo.

Para ello, se deben colocar todos los archivos .java y .class (junto con la carpeta de imágenes, en caso de que corresponda) en una misma carpeta.

En dicha carpeta, se debe crear un archivo de texto llamado **MANIFEST.MF**

El contenido del archivo manifest debe contener:

```
Manifest-Version: 1.0
Created-By: Mi nombre
Main-Class: Nombre_de_la_class_donde_está_el_main
X-COMMENT: un comentario

```

es necesario dejar esta línea en blanco

Luego, en el símbolo del sistema, se debe ejecutar la siguiente línea de código (estando ubicados en el directorio con todos los archivos):

```
jar cvfm MiProyecto.jar manifest.mf Introduccion.class images/mi_imagen.png
```

Nombre del proyecto | Todos los class del proyecto | Todas las imágenes

Finalmente, se ejecuta la siguiente línea de código en el símbolo del sistema para crear el archivo .jar ejecutable:

```
java -jar MiProyecto.jar
```

Tipos de datos

Primitivos

	nombre	*rango*
Enteros	byte	± 2^8 / 2
	short	± 2^{16} / 2
	int	± 2^{32} / 2
	long	± 2^{64} / 2
Decimales	float	8 decimales
	double	15 decimales
Caracteres	char	un solo caracter
Lógicos	bool	**true** *o* **false**

Objetos

Se caracterizan por ser necesaria su invocación debido a que son clases y, como tales, su inicial es en mayúscula.

- Cadenas de texto **String**
- Arreglos **Array** *vectores y matrices*

Variables

Una variable es un espacio en memoria donde se puede alojar información o datos.
Está conformado por 2 elementos: el tipo de dato y un nombre.

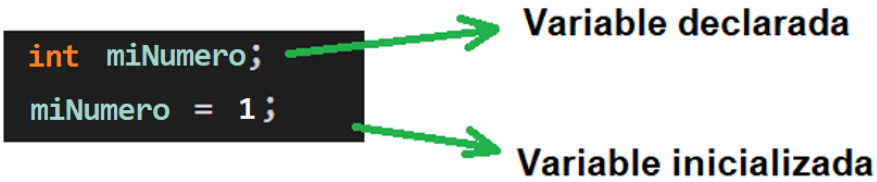
```
int  miNumero;
```

El nombre de una variable tiene ciertas reglas.

1. La letra inicial no puede ser un número ni un símbolo (excepto guion bajo o $).
2. Las constantes deben ser escritas sólo con letras mayúsculas.
3. Deben describir brevemente lo que la variable representa.
4. Si el nombre de la variable tiene más de una palabra, la inicial de las siguientes palabras se escribe en mayúscula (**camel case**).

Inicializar variables

Es el proceso de asignar por primera vez un valor a una variable.



Se puede resumir de la siguiente manera:

```
int  miNumero = 1;
```

Scope de una variable

El ámbito de una variable define su alcance de uso. Indica en qué secciones de código una variable estará disponible.

- **Local / Bloque** — Sólo puede utilizarse dentro del método o bloque de código en que se encuentra.
- **Global / de instancia** — Puede utilizarse en cualquier parte de la class. Incluso puede utilizarse desde fuera de esta class según su modificador de acceso. No debe inicializarse.
- **Estático / de clase** — Pertenecen a la propia clase y para acceder a ellas desde otra class no es necesario crear una instancia de clases. Se denominan atributos de clase.

```
public class Tutorial {
    static int variableEstatica;

    int variableGlobal;

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

Operadores

Símbolo que indica que debe ser llevada a cabo una operación especificada sobre cierto número de operador (tipo de dato).

- **Asignación** **=** Asigna un valor a un operando.

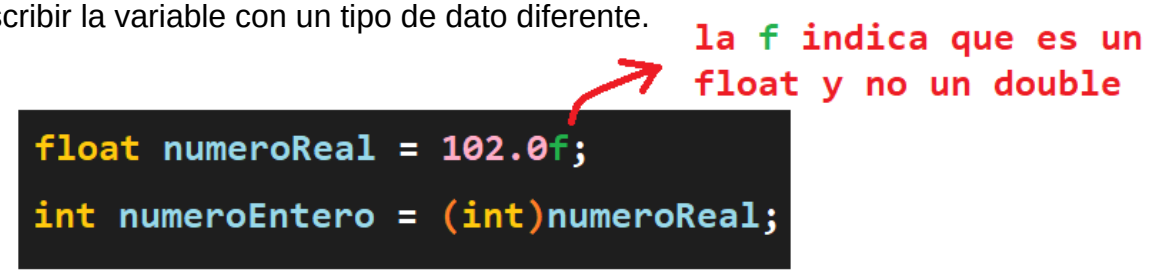
- **Aritméticos**
  - **Suma** **+** Suma dos números o concatena textos.
  - **Sustracción** **-** Resta dos números.
  - **Multiplicación** ***** Multiplica números.
  - **División** **/** Divide números.
  - **Resto** **%** Divide números y se obtiene el resto de dicha división.
  - **Exponenciación** **^** Aplica potencias.

- **Relacionales** Compara dos valores del mismo tipo y devuelve un valor lógico.
  - **Mayor estricto** **>** **true** *o* **false**
  - **Menor estricto** **<**
  - **Mayor o igual** **>=**
  - **Menor o igual** **<=**
  - **Desigualdad** **!=**
  - **Igualdad** **==**
    - Si se desea comprar dos cadenas de texto (string), se debe usar el método **`.ecuals`**

- **Lógicos**
  - **And** **&&** Si todo es verdadero, retorna verdadero.
  - **Or** **||** Si al menos uno es verdadero, retorna verdadero.
  - **Not** **!** Convierte lo verdadero en falso y viceversa.

Casting

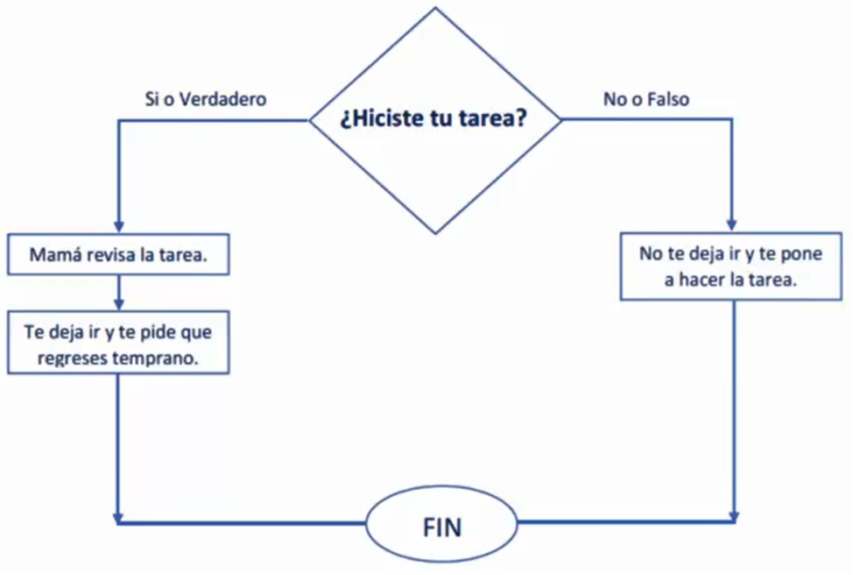
Conversión de un tipo de dato a otro mediante el uso del tipo de dato entre paréntesis antes de escribir la variable con un tipo de dato diferente.



Estructuras de control

Estructura condicional

Ejecutan un bloque de código si se cumple una condición (**if**, **if-else**, **switch**).



Iteradores

Ejecuta un bloque de código varias veces (**while**, **do-while**, **for**).

Funciones y procedimientos

Bloque de código el cual se puede utilizar declarándolo fuera del main (modularizar).

```
tipo_de_dato nombre_de_la_funcion ( datos de entrada ) {

    . . .

    return dato de salida;

}
```

- **Funciones**
  Se utilizan para realizar una acción específica.

```
/*
 * Pre-condiciones: -
 * Pos-condiciones: devuelve un entero, suma de ambos operandos.
*/
int suma(int operando_1, int operando_2) {
    int resultado = operando_1 + operando_2;
    return resultado;
}
```

- **Procedimientos (void)**
  Realizan acciones sin devolver ningún resultado.
  Se nombran con verbos.

```
/*
 * Pre-condiciones: -
 * Pos-condiciones: imprime formato por pantalla
 *                  con el caracter recibido como parámetro
*/
void saludar(char inicial) {
    System.out.println("Hola " + inicial + ", ¿cómo estás?");
}
```

Ninguna función o procedimiento debe recibir más de 5 o 7 parámetros.

```
/*
 * Pre-condiciones: -
 * Pos-condiciones: imprime formato por pantalla
 *                  con el caracter recibido como parámetro
*/
void saludar(char inicial) {
```

firma

Control de errores

Gestión de la excepción

Las excepciones son el medio para tratar situaciones anómalas que pueden suceder, como por ejemplo:

- Invocar a un método sobre un objeto "null".
- Intentar abrir un fichero que no existe para leerlo.
- Intentar dividir un numero sobre cero.

Si un código no se ejecuta de la forma prevista, se establece cómo debe responder el programa.

Para ello, se utilizan las palabras reservadas:

- **try** función que se intentará realizar.
- **catch** lo que sucederá si surge un error.
- **finally** lo que sucederá independientemente de si hubo un error o no.

```
void dividirPorCero() {

    // Declaración de variables locales (sólo disponibles para la función)
    int resultado = 0;
    int operando1 = 10;
    int operando2 = 0;

    // Gestión de errores
        try {
            resultado = operando1 / operando2;
            // Una vez ocurrido el error, se pasa a ejecutar el bloque catch
            System.out.println(resultado);// este mensaje nunca será mostrado
        } catch (Exception e) {
            System.out.println(e.toString());// muestra un mensaje de error
        } finally {
            System.out.println("Este mensaje siempre será mostrado");
        }
}
```

Definición de un error

Se puede definir un tipo de error mediante la palabra reservada **throws** / **throw**.

Al crearse una función con este tipo de declaración, forzosamente se deberá utilizar mediante una gestión de errores, es decir, deberá usarse con **try – catch**.

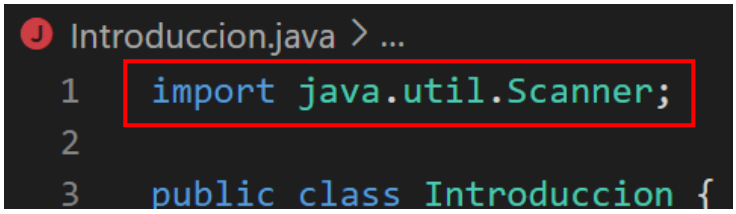
En código, se escribe de la siguiente manera:

```
int division(int numerador, int denominador) throws Exception {

    if(denominador != 0) {
        return numerador / denominador;
    } else {
        throw new Exception("No se puede dividir por 0");
    }

}
```

Bibliotecas

Una biblioteca es un conjunto de clases que contienen atributos y métodos, es decir, un conjunto de códigos escritos por alguien para facilitar ciertas tareas.

Existen diversas bibliotecas nativas de Java, y para hacer uso de ellas, se necesita utilizar la palabra clave **import** seguida de la ruta donde se encuentra la biblioteca.



Scanner

Es una biblioteca que permite, entre otras cosas, obtener datos ingresados por el usuario en pantalla.

