



GOBIERNO DE LA
CIUDAD DE MÉXICO

ADP AGENCIA
DIGITAL DE
INNOVACIÓN
PÚBLICA

PiLARES

Programador Junior

Guía de temario - Tallerista



Índice de contenido

1. Introducción al lenguaje de programación.....	1
2. Variables.....	16
3. Operadores.....	22
4. Sentencias.....	30
5. Instrucciones	47
6. Arreglos.....	51
7. Programación orientada a objetos.....	67
8. Variables de texto.....	96
9. Pilas.....	111
10. Colas.....	117
11. Métodos de ordenamiento.....	119
12. Entorno de Desarrollo para crear aplicaciones gráficas.....	128
Referencias.....	132

I. Introducción al lenguaje de programación

a. ¿Qué es Java?

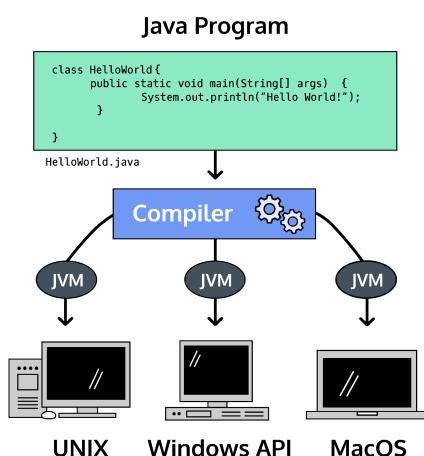
Los lenguajes de programación permiten a los humanos escribir instrucciones que una computadora puede realizar. Con instrucciones precisas, las computadoras coordinan aplicaciones y sistemas que ejecutan el mundo moderno.

Una plataforma es cualquier entorno de hardware o software en el que se ejecuta un programa.

Java es un lenguaje de programación y una plataforma. Java es un lenguaje de programación de alto nivel, robusto, orientado a objetos y seguro. Como plataforma es una colección de programas que ayudan a los programadores a desarrollar y ejecutar aplicaciones de programación Java de manera eficiente. Incluye un motor de ejecución, un compilador y un conjunto de bibliotecas.

Sun Microsystems lanzó el lenguaje de programación Java en 1995. Aunque se lanzó hace más de veinte años, Java sigue siendo uno de los lenguajes de programación más populares en la actualidad.

Una de las razones por las que a la gente le encanta Java es por la máquina virtual Java, que garantiza que el mismo código Java se pueda ejecutar en diferentes sistemas operativos y plataformas. El eslogan de Sun Microsystems para Java fue "escribe una vez, ejecuta en todas partes".



Los lenguajes de programación se componen de *sintaxis*, las instrucciones específicas que entiende Java. Escribimos sintaxis en archivos para crear programas, que son ejecutados por la computadora para realizar la tarea deseada.

¿Para qué se usa Java?

Aquí hay algunas aplicaciones importantes de Java:

- Utilizado para desarrollar aplicaciones de Android
- Ayuda a crear software empresarial
- Amplia gama de aplicaciones java móviles
- Aplicaciones de Computación Científica
- Uso para análisis de Big Data
- Programación en Java de dispositivos de Hardware
- Se utiliza para tecnologías del lado del servidor como Apache, JBoss, GlassFish, etc.

Componentes del lenguaje de programación Java

Un programador de Java escribe un programa en un lenguaje legible por humanos llamado *código fuente*. Por lo tanto, la CPU o los chips nunca entienden el código fuente escrito en ningún lenguaje de programación.

Estas computadoras o chips solo entienden una cosa, que se llama *código o lenguaje de máquina*. Estos códigos de máquina se ejecutan a nivel de CPU. Por lo tanto, serían códigos de máquina diferentes para otros modelos de CPU.

Sin embargo, no necesitas preocuparte por el código de la máquina, ya que la programación tiene que ver con el código fuente. La máquina entiende este código fuente y lo traduce a un código comprensible para la máquina, que es un código ejecutable.

Todas estas funcionalidades suceden dentro de los siguientes 3 componentes de la plataforma Java:

- **Kit de desarrollo de