```java
import java.util.Scanner;

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

        Scanner input = new Scanner(System.in);
        String nombre = "";
        int edad = 0;

        System.out.println("¿Cuál es tu nombre?");
        nombre = input.nextLine();

        System.out.println("¿Cuál es tu edad?");
        edad = input.nextInt();

        System.out.println("Tu nombre es " + nombre);
        System.out.println("Y tu edad es " + edad);

    }
}
```

Objetos y Clases

Una **class** es un objeto, se escriben con la primera inicial en mayúsculas y su función es, básicamente, la de un algoritmo (resolver un problema).

Para crearse, se deben declarar e inicializar de la siguiente forma:

```
public static class Nombre {// declaración de la class

    public Nombre (datos de entrada) {// inicialización de la class
        // código
    }
}
```

Para que esta class pueda ser utilizada como una biblioteca, se debe crear el código completo de la class con su respectiva funcionalidad.

Luego, desde la class en la cual quiere ser utilizado como biblioteca, se la debe importar (**import**) para utilizarla y luego realizar una **instancia de clases**.

```
Nombre operaciones = new Nombre(datos);
```

Cuando se empieza a trabajar con más de una class, es necesario conocer la palabra reservada **this**, la cual es una forma de indicar que se está trabajando con la class en la cual está metida dicho **this**.

Atributos de clases

Recordando que las class pueden ser trabajadas como objetos, se le pueden asignar atributos mediante la palabra reservada **static** (variables estáticas, específicas de una class).

```
public static String nombre = "Soy un atributo de clase";
```

Métodos de clases

Son las funciones o procedimientos pertenecientes a una class.

La palabra reservada **public** permite que las variables / funciones declaradas puedan ser utilizadas por otra class.

La palabra reservada **private** indica que las variables / funciones declaradas sólo puedan ser utilizadas por la class en la cual existen.

Ejemplo de una class

```java
public class Culculadora {

   // Declaración de variables que no serán accesibles (privadas)
    private int operando1;
    private int operando2;

    // Constructor de la class (donde se inicializa)
    public Culculadora(int operando1, int operando2) {
        this.operando1 = operando1;
        this.operando2 = operando2;
    }

    /* Pre-condiciones: -
     * Pos-condiciones: devuelve la sumatoria entre los operandos ingresados
    */
    public int suma() {
        return operando1 + operando2;
    }

    /* Pre-condiciones: -
     * Pos-condiciones: devuelve la sustracción entre los operandos ingresados
    */
    public int resta() {
        return operando1 - operando2;
    }

    /* Pre-condiciones: -
     * Pos-condiciones: devuelve el producto de los operandos ingresados
    */
    public int multiplicacion() {
        return operando1 * operando2;
    }

    /* Pre-condiciones: denominador debe ser distinto de cero
     * Pos-condiciones: devuelve la división de los números ingresados
    */
    public int division() throws Exception {
        if(operando2 != 0) {
            return operando1 / operando2;
        } else
            throw new Exception("No se puede dividir por 0");
    }
}
```

Para poder utilizarlo desde otra class, se debe realizar una **instancia de clases**:

```java
Culculadora operacion = new Culculadora(numero1, numero2);
operacion.suma();// suma el numero1 y el numero2
```

Otra manera de utilizar una class como biblioteca sin necesidad de instanciarla, es haciendo uso de la palabra reservada **static**. La cual, hace referencia a los atributos de una class como si fueran las características un objeto.

```java
public class Culculadora {

    // No hay necesidad de declarar variables privadas
    // ni de inicializar la class

    /*
     * Pre-condiciones: -
     * Pos-condiciones: devuelve la sumatoria entre los operandos ingresados
    */
    public static int suma(int operando1, int operando2) {
        return operando1 + operando2;
    }

    /*
     * Pre-condiciones: -
     * Pos-condiciones: devuelve la sustracción entre los operandos ingresados
    */
    public static int resta(int operando1, int operando2) {
        return operando1 - operando2;
    }

    /*
     * Pre-condiciones: -
     * Pos-condiciones: devuelve el producto de los operandos ingresados
    */
    public static int multiplicacion(int operando1, int operando2) {
        return operando1 * operando2;
    }

    /*
     * Pre-condiciones: denominador debe ser distinto de cero
     * Pos-condiciones: devuelve la división de los números ingresados
    */
    public static int division(int operando1, int operando2) throws Exception {
        if(operando2 != 0) {
            return operando1 / operando2;
        } else
            throw new Exception("No se puede dividir por 0");
    }
}
```

Para hacer uso de las funciones de esta class, se procede sin necesidad de instanciarla:

```java
Culculadora.suma(numero1, numero2);// suma el numero1 y el numero2
```

Existen muchas clases nativas de Java, las cuales incluyen los Strings y muchos objetos más.

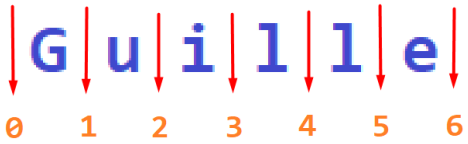
Cadenas de texto (Strings)

Las cadenas de texto tienen métodos exclusivos que permiten manipularlos.

- **.substring(desde, hasta)** Permite obtener una parte en específico de un string.

  Para hacer buen uso del método, es necesario saber que una cadena de texto se divide en posiciones, las cuales comienzan en 0 y finalizan luego del último carácter.

```
String texto = "Guille";
```



  Por lo tanto, si se utiliza la siguiente instrucción:

```
texto.substring(1 , 5);
```

  → uill

- **.lenght** Permite obtener el largo de una cadena de texto.

- **.equals(otroTexto)** Permite comparar dos cadenas de texto.

```
if(nombre.equals(otroNombre)) {
```

Arreglos

Estructura de datos que almacena una colección de datos bajo un mismo nombre. Todos deben pertenecer a un mismo tipo de dato.

- Unidimensional (Vectores)

Los elementos se almacenan en posiciones contiguas de memoria, identificados por un índice (índex).

```
int nombre[] = new int[5];
```

Su representación gráfica es la siguiente:

Para guardar un dato en una de las posiciones, se debe especificar dicha posición entre corchetes.

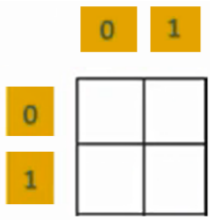
```
nombre[2] = 9;
```

- Bidimensionales (Matrices)

Los elementos se almacenan en posiciones contiguas de memoria, identificados por dos índices (índex).

```
int nombre[][] = new int[2][2];
```

Su representación gráfica como matriz es la siguiente:



Para guardar un dato en una de las posiciones, se debe especificar dicha posición entre corchetes.

```
int nombre[0][1] = 7;
```

Interfaz gráfica

Programa informático que facilita el entorno al usuario al momento de manipular el programa.

Se debe pensar a la clase (class) como un objeto, el cual es, en este caso, una interfaz gráfica con sus propiedades y componentes.

Para usar la class como interfaz gráfica, se debe hacer uso de la **herencia** de JFrame. Para ello, se debe importar la biblioteca **javax.swing.***

```
import javax.swing.*;

public class MiPrimerFrame extends JFrame {

    // Método main: donde inicia el programa
    public static void main(String[] args) {
        MiPrimerFrame interfaz = new MiPrimerFrame();
        interfaz.initComponents();// Componentes de interfaz
        interfaz.setVisible(true);// Visibiliza interfaz
    }

    // Constructor de igual nombre que la class
    public MiPrimerFrame() {
        setTitle("Tutorial");// Establece el título de la interfaz
        setLayout(null);// permite colocar las coordenadas de la interfaz manualmente
        setBounds(0,0,500,500);// posición X, posición Y, Ancho, Alto
        setLocationRelativeTo(null);// Interfaz en el centro de la pantalla
        setResizable(false);// Establece inmodificable el tamaño de la interfaz de forma manual
        setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);// Termina el programa al cerrar la interfaz
        //setSize(640,480);// (opcional) cambia el tamaño de la interfaz
    }

    // Crear componentes de la interfaz
    public void initComponents() {
        . . .
    }

}
```

¡ATENCIÓN!
ES NECESARIO QUE .setVisible SEA EJECUTADO LUEGO DE CREARSE TODO EL FRAME Y SUS COMPONENTES

inicialización de la class, la cual es tratada como un objeto (en este caso, una interfaz)

Una forma de crear interfaces utilizando el IDE Netbeans es mediante la creación de un JFrame Form.

Componentes de la interfaz

Todos los objetos de la interfaz deben ser declarados justo después del **public class**

- **JLabel** (TextView): etiqueta

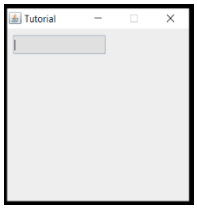
```
private JLabel etiqueta;
```

```
etiqueta = new JLabel("Este es un Label");
// Fuente, (0 = normal, 1 = negrita, 2 = cursiva, 3 = ambos), tamaño
etiqueta.setFont(new java.awt.Font("Andale Mono", 3, 12));
etiqueta.setBounds(10,10,300,30);// X, Y, Ancho, Alto
etiqueta.setForeground(new java.awt.Color(255,0,255));// R,G,B (colorea el texto)
add(etiqueta);
```

- **JTextField** (EditText): caja de texto *o* input-text

```
private JTextField inputText;
```

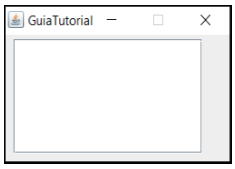
```
inputText = new JTextField();// se puede poner texto dentro con ""
inputText.setBounds(10,10,150,30);// X, Y, Ancho, Alto
inputText.setBackground(new java.awt.Color(224,224,224));// R,G,B (colorea el contorno)
inputText.setFont(new java.awt.Font("Andale Mono", 1, 12));
inputText.setForeground(new java.awt.Color(50,50,50));// R,G,B (colorea el texto)
add(inputText);
```



- **JTextArea**

```
private JTextArea textArea;
private JScrollPane paneldescroll;
```

```
textArea = new JTextArea();// se puede poner texto dentro con ""
textArea.setEditable(true);// permite que se edite el texto
textArea.setFont(new java.awt.Font("Andale Mono", 0, 14));
paneldescroll = new JScrollPane(textArea);// Brinda una barra de scroll al textArea
paneldescroll.setBounds(10,50,450,130);// X, Y, Ancho, Alto
add(paneldescroll);// Si no se quiere añadir panel de scroll, se trabaja sobre textoarea
```



Se puede obtener el texto introducido en una caja de texto mediante el siguiente método:

```
jTextField.getText();
```

Y se puede pegar un texto dentro de una caja de texto mediante **.setText** o agregar un texto mediante **.append**

```
jTextField.setText(texto);
```

```
jFieldText.append(texto);
```

Si se desea quitar los espacios en blanco de un texto, se puede usar **.trim()**

```
texto.trim();
```

Se puede obtener un valor numérico de un texto mediante el uso del siguiente método:

```
int numero = Integer.parseInt(jTextField.getText());
```

Puede ser necesario usar **try – catch** debido a que, si no hay números, se producirá un error.

- **JButton**: permite crear botones con eventos.

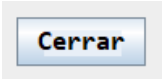
```
private JButton botonCerrar;
```

```
import java.awt.event.*;// permite agregar botones con eventos
```

```
public class MiPrimerFrame extends JFrame implements ActionListener {
```

```
botonCerrar = new JButton("Cerrar");
botonCerrar.setBounds(370,400,80,30);// X, Y, Ancho, Alto
botonCerrar.setBackground(new java.awt.Color(255,255,255));// R,G,B (colorea el fondo)
botonCerrar.setFont(new java.awt.Font("Andale Mono", 1, 12));
botonCerrar.setForeground(new java.awt.Color(255,255,0));// R,G,B (colorea el texto)
add(botonCerrar);// Crea el boton
botonCerrar.addActionListener(this);// boton tendrá un evento/trigger
botonCerrar.setEnabled(true);
```

```
@Override
public void actionPerformed(ActionEvent ae) {
    if(ae.getSource() == botonCerrar) {
        System.exit(0);// Cerrar programa
    }
}
```



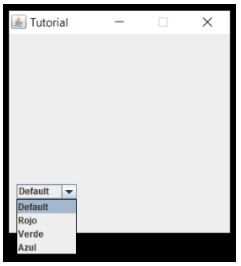
- **JComboBox** (Spinner): lista desplegable

```
private JComboBox spinner;
```

```
import java.awt.event.*;// permite agregar listas con eventos
```

```
public class MiPrimerFrame extends JFrame implements ItemListener {
```

```
spinner = new JComboBox();// aqui dentro no se puede colocar texto
spinner.setBounds(10,200,80,20);// X, Y, Ancho, Alto
add(spinner);
spinner.addItem("Default");// añade texto/opciones a la lista
spinner.addItem("Rojo");// añade texto/opciones a la lista debajo del anterior
spinner.addItem("Verde");// añade texto/opciones a la lista debajo del anterior
spinner.addItem("Azul");// añade texto/opciones a la lista debajo del anterior
spinner.addItemListener(this);// lista tendrá un evento/trigger
```




```
@Override
public void itemStateChanged(ItemEvent ie) {

    if(ie.getSource() == spinner) {
        // obtiene lo seleccionado de la lista y lo copia en el string "seleccion"
    String seleccion = spinner.getSelectedItem().toString();
    java.awt.Color colorear = new java.awt.Color(0,0,0);
            if(seleccion.equals("Rojo") == true) colorear = new java.awt.Color(255,0,0);
            else if(seleccion.equals("Verde") == true) colorear = new java.awt.Color(0,255,0);
            else if(seleccion.equals("Azul") == true) colorear = new java.awt.Color(0,0,255);
            else colorear = null;
    System.out.println("Seleccionaste '" + seleccion + "'");
    }

}
```

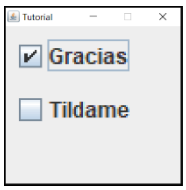
- **JCheckBox**

```
private JCheckBox cbCheck1, cbCheck2;
```

```
import javax.swing.event.*; // Permite agregar eventos en checkbox
```

```
public class MiPrimerFrame extends JFrame implements ChangeListener {
```

```
cbCheck1 = new JCheckBox("Checkeame");
cbCheck1.setBounds(10,10,150,30);// X, Y, Ancho, Alto
cbCheck1.addChangeListener(this);// checkbox tendrá un evento/trigger
add(cbCheck1);
//
cbCheck2 = new JCheckBox("Tildame");
cbCheck2.setBounds(10,40,150,30);// X, Y, Ancho, Alto
cbCheck2.addChangeListener(this);// checkbox tendrá un evento/trigger
add(cbCheck2);
```




```
@Override
public void stateChanged(ChangeEvent e) {

    if(cbCheck1.isSelected() == true) {
        cbCheck1.setText("Gracias");
    } else cbCheck1.setText("Checkeame");

    if(cbCheck2.isSelected() == true) {
        cbCheck2.setText("Thank you");
    } else cbCheck2.setText("Tildame");

}
```

¡ATENCIÓN!
El procedimiento **stateChanged**
se ejecuta muchas veces seguidas

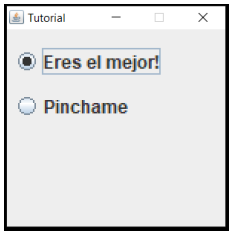
- **JRadioButton**: sólo un radio button puede estar activo a la vez.

```
import javax.swing.event.*; // Permite agregar eventos en radio button
```

```
private ButtonGroup bg1;
private JRadioButton bg1_rb1, bg1_rb2;
```

```
public class MiPrimerFrame extends JFrame implements ChangeListener {
```

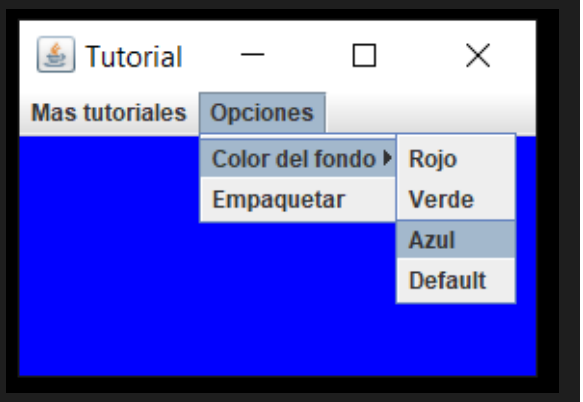
```
bg1 = new ButtonGroup();// se crea el grupo de botones de radio

bg1_rb1 = new JRadioButton("Tocame");
bg1_rb1.setBounds(300,240,150,30);// X, Y, Ancho, Alto
bg1_rb1.addChangeListener(this);// boton de radio tendrá un evento/trigger
add(bg1_rb1);
bg1.add(bg1_rb1);

bg1_rb2 = new JRadioButton("Pinchame");
bg1_rb2.setBounds(300,270,150,30);// X, Y, Ancho, Alto
bg1_rb2.addChangeListener(this);// boton de radio tendrá un evento/trigger
add(bg1_rb2);
bg1.add(bg1_rb2);
```




```
@Override
public void stateChanged(ChangeEvent e) {

    if(bg1_rb1.isSelected() == true) {
            bg1_rb1.setText("Eres el mejor!");
    } else bg1_rb1.setText("Tocame");
    if(bg1_rb2.isSelected() == true) {
            bg1_rb2.setText("Fenomenal!");
    } else bg1_rb2.setText("Pinchame");

}
```

¡ATENCIÓN!
El procedimiento **stateChanged**
se ejecuta muchas veces seguidas

- **JMenu**




```
import javax.swing.*;

import java.awt.Color;
import java.awt.Container;

import java.awt.event.*;// Permite añadir eventos al menú

public class MiPrimerFrame extends JFrame implements ActionListener {

    private JMenuBar barra_menus;
    // Menú 1:
    private JMenu menu1, menu2;
    private JMenu menu1_item1;// un submenu
    private JMenuItem
            menu1_item1_subitem1,
            menu1_item1_subitem2,
            menu1_item1_subitem3,
            menu1_item1_subitem4;
    private JMenuItem menu1_item2;
    // Menu 2:
    private JMenu menu2_item1;// submenu
    private JMenuItem
            menu2_item1_subitem1,
            menu2_item1_subitem2,
            menu2_item1_subitem3,
            menu2_item1_subitem4;
    private JMenuItem menu2_item2,menu2_item3,menu2_item4,menu2_item6;
    private JMenu menu2_item5;// submenu
    private JMenuItem menu2_item5_subitem1;

    // Método main: donde inicia el programa
    public static void main(String[] args) {
        MiPrimerFrame interfaz = new MiPrimerFrame();// Crea la interfaz
        interfaz.initComponents();// Crea los componentes de la interfaz
        interfaz.setVisible(true);// Visibiliza interfaz
    }

    // Constructor que debe tener el mismo nombre de la class
    public MiPrimerFrame() {
        setLayout(null);// permite colocar las coordenadas de la interfaz manualmente
        setBounds(0,0,500,500);// posición X, posición Y, Ancho, Alto
        setLocationRelativeTo(null);// Interfaz en el centro de la pantalla
        setResizable(false);// Establece que no se pueda modificar el tamaño de la interfaz de forma manual
        setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);// Termina el programa al cerrar la interfaz
        //setSize(640,480);// Cambia el tamaño de la interfaz
    }

    // Crear componentes de la interfaz
    public void initComponents() {

    barra_menus = new JMenuBar();
    setJMenuBar(barra_menus);// se aplica automaticamente en la parte superior de la interfaz
    //barra_menus.setBackground(new java.awt.Color(255,0,0));// R,G,B (colorea la barra de menus)
    //
        menu2 = new JMenu("Mas tutoriales");
    barra_menus.add(menu2);// añade el menu2 a la barra de menus
        //
            menu2_item1 = new JMenu("Programacion Orientada a Objetos");
            menu2.add(menu2_item1);
            //
                menu2_item1_subitem1 = new JMenuItem("Main");
        menu2_item1.add(menu2_item1_subitem1);
        menu2_item1_subitem1.addActionListener(this);
```

```
            menu2_item1_subitem2 = new JMenuItem("Herencia (extends)");
    menu2_item1.add(menu2_item1_subitem2);
    menu2_item1_subitem2.addActionListener(this);

            menu2_item1_subitem3 = new JMenuItem("Polimorfismo (abstract)");
    menu2_item1.add(menu2_item1_subitem3);
    menu2_item1_subitem3.addActionListener(this);

            menu2_item1_subitem4 = new JMenuItem("Palabra reservada 'super'");
    menu2_item1.add(menu2_item1_subitem4);
    menu2_item1_subitem4.addActionListener(this);
        //
        menu2_item2 = new JMenuItem("File y PrintWriter");
        menu2.add(menu2_item2);
        menu2_item2.addActionListener(this);

        menu2_item3 = new JMenuItem("Ambitos de variables");
        menu2.add(menu2_item3);
        menu2_item3.addActionListener(this);

        menu2_item4 = new JMenuItem("Excepciones (try-catch-finally)");
        menu2.add(menu2_item4);
        menu2_item4.addActionListener(this);

        menu2_item5 = new JMenu("Threads / Hilos");
        menu2.add(menu2_item5);
        //
            menu2_item5_subitem1 = new JMenuItem("Main");
    menu2_item5.add(menu2_item5_subitem1);
    menu2_item5_subitem1.addActionListener(this);
        //
        menu2_item6 = new JMenuItem("Excepciones (try-catch-finally)");
        menu2.add(menu2_item6);
        menu2_item6.addActionListener(this);
//


menu1 = new JMenu("Opciones");
barra_menus.add(menu1);// añade el menu1 a la barra de menus
//
        menu1_item1 = new JMenu("Color del fondo");
        menu1.add(menu1_item1);
        //
            menu1_item1_subitem1 = new JMenuItem("Rojo");
    menu1_item1.add(menu1_item1_subitem1);
    menu1_item1_subitem1.addActionListener(this);

            menu1_item1_subitem2 = new JMenuItem("Verde");
    menu1_item1.add(menu1_item1_subitem2);
    menu1_item1_subitem2.addActionListener(this);

    menu1_item1_subitem3 = new JMenuItem("Azul");
    menu1_item1.add(menu1_item1_subitem3);
    menu1_item1_subitem3.addActionListener(this);

    menu1_item1_subitem4 = new JMenuItem("Default");
    menu1_item1.add(menu1_item1_subitem4);
    menu1_item1_subitem4.addActionListener(this);
        //
        menu1_item2 = new JMenuItem("Empaquetar");
        menu1.add(menu1_item2);
        menu1_item2.addActionListener(this);
//
}
```

```
    // Eventos del menú
    @Override
    public void actionPerformed(ActionEvent ae) {

        if(ae.getSource() == menu1_item1_subitem1) {
            Container fondo = this.getContentPane();// obtiene el fondo de la interfaz, incluso si se modifica el tamaño
            fondo.setBackground(new Color(255,0,0));
            System.out.println("Pintaste el fondo de rojo.");
        }
        if(ae.getSource() == menu1_item1_subitem2) {
            Container fondo = this.getContentPane();// obtiene el fondo de la interfaz, incluso si se modifica el tamaño
            fondo.setBackground(new Color(0,255,0));
            System.out.println("Pintaste el fondo de verde.");
        }
        if(ae.getSource() == menu1_item1_subitem3) {
            Container fondo = this.getContentPane();// obtiene el fondo de la interfaz, incluso si se modifica el tamaño
            fondo.setBackground(new Color(0,0,255));
            System.out.println("Pintaste el fondo de azul.");
        }
        if(ae.getSource() == menu1_item1_subitem4) {
            Container fondo = this.getContentPane();// obtiene el fondo de la interfaz, incluso si se modifica el tamaño
            fondo.setBackground(null);
            System.out.println("Pintaste el fondo de default.");
        }
        if(ae.getSource() == menu1_item2) {
            Container fondo = this.getContentPane();// obtiene el fondo de la interfaz, incluso si se modifica el tamaño
            fondo.setBackground(new Color(244,244,244));
            System.out.println("Ahora sabes empaquetar");
        }

       if(ae.getSource() == menu2_item1_subitem1) {// Programacion Orientada a Objetos (base)
            System.out.println("¿Ya viste este tutorial?");
        }
        if(ae.getSource() == menu2_item1_subitem2) {// Herencia (extends)
            System.out.println("¿Ya viste este tutorial?");
        }
        if(ae.getSource() == menu2_item1_subitem3) {// Polimorfismo (abstract)
            System.out.println("¿Ya viste este tutorial?");
        }
        if(ae.getSource() == menu2_item1_subitem4) {// palabra reservada super
            System.out.println("¿Ya viste este tutorial?");
        }
        if(ae.getSource() == menu2_item2) {// Files y PrintWriter
            System.out.println("¿Ya viste este tutorial?");
        }
        if(ae.getSource() == menu2_item3) {// Ámbitos de variables
            System.out.println("¿Ya viste este tutorial?");
        }
        if(ae.getSource() == menu2_item4) {// Excepciones (try-catch-finally)
            System.out.println("¿Ya viste este tutorial?");
        }
        if(ae.getSource() == menu2_item5_subitem1) {// Threads / Hilos
            System.out.println("¿Ya viste este tutorial?");
        }
        if(ae.getSource() == menu2_item6) {// Randomize
            System.out.println("¿Ya viste este tutorial?");
        }
    }

}
```

Fondo de la interfaz

Se puede modificar el fondo de la interfaz gráfica mediante las siguientes líneas

```
Container fondo = this.getContentPane();// obtiene el fondo de la interfaz
fondo.setBackground(new Color(244,244,244));
```

Ícono de la interfaz

Para agregar un ícono en nuestra interfaz gráfica, se procede de la siguiente manera:

1. Establecer una carpeta donde se ubicará el proyecto.
   Esta carpeta es donde se ubicarán los archivos .java y los archivos .class

2. Dentro de la carpeta del proyecto, crear una carpeta llamada **images**.

3. Ubicar dentro de la carpeta **images**, un archivo de imagen con extensión .png

4. Una vez establecido todo, se debe utilizar la siguiente línea de código en el procedimiento de creación de los componentes de la interfaz.

```
setIconImage( new ImageIcon( getClass().getResource("images/icon.png")) ).getImage());
```

Imágenes

Para agregar una imagen a nuestra interfaz gráfica, se debe utilizar un JLabel.

Al igual que para agregar un ícono, se debe crear la carpeta **images** con un archivo de imagen en su interior, por lo tanto, los primeros 3 pasos son idénticos.

4. Crear un ImageIcon:

```
ImageIcon imagen = new ImageIcon("images/mi-imagen.png");
```

5. Crear una etiqueta (JLabel) e inicializarlo con la imagen.

```
private JLabel etiqueta;
```

```
etiqueta = new JLabel(imagen);
etiqueta.setBounds(25,15,300,150);// X, Y, Ancho, Alto
add(etiqueta);
```

Conexión entre interfaces gráficas

Se puede pasar de una interfaz gráfica a otra mediante el ocultamiento de la interfaz actual y creación de otra interfaz tal cual se hizo con la primera.

```
// Procedimiento de cambiar de interfaz
private void cambiarInterfaz() {
    OtraInterfaz frame = new OtraInterfaz();// Crea interfaz
    frame.initComponents();// Crea los componentes de la otra interfaz
    frame.setVisible(true);// Muestra la otra interfaz al usuario

    this.setVisible(false);// Oculta la interfaz actual
}
```

Igual al método main de la primer interfaz

En la class creada con la otra interfaz, no debe existir el método main.

```
import javax.swing.*;

public class OtraInterfaz extends JFrame {

    // Constructor de igual nombre que la class
    public OtraInterfaz() {
        setLayout(null);
        setBounds(0,0,200,200);
        setLocationRelativeTo(null);
        setResizable(false);
        setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
    }

    // Crea los elementos de la interfaz
    public void initComponents() {
        ...
    }
}
```

NO EXISTE MÉTODO MAIN

Si se desean pasar datos de una interfaz a otra, se puede realizar una instancia de clases y usar variables estáticas.

```
// Declaración de atributos de la class
public static String texto1 = "Soy un texto de la primera interfaz";
```

```
// Crea los elementos de la interfaz
public void initComponents() {

    // Etiqueta
    PrimeraInterfaz interfazAnterior = new PrimeraInterfaz();
    String textoDeLaPrimerInterfaz = interfazAnterior.textoEstatico;

    etiqueta = new JLabel(textoDeLaPrimerInterfaz);
```

Si la interfaz anterior no se utilizará más y desea liberar espacio en memoria, en vez de utilizar **.setVisible** puede utilizar **.disponse()**

```
this.dispose();// Destruye la interfaz actual
```

Dialogs

Se puede crear una ventana de diálogo simple mediante la línea de código utilizando la biblioteca: `import javax.swing.JOptionPane;`

- Información

```
JOptionPane.showMessageDialog(null, "Ventana de información.","Info", JOptionPane.INFORMATION_MESSAGE);
```

- Advertencia

```
JOptionPane.showMessageDialog(null, "Ventana de advertencia.","Warning", JOptionPane.WARNING_MESSAGE);
```

- Error

```
JOptionPane.showMessageDialog(null, "Ventana de error","Error", JOptionPane.ERROR_MESSAGE);
```

Función para establecer un limite de caracteres en una caja de texto

```
public static class JTextFieldLimit extends javax.swing.text.PlainDocument {
    private int limit;
    JTextFieldLimit(int limit) {
        super();
        this.limit = limit;
    }
    @Override
    public void insertString(int offset, String str, javax.swing.text.AttributeSet attr)
    throws javax.swing.text.BadLocationException {
        if(str == null) return;
        if((getLength() + str.length()) <= limit) {
            super.insertString(offset, str, attr);
        }
    }
}
```

Se aplica utilizando el método: `jFieldText.setDocument(new JTextFieldLimit(MAX_CHARACTERS));`

Funcion para crear una barra de progreso

```
    public static class ProgressBar extends JFrame {
       private NombreClass frame;
       private javax.swing.JLabel encryptingTitle;
       private javax.swing.JProgressBar pb;
       private String title = "Title";

       public ProgressBar(NombreClass aThis) {
           this.frame = aThis;// obtiene el ID de la class desde la cual se lo llama

           this.setResizable(false);
           t = new javax.swing.JLabel();
           pb = new javax.swing.JProgressBar();
           //setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
           setFocusable(false);
           setFocusableWindowState(false);
           t.setFont(new java.awt.Font("Arial", 0, 12)); // NOI18N
           t.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
           t.setText("Encrypting...");
           t.setMinimumSize(new java.awt.Dimension(129, 14));


           javax.swing.GroupLayout layout = new javax.swing.GroupLayout(getContentPane());
           getContentPane().setLayout(layout);
           layout.setHorizontalGroup(
           layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
           .addGroup(layout.createSequentialGroup()
               .addContainerGap()
               .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                   .addGroup(layout.createSequentialGroup()
                       .addComponent(t, javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE)
                       .addGap(17, 17, 17))
                   .addGroup(layout.createSequentialGroup()
                       .addComponent(pb, javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE)
                       .addGap(10, 10, 10)))
               .addContainerGap())
           );
           layout.setVerticalGroup(
           layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
           .addGroup(layout.createSequentialGroup()
               .addContainerGap()
               .addComponent(t, javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)
               .addGap(18, 18, 18)
               .addComponent(pb, javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)
               .addContainerGap(javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))
           );
           pb.setStringPainted(true);
           pb.setValue(0);
           pb.setString("100%");
           pack();
           this.setTitle(title);
       }

       public void showFrame() {
           this.setVisible(true);
           this.setLocationRelativeTo(null);
           this.setAlwaysOnTop(true);
       }

       public void setPercent(int min, int max) {
           if(min < 0) min = 0;
           int p = min*100/max;
           pb.setValue(p);
           pb.setString(p+"%");
       }

       public void closeFrame() {
           this.setVisible(false);
       }
```

Programación Orientada a Objetos

Paradigma de programación orientado a mejorar la claridad, calidad y tiempo de desarrollo de un programa de computadora utilizando solamente **subrutinas** y tres **tipos de estructuras**:

- **Secuencia**
- **Selección** (*if, switch*)
- **Iteración** (*for, while, do-while*)

```java
public class Perro {

    public Perro() {
    }

    // Estado: características
    String raza = "Dálmata";
    int edad = 5;
    int fuerza = 11;

    // Comportamiento: funciones
    public void ladrar() {
        ...
    }

    public void dormir() {
        ...
    }
}
```

Se trata de una manera de diseñar y desarrollar software que trata de imitar la realidad tomando algunos conceptos esenciales de ella.

- **Objeto**
  - Identidad — nombre del objeto.
  - Estado — características del objeto.
  - Comportamiento — métodos y funciones.

En la programación orientada a objetos, se pueden utilizar variables que se encuentren en otras clases. Para ello se debe crear una instancia de clases.

```java
// Identificador: nombre del objeto
Perro nombreDelPerro = new Perro();
```

Instancia de clases

Una instancia de clases es cuando hacemos que dos clases interactúen entre sí.

```java
Culculadora casio = new Culculadora(operando1, operando2);
```

```java
public class Culculadora {

    private int operando1;
    private int operando2;

    public Culculadora(int operando1, int operando2) {
        this.operando1 = operando1;
        this.operando2 = operando2;
    }
```

Al acceder a variables de instancia de una clase, podemos encontrarnos con variables que se llaman igual que en la clase de donde estamos accediendo.

Cuando esto sucede, se puede usar la palabra **this** ya que esta indica si se están utilizando las variables de instancia o las variables de la clase desde la cual estamos accediendo.

Componentes de una class

Las clases están compuestas por:

- **Atributos** variables pertenecientes a la class.
- **Métodos** funciones y procedimientos de la class.

Encapsulamiento

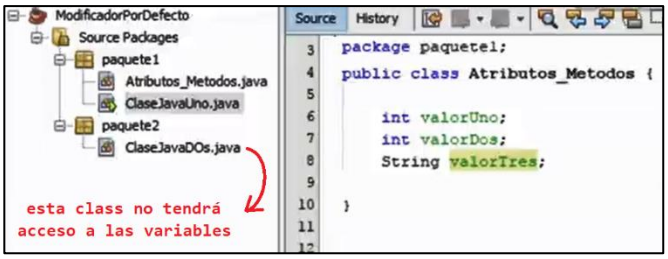
Consiste en controlar el acceso a los datos, que conforman un objeto o instancia de clase. Es decir, indica cuáles componentes de una class pueden ser accedidos e incluso modificados.

Con el encapsulamiento se pueden ocultar los atributos y métodos de una clase para evitar que sean modificados desde otra clase para evitar cambios imprevistos e incontrolados que puedan provocar un cambio en la funcionalidad de dicha clase.

```java
public class Perro {

    // private
    private int energia = 0;
    private int edad = 0;
    private int fuerza = 0;

    // public
    public void ladrar() {
        fuerza++;
    }

    // protected
    protected void crecer() {
        edad++;
    }

    // default
    void dormir() {
        energia++;
    }
}
```

Para el encapsulamiento, se utilizan **modificadores de acceso**.

- **public** permite el acceso a cualquier otra class.
- **private** sólo tiene acceso la class a la cual pertenece.
- **protected** tienen acceso las class heredadas.
- **package-private** (**default**)
  sólo tienen acceso las class ubicadas en el mismo subdirectorio o paquete.



Método getter y setter

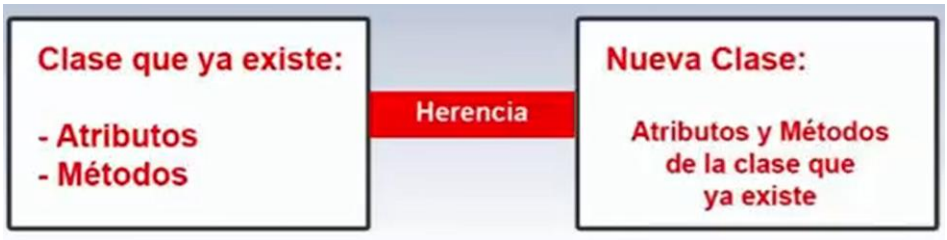
Son métodos públicos de una class que sirven para otorgar acceso a algunos atributos de dicha class sin eliminar el encapsulamiento.

- **set** procedimiento que permite asignar un valor a un atributo privado de la class.
- **get** función que permite obtener el valor de un atributo privado de la class.

```java
private int fuerza = 0;

public void setFuerza(int nivelFuerza) {
    fuerza = nivelFuerza;
}

public int getFuerza() {
    return fuerza;
}
```

Herencia

Procedimiento utilizado para reutilizar código cuando creamos nuevas clases, ya que permite utilizar atributos (variables) y métodos (funciones y procedimientos) de una clase que ya hemos creado anteriormente y colocarlos dentro de una nueva clase.



- **Clase padre / base**

  Es la clase que se debe crear primero, donde se encuentra escrito el código que contiene a los atributos y métodos que se van a reutilizar.

- **Clase hija / derivada**

  Es la clase que se creará luego, donde se reutilizarán los atributos y métodos que se crearon en la clase padre / base sin necesidad de volver a escribir el mismo código para poder utilizarlos.

  La clase hija sólo puede heredar los atributos y métodos de una única clase padre.

Es buena práctica establecer el modificador de acceso **protected** sólo a los atributos de la clase padre / base, mientras que a los métodos se les establece como **public**.

La palabra **extends** indica que se quiere heredar el código de una clase.

```
public class ClaseHija extends ClasePadre {
```

CLASE PADRE

```java
package operaciones;

import java.util.Scanner;

public class ClasePadre {

    protected int valor1;
    protected int valor2;
    protected int resultado;

    private Scanner entrada = new Scanner(System.in);

    // Este método pide los valores al usuario
    public void PedirDatos() {
        System.out.print("Dame el primer valor: ");
        valor1 = entrada.nextInt();

        System.out.print("Dame el segundo valor: ");
        valor2 = entrada.nextInt();
    }

    // Este método muestra el resultado
    public void MostrarResultado() {
        System.out.println("El resultado es " + resultado);
    }
}
```

Se pueden utilizar las variables de la clase padre sin necesidad de declararlas en las clases hijas.

HERENCIA

CLASE HIJA

```java
package operaciones;

public class ClaseHija_Suma extends ClasePadre {

    public void Sumar() {
        resultado = valor1 + valor2;
    }
}
```

CLASE HIJA

```java
package operaciones;

public class ClaseHija_Resta extends ClasePadre {

    public void Restar() {
        resultado = valor1 - valor2;
    }
}
```

CLASE PRINCIPAL (donde se encuentra el método main)

```java
package main;

import operaciones.ClaseHija_Suma;
import operaciones.ClaseHija_Resta;

public class ClasePrincipal {

    public static void main(String[] args) {

        // Instancia de clases
        ClaseHija_Suma instanciaSuma = new ClaseHija_Suma();

        instanciaSuma.PedirDatos();
        instanciaSuma.Sumar();
        instanciaSuma.MostrarResultado();
    }
}
```

No se necesita importar la clase padre porque su código ya fue heredado a las clases hijas

Se pueden utilizar los métodos declarados en la clase padre sin necesidad de declararlos en la clase hija

Polimorfismo

Consiste en hacer que un método se comunique con clases diferentes y, dependiendo de la clase con la que se esté comunicando, su comportamiento será completamente diferente.

Para hacer uso del polimorfismo, es necesario hacer uso de la herencia (**extends**).

La palabra reservada **abstact** indica que se trabajará con polimorfismo.

- En la clase padre, se declara el método sin la parte lógica.

```
package operaciones;

public abstract class ClasePadre_Operaciones{

    public abstract void  Operaciones();
}
```

- En la clase hija, se **sobrescribe el método** utilizando la palabra @**Override** y se agrega la parte lógica del método para esa clase.

```
@Override
public void Operaciones(){
    . . .
}
```

De esta manera, se puede utilizar el mismo método en distintas clases, sin embargo en cada clase tendrá una funcionalidad diferente.

Así, cuando se realice una instancia de clases, se podrán utilizar los métodos de la clase padre definidos con su parte lógica y métodos abstractos definidos en la clase hija (la cual es a la que estaremos comunicándonos mediante la instancia de clases).

```java
package operaciones;

import java.util.Scanner;

public class ClasePadre {

    protected int valor1;
    protected int valor2;
    protected int resultado;

    private Scanner entrada = new Scanner(System.in);

    // Este método pide los valores al usuario
    public void PedirDatos() {
        System.out.print("Dame el primer valor: ");
        valor1 = entrada.nextInt();

        System.out.print("Dame el segundo valor: ");
        valor2 = entrada.nextInt();
    }

    // Este método muestra el resultado
    public void MostrarResultado() {
        System.out.println("El resultado es " + resultado);
    }


    public abstract void Operaciones();

}
```


```java
package operaciones;

public class ClaseHija_Suma extends ClasePadre {

    @Override
    public void Operaciones() {
        resultado = valor1 + valor2;
    }
}
```

```java
package operaciones;

public class ClaseHija_Resta extends ClasePadre {

    @Override
    public void Operaciones() {
        resultado = valor1 - valor2;
    }
}
```

```java
package main;

import operaciones.ClaseHija_Suma;
import operaciones.ClaseHija_Resta;

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

        // Instancia de clases
        ClaseHija_Suma instancia = new ClaseHija_Suma();

        instancia.PedirDatos();

        instancia.Operaciones();

        instancia.MostrarResultado();
    }
}
```

En este caso, al instanciarse la class ClaseHija_Suma, el método Operaciones realizará una suma.

Si se hubiera instanciado ClaseHija_Resta, el método Operaciones hubiera realizado una resta.

Palabra reservada **super**

Se utiliza para acceder a un elemento de la clase padre en una herencia.

El uso más común es el de invocar al constructor de la clase padre.

1. Crear un método en la clase padre.

```java
public class Padre {
    public void Saludar() {
        System.out.println("Hola, soy la clase padre.");
    }
}
```

2. Crear un método en la clase hija, el cual llamará al método de la clase padre mediante la palabra reservada **super**.

```java
public class Hija extends Padre {
    public void SaludoDeMiPadre() {
        super.Saludar();
    }
}
```

Hilos

Es un flujo de control que permite tener múltiples procesos de forma simultánea.

Con ayuda del **thread**, se puede tener más de un proceso ejecutándose al mismo tiempo en vez de tener que esperar a que finalice uno para que inicie el siguiente.

Para que funcionen correctamente, es necesario crear todos los hilos y posteriormente iniciarlos todos juntos.

Existen dos maneras de crear un hilo:

- Por herencia de la clase **Thread**

  Se debe sobrescribir el método **run** para ejecutar el hilo, el cual es un método de la clase heredada.

```java
public class Hilo extends Thread {

    @Override
    public void run() {
        . . .
    }
}
```

Desde otra class se debe crear una instancia de clases.

```java
Hilo hilo = new Hilo();
```

Luego, una vez creadas todas las instancias de clase para todos los hilos que deseemos, se los debe iniciar.

```java
hilo.start();
```

- Por implementación de la interfaz **Runneable**

  Se utiliza **implements** sobre la class en que deseamos tener el hilo, luego sobrescribir el método **run** para ejecutar el hilo.

```java
public class Hilo implements Runnable{

    @Override
    public void run() {
        . . .
    }
}
```

Desde otra class se debe crear un objeto de tipo **Thread** con la class del Hilo como parámetro **new**.

```java
Thread hilo = new Thread(new Hilo());
```

Luego, una vez creados todos los objetos de los hilos, se los debe iniciar.

```java
hilo.start();
```

- Dos procesos sin threads: se envía siempre el Proceso1 primero y el Proceso2 después.

```
for(int i = 0; i <= 5; i++) {
    System.out.println("Proceso 1 - " + i);
}

System.out.println("");

for(int j = 0; j <= 5; j++) {
    System.out.println("Proceso 2 - " + j);
}
```

```
run:
Proceso 1
Proceso 1
Proceso 1
Proceso 1
Proceso 1
Proceso 1

Proceso 2
Proceso 2
Proceso 2
Proceso 2
Proceso 2
Proceso 2
BUILD SUCCESSFUL (total time: 0 seconds)
```

- Dos procesos con threads: no siempre se ejecutan los procesos en el mismo orden.

```
public class Proceso1 extends Thread {
    @Override
    public void run() {
        for(int i = 0; i <= 5; i++) {
            System.out.println("Proceso 1 - " + i);
        }
    }
}
```

```
public class Proceso2 extends Thread {
    @Override
    public void run() {
        for(int i = 0; i <= 5; i++) {
            System.out.println("Proceso 2 - " + i);
        }
    }
}
```

```
Proceso1 hilo1 = new Proceso1();
Proceso2 hilo2 = new Proceso2();

hilo1.start();
hilo2.start();
```

```
run:
Proceso 2
Proceso 2
Proceso 2
Proceso 2
Proceso 2
Proceso 2
Proceso 1
Proceso 1
Proceso 1
Proceso 1
Proceso 1
Proceso 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

Pasaje de parámetros a un Hilo

Para ello se necesita declarar una variable de ámbito global (sin inicializar) en la class donde está alojado el **thread**, la cual contendrá el valor que pasaremos por parámetros.

Para que la variable creada reciba el parámetro, se debe crear un procedimiento el cual asigne dicho parámetro a la variable.

```
public class Hilo extends Thread {

    // Variable global sin inicializar
    int parametro;

    @Override
    public void run() {
        for(int i = 0; i <= parametro; i++) {
            System.out.println("Proceso: " + i);
        }
    }

    // Procedimiento que asignará el valor
    public void asignarParametro(int valor) {
        this.parametro = valor;
    }
}
```

Ahora, se puede pasar el parámetro por medio de la instancia de clases.

```
Hilo hiloParametrizado = new Hilo();

hiloParametrizado.asignarParametro(parametro);
```

No hay que olvidar que, para que el hilo se ejecute, hay que iniciarlo.

```
hiloParametrizado.start();
```

Estados de un Hilo

- **New** El thread fue creado pero no iniciado (start no fue ejecutado).
- **Ejecutable** Cuando el thread fue iniciado (start fue ejecutado) y comienza el método **run**.
- **Bloqueado** El hilo se encuentra en ejecución pero existe una tarea o actividad del mismo hilo que impide que continúe.
    - **sleep**(*milisegundos*) El hilo se "duerme" durante los segundos establecidos. Requiere **try – catch**.
- **Muerto** Se produce cuando su método **run** termina normalmente o cuando existe una instrucción que obligue al hilo a finalizar sin haber concluido su tarea por completo.

Sincronización de hilos

Permite controlar el tiempo y forma de ejecución de varios hilos ejecutándose de manera simultánea. Su finalidad es evitar que un hilo entorpezca a otro hilo al momento de estar ejecutando sus respectivas tareas o bien para establecer un orden de ejecución.

Para lograrlo, se necesita utilizar el método **sleep** tras ser iniciado un hilo, estableciendo un estado durmiente de unos pocos milisegundos.

```java
public class ClasePrincipal {

    public static void main(String[] args) {

        HiloSincronizado hilo1 = new HiloSincronizado("G", 5);
        HiloSincronizado hilo2 = new HiloSincronizado("o", 5);
        HiloSincronizado hilo3 = new HiloSincronizado("l\n",5);

        hilo1.start();
        try {
            hilo1.sleep(10);
        } catch(InterruptedException e) {
            System.out.println("Error en el hilo1: " + e.toString());
        }

        hilo2.start();
        try {
            hilo2.sleep(10);
        } catch(InterruptedException e) {
            System.out.println("Error en el hilo2: " + e.toString());
        }

        hilo3.start();
        try {
            hilo3.sleep(10);
        } catch(InterruptedException e) {
            System.out.println("Error en el hilo3: " + e.toString());
        }
    }
}
```

```java
public class HiloSincronizado extends Thread {

    // Variables globales (sin inicializar)
    private String texto;
    private int iteraciones;

    // Constructor (donde inicia al ser instanciado)
    public HiloSincronizado(String texto, int iteraciones) {
        this.texto = texto;
        this.iteraciones = iteraciones;
    }

    @Override
    public void run() {
        for(int i = 0; i <= iteraciones; i++) {
            System.out.println(texto);
        }
    }
}
```

```
run:
Gol
Gol
Gol
Gol
Gol
BUILD SUCCESSFUL (total time: 0 seconds)
```

Listener

Los escuchadores o listener son formas en que una clase puede comunicarse con otra a tiempo real, por lo general por medio de hilos (threads).

Para ello, se utiliza en la clase creada el siguiente bloque de código:

OtraActivity.java

```
private OnListenerResult onListenerResult;
public interface OnListenerResult {
    void funcionEnviada1(int unParametroEnviado);
    void funcionEnviada2();
}
public void setOnListenerResult(OnListenerResult listener) {
    this.onListenerResult = listener;
}
```

Luego, dentro de esta misma clase creada, se debe utilizar la siguiente función para que el listener se active

OtraActivity.java

```
onListenerResult.funcionEnviada1(1234);
```

Finalmente, en la activity donde se quiere recibir dicha información, se debe utilizar el listener asignado a la clase o activity creada.

MainActivity.java

```
final MainActivity mainActivity = this;
OtraActivity otraActivity = new OtraActivity(...);

otraActivity.setOnDialogResult(new BasicDialog.OnDialogResult() {
    @Override
    public void funcionEnviada1(int unParametroEnviado) {
        // Acciones que realizará esta función cuando sea llamada
        // El parámetro recibido será el mismo que se especificó antes
    }

    @Override
    public void funcionEnviada2() {
        // Acciones que realizará esta función cuando sea llamada
    }
});
```

Math

Es una class nativa de Java que permite realizar operaciones matemáticas. Se trabaja con el tipo de dato **double**, números reales de hasta 15 decimales.

La class Math se puede utilizar sin necesidad de importarla.

15 decimales (double)

```
System.out.println("Número PI = " + Math.PI);// 3.141592653589793
System.out.println("Número e  = " + Math.E);//  2.718281828459045
```

- Raíces cuadradas

```
Math.sqrt(numero);
```

- Conversión de ángulos

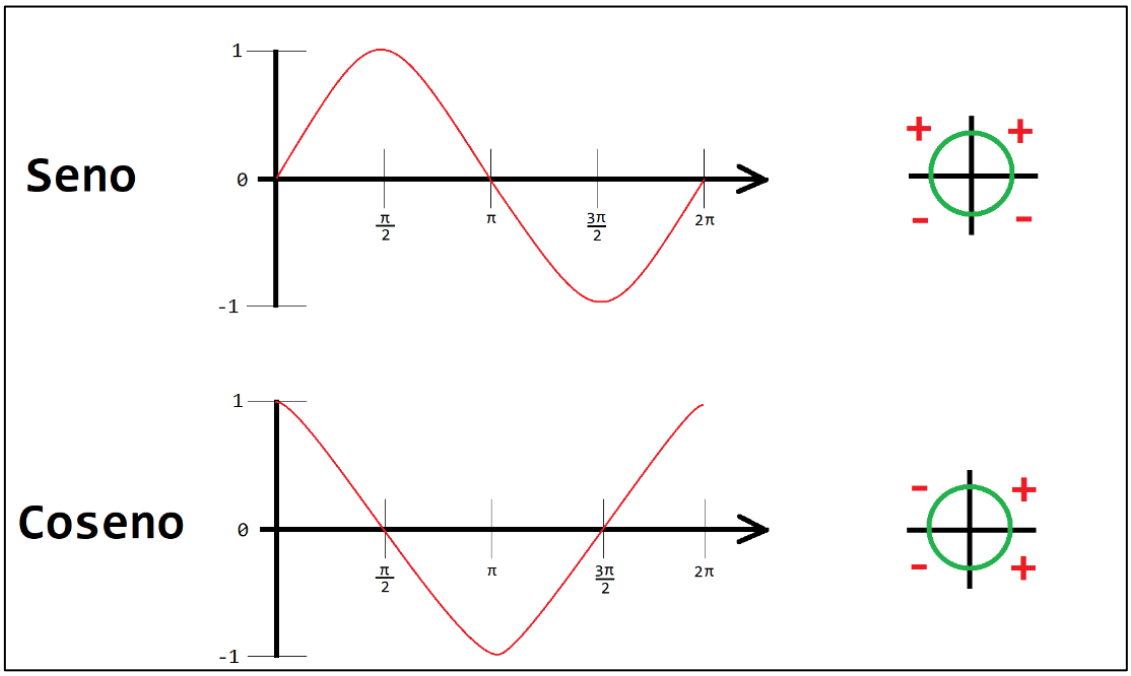
```
// Conversión de grados en radianes
double radianes = Math.toRadians(angulo_en_grados);
```

```
// Conversión de radianes en grados
double grados = Math.toDegrees(angulo_en_radianes);
```

- Funciones trigonométricas: **utiliza ángulos en radianes**.

$$f(x) = A \cdot sen(B\ x - C) + D$$

→ Amplitud (A) modifica altitud del gráfico.
→ Frecuencia (B) modifica el período del gráfico (gráficos cada 2 π).
→ Desplazamiento horizontal (C)
→ Desplazamiento vertical (D)



1 π = 180°

$$Cos\ (\alpha) = \frac{cateto\ adyacente}{hipotenusa}$$

```
Math.cos(radianes);
```

$$arco\ Cos(\alpha) = \alpha$$

```
Math.acos(radianes);
```

$$Sen\ (\alpha) = \frac{cateto\ opuesto}{hipotenusa}$$

```
Math.sin(radianes);
```

$$arco\ Sen(\alpha) = \alpha$$

```
Math.asin(radianes);
```

$$Tan\ (\alpha) = \frac{cateto\ opuesto}{cateto\ adyacente}$$

```
Math.tan(radianes);
```

$$arco\ Tan(\alpha) = \alpha$$

```
Math.atan(radianes);
```

$$Cot\ (\alpha) = \frac{1}{Tan(\alpha)} \qquad Sec\ (\alpha) = \frac{1}{Cos(\alpha)} \qquad Csc\ (\alpha) = \frac{1}{Sen(\alpha)}$$

$$Cos^2(\theta) + Sen^2(\theta) = 1$$

$$Sen(\alpha + \beta) = Sen(\alpha) \cdot Cos(\beta) + Cos(\alpha) \cdot Sen(\beta)$$

$$Cos(\alpha + \beta) = Cos(\alpha) \cdot Sen(\beta) - Sen(\alpha) \cdot Sen(\beta)$$

Control de decimales

❖ DecimalFormat

Se puede recurrir a esta biblioteca para disminuir la cantidad de decimales mostrados.

```
import java.text.DecimalFormat;
```

Luego, se crea una instancia de clases donde, como parámetro, se establecerá la cantidad de decimales que deseamos mostrar con el siguiente formato (la cantidad de `0` es la cantidad de decimales):

```
DecimalFormat df = new DecimalFormat("#.00");
```

Finalmente, se utiliza el método **.format**

```
df.format(numero_con_decimales);
```

se reemplaza el punto por una coma

❖ String format

Si se desea expresar el número en un string, se puede utilizar este método para controlar los decimales a mostrar (indicado con el ***número***).

```
String.format("%.2f", numero_con_decimales);
```

se reemplaza el punto por una coma

❖ Math round

Teniendo un número con decimales, se aplica el casting **(double)** ya que la clase Math trabaja únicamente con este tipo de datos.

```
(double)Math.round(numero_con_decimales * 100d) / 100;
```

Donde la cantidad de ceros es la cantidad de decimales que se mostrará (debe usarse la misma cantidad tanto en la multiplicación como en la división.

❖ BigDecimal

Se puede recurrir a esta clase:

```
import java.math.BigDecimal;
import java.math.RoundingMode;
```

Luego, mediante una instancia de clase con el número como parámetro, se aplica el método de redondeo deseado y se aplica sobre una variable.

```
BigDecimal bd = new BigDecimal(numero_con_decimales);

bd = bd.setScale(3, RoundingMode.HALF_UP);

double numero_redondeado = bd.doubleValue();
```

Números aleatorios

Para crear un número aleatorio se puede utilizar la clase Math con el método:

```
numeroAleatorio = (int) (Math.random() * numero_maximo);
```

De esta manera, se genera un número desde el **0** hasta el **numero_maximo**.

Otra manera de generar un número aleatorio es con la class Random.

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

Luego, mediante una instancia de clases, se puede generar el número aleatorio de la siguiente manera:

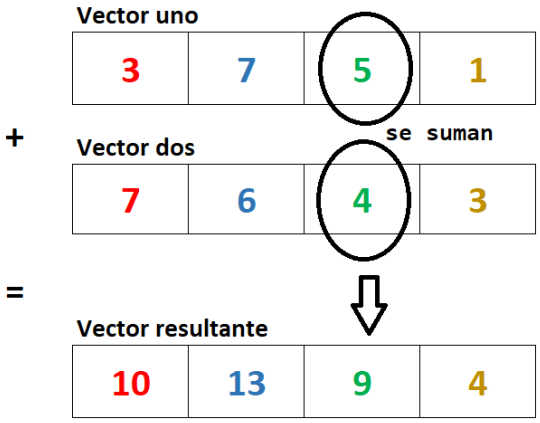
```
Random r = new Random();
numeroAleatorio = (int) (r.nextDouble() * numero_maximo);
```

Vectores

Suma (y resta)

La suma de vectores (traslación) se realiza sumando elemento a elemento del vector.

Los vectores a operar deben tener la misma dimensión.

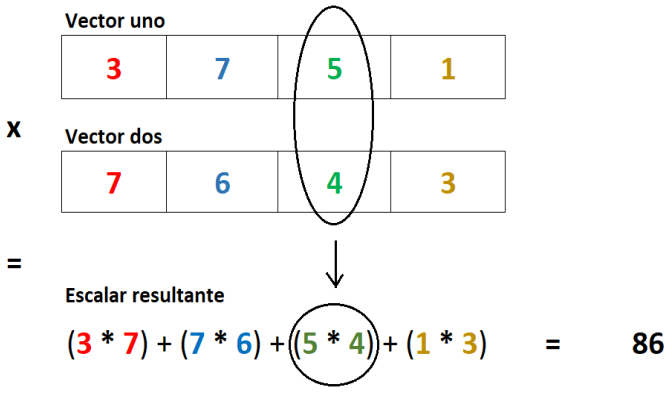



```java
int vectorUno[] = new int[4];
int vectorDos[] = new int[4];
int vectorResultante[] = new int[4];

// Llenado del primer vector
for(int i = 0; i < vectorUno.length; i++) {
    vectorUno[i] = (int)(Math.random() * 5);
}

// Llenado del segundo vector
for(int i = 0; i < vectorDos.length; i++) {
    vectorDos[i] = (int)(Math.random() * 5);
}

// Suma (y resta) de vectores
for(int i = 0; i < vectorUno.length; i++) {
    vectorResultante[i] = vectorUno[i] + vectorDos[i];
}
```

Multiplicación

Para multiplicar vectores, se realiza la multiplicación entre cada elemento de los vectores y, al resultado de dicha multiplicación, se la suma al resultado de la multiplicación del siguiente par de elementos.

El resultado de la multiplicación de dos vectores, es un número escalar.



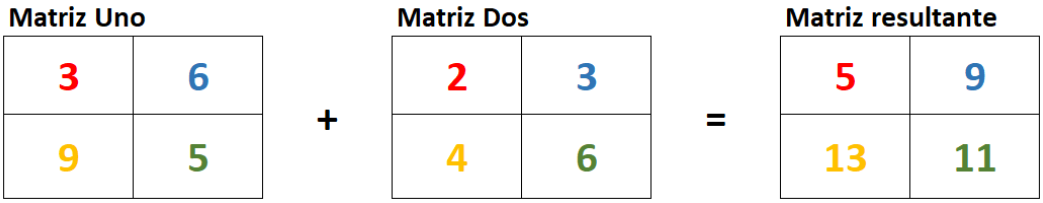

```java
// Multiplicación (producto interno)
for(int i = 0; i < vectorUno.length; i++) {
    escalarResultante += vectorUno[i] * vectorDos[i];
}
```

Matrices

Suma (y resta)

La suma de matrices solo se puede realizar entre matrices de la misma dimensión, es decir, deben tener la misma cantidad de filas y columnas. Así mismo, la matriz resultante mantendrá las mismas dimensiones.

La suma se efectúa en cada uno de los elementos correspondiente a cada matriz en su posición actual:



Se sumará matrizUno**[0,0]** con matrizDos**[0,0]** y el resultado se asignará en matrizResultante**[0,0]**.

filas
columnas

```
int matrizUno[][] = new int[3][3];
int matrizDos[][] = new int[3][3];
int matrizResultante[][] = new int[3][3];

// Llenado de la primer matriz
for(int i = 0; i < matrizUno.length; i++) {
    for(int j = 0; j < matrizUno.length; j++) {
        matrizUno[i][j] = (int)(Math.random() * 5);
    }
}

// Llenado de la segunda matriz
for(int i = 0; i < matrizDos.length; i++) {
    for(int j = 0; j < matrizDos.length; j++) {
        matrizDos[i][j] = (int)(Math.random() * 5);
    }
}

// Suma (y resta) de matrices
for(int i = 0; i < matrizDos.length; i++) {
    for(int j = 0; j < matrizDos.length; j++) {
        matrizResultante[i][j] = matrizUno[i][j] ± matrizDos[i][j];
    }
}
```

Multiplicación de matrices

Para multiplicar matrices, es necesario que:

- ✓ El **número de columnas** de la **primer matriz** sea igual al **número de filas** de la **segunda matriz**.

La matriz resultante tendrá el número de filas igual al número de filas de la primer matriz, y el número de columnas igual al número de columnas de la segunda matriz.

$$M_A \epsilon R^{m \times K} \cdot M_B \epsilon R^{K \times n} = M_r \epsilon R^{m \times n}$$

Una forma de representarlo y comprenderlo adecuadamente es la siguiente:

Matriz dos

1	1	2
2	3	4
5	5	6
6	7	8
7	9	10

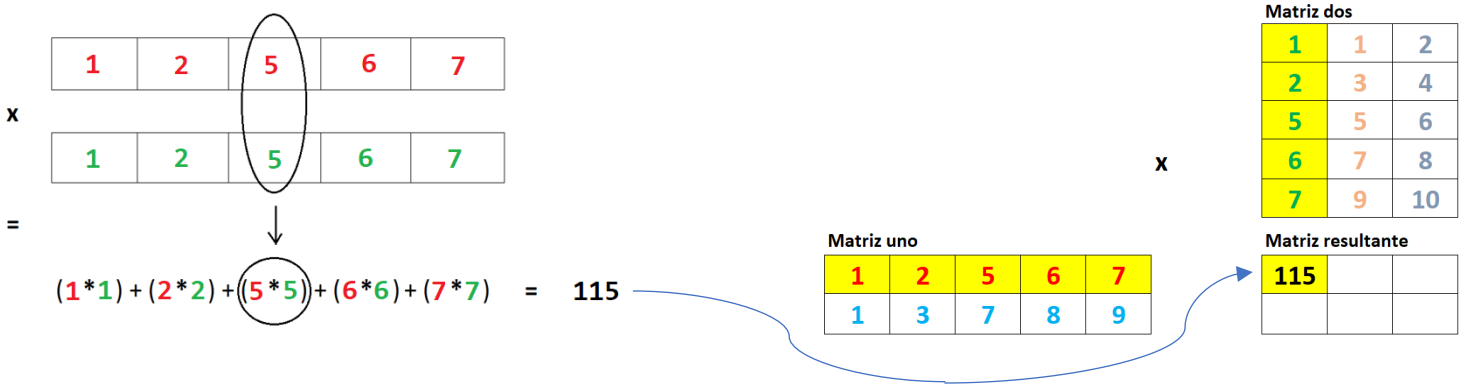
x

Matriz uno

1	2	5	6	7
1	3	7	8	9

Matriz resultante

Se multiplica la primera fila de la primera matriz con la primera columna de la segunda matriz para obtener el primer elemento de la matriz resultante. La multiplicación se efectúa tal como se hace con vectores.




```java
// Multiplicar matrices
int matrizResultante[][];
if(matrizUno[0].length == matrizDos.length) {
    matrizResultante = new int[matrizUno.length][matrizDos[0].length];
    for(int fila = 0; fila < matrizResultante.length; fila++) {
        for(int columna = 0; columna < matrizResultante[0].length; columna++) {

            // Multiplicar fila de matrizUno con columna de matrizDos
            int escalar = 0;
            for(int k = 0; k < matrizDos.length; k++) {
                escalar += matrizUno[fila][k] * matrizDos[k][columna];
            }
            matrizResultante[fila][columna] = escalar;

        }
    }
} else throw new IndexOutOfBoundsException("Columnas de matrizUno no es igual a Filas de matrizDos");
```

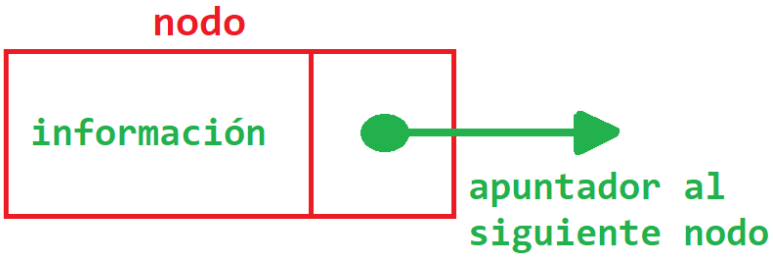
Estructura de datos

Listas

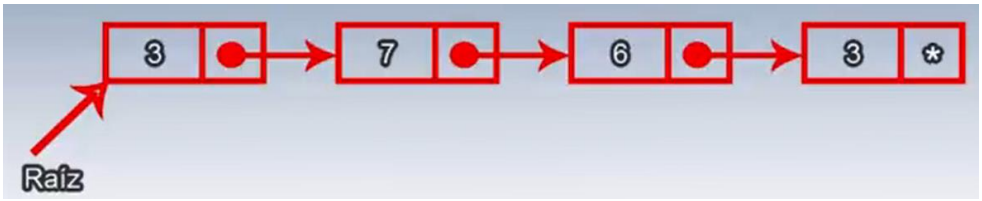
Tipo de dato **abstracto** que permite almacenar datos de una forma organizada al igual que los vectores, pero estos tienen una **estructura dinámica**.

Cada elemento de la lista (**nodo**) tiene dos campos:

- Información — Dato a almacenar.
- Enlace — Apuntador al siguiente elemento, con excepción del último que enlaza un **null**.



Representación de una lista



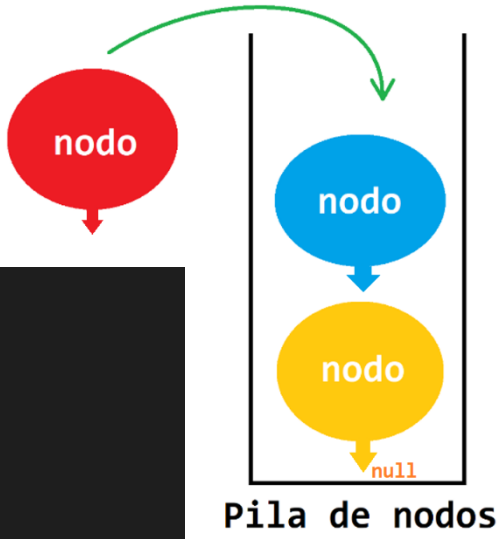

```
public class Nodo {
    int informacion;// Dato del elemento
    Nodo enlace;// Siguiente elemento de la lista

    public Nodo(int informacion) {
        this.informacion = informacion;
        enlace = null;
    }
}
```

Dependiendo del procedimiento de inserción y extracción de nodos en la lista, existen distintos tipos de listas.

- **Lista de tipo pila**

  Las inserciones y extracciones se realizan por el mismo lado de la lista.

  El último elemento en entrar a la lista es el primero en poder salir.

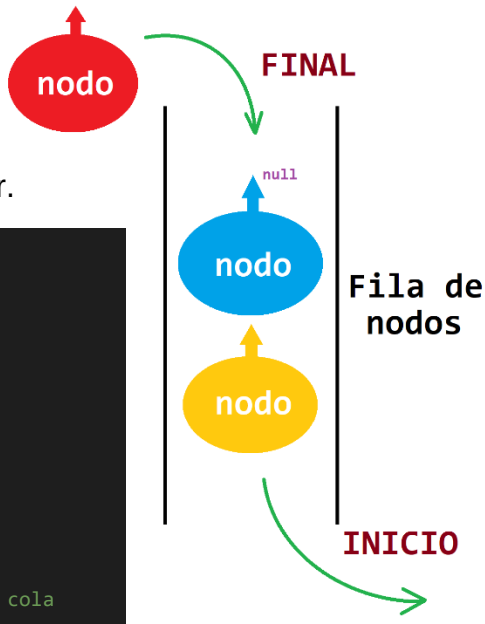



```java
public class Pila {
    private Nodo ultimoNodo;
    int tamano = 0;

    // Inicialización de la lista
    public Pila() {
        ultimoNodo = null;
        tamano = 0;
    }

    public boolean EstaLaPilaVacia() {// Método para saber si la pila está vacía
        return ultimoNodo == null;
    }

    public void InsertarNodo(int informacion) {// Método para ingresar un nodo en la pila
        Nodo nuevoNodo = new Nodo(informacion);
        nuevoNodo.enlace = ultimoNodo;
        ultimoNodo = nuevoNodo;
        tamano++;
    }

    public int EliminarNodo() {// Método para borrar el último nodo añadido a la pila
        if(!EstaLaColaVacia()) {
            int informacion = ultimoNodo.informacion;
            ultimoNodo = ultimoNodo.enlace;
            tamano--;
            return tamano;
        }
    }

    public int UltimoValorIngresado() {// Método para obtener el último valor ingresado
        return ultimoNodo.informacion;
    }

    public int TamanoPila() {// Método para conseguir el tamaño de la pila
        return tamano;
    }

    public void VaciarPila() {// Método para vaciar la pila
        while(!EstaLaPilaVacia()) {
            EliminarNodo();
       }
    }

    public int[] getPila() {// Método para obtener el contenido de la pila
       Nodo nodo = ultimoNodo;
       int pila[] = new int[tamano];
       int i = 0;
       while(nodo!= null) {
           pila[i] = nodo.informacion;
           nodo = nodo.enlace;
           i++;
       }
       return pila;
    }
}
```

- **Lista de tipo cola (FIFO)**

  Las inserciones se realizan en el inicio de la lista y las extracciones se realizan al final de la lista.

  El último elemento en entrar es el último en poder salir.




```java
public class Cola {
    private Nodo inicioCola, finalCola;
    private int tamano = 0;

    public Cola() {// Inicialización de la lista de tipo Cola
        inicioCola = null;
        finalCola = null;
    }

    // Método para saber si la cola está vacía
    public boolean EstaLaColaVacia() {
        return inicioCola == null;
    }

    public void InsertarNodo(int informacion) {// Método para insertar elemento a la cola
        Nodo nuevoNodo = new Nodo(informacion);
        if(EstaLaColaVacia()) {
            inicioCola = nuevoNodo;
            finalCola = nuevoNodo;
        } else {
            finalCola.enlace = nuevoNodo;
            finalCola = nuevoNodo;
        }
        tamano++;
    }

    public int EliminarNodo() {// Método para borrar el nodo al inicio de la cola
        if(!EstaLaColaVacia()) {
            int informacion = inicioCola.informacion;
            if(inicioCola == finalCola) {
                inicioCola = null;
                finalCola = null;
            } else {
                inicioCola = inicioCola.enlace;
            }
            tamano--;
            return informacion;
        } else {
            return Integer.MAX_VALUE;
        }
    }

    public int TamanoCola() {// Método para conseguir el tamaño de la cola
        return tamano;
    }


    public int[] getCola() {// Método para obtener el contenido de la cola
        Nodo nodo = inicioCola;
        int cola[] = new int[tamano];
        int i = 0;
        while(nodo != null) {
            cola[i] = nodo.informacion;
            nodo = nodo.enlace;
            i++;

        }
        return cola;
    }
}
```

Recursividad

Técnica que permite que un bloque de instrucciones codificadas se ejecute una cierta cantidad de veces, logrando en ocasiones reemplazar a las estructuras repetitivas como **while** y **for**.

Para lograr esto, los métodos deben llamarse (invocarse) a sí mismos. Se puede controlar la cantidad de ejecuciones mediante estructuras condicionales.

Un procedimiento recursivo está dividido en tres fases principales.

- **Caso base**
  Cuando termina la recursión.

- **Proceso**
  Valor agregado o acción de la función.

- **Llamada recursiva**
  La función/procedimiento se llama a sí mismo

```
void imprimir_numeros(int n) {

    /* Caso base */
    if (n == 0) {
        return;
    }

    /* Proceso */
    printf("%i\n", n);

    /* Llamado recursivo */
    imprimir_numeros(n-1);
}
```

Ejemplo para el cálculo de un número factorial ($n!$)

El factorial de un número es un valor que se obtiene como resultado de la multiplicación de todos los números enteros que anteceden al número del cual se desea conocer su factorial, a excepción del cero.

4! = 4 * 3 * 2 * 1 = 24

5! = 5 * 4 * 3 * 2 * 1 = 120

6! = 6 * 5 * 4 * 3 * 2 * 1 = 720

7! = 7 * 6 * 5 * 4 * 3 * 2 * 1 = 5,040

```
public int Factorial(int n) {
    int resultado = 1;
    if(n > 0) {
        resultado = n * Factorial(n - 1);
    } else if(n == 0) {
        resultado = 1;
    } else {
        resultado = Integer.MAX_VALUE;
    }
    return resultado;
}
```

Archivos

Se puede trabajar con archivos externos mediante la biblioteca File.

Para ello, es necesario saber los siguientes conceptos:

- Path
- Absoulte path
- Canonical path
- Name
- Parent

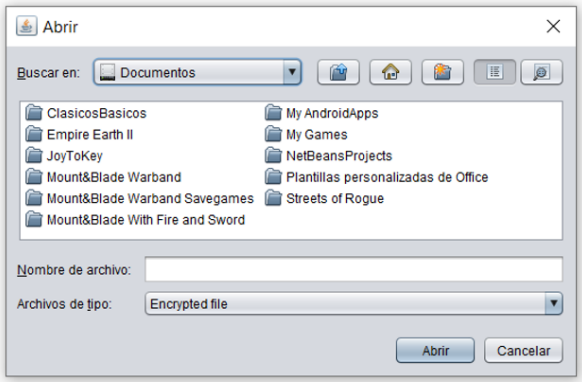
Apertura de un archivo externo con explorer.exe

Se puede abrir una carpeta con la siguiente línea de código:

```
Process p = new ProcessBuilder("explorer.exe", "/select.","./Documents").start();
```

Buscador de archivos externos

```
try {
    JFileChooser fileChooser = new JFileChooser("./");// Path
    fileChooser.setFileSelectionMode(JFileChooser.FILES_ONLY);// Sólo seleccion de archivos (no directorios)
    fileChooser.setMultiSelectionEnabled(false);// Sólo se puede seleccionar un archivo a la vez
    fileChooser.setFileFilter(new javax.swing.filechooser.FileNameExtensionFilter("Encrypted file", "cpt", "txt"));
    int result = fileChooser.showOpenDialog(this);// Ejecuta la ventana
} catch(java.awt.HeadlessException e) {
```



Para obtener el file seleccionado, se utiliza el siguiente condicional:

```
if(result != JFileChooser.CANCEL_OPTION) {
    String filePath = fileChooser.getSelectedFile().getPath();
    System.out.println(filePath);
}
```

Bases de datos

Colección de datos almacenados de forma organizada con una lógica coherente.

Permite compartir estos datos entre distintos usuarios y programas, con lo cual se facilita el intercambio y consulta de información.

- Registros
- Campos

Para crear una base de datos, es necesario utilizar un programa externo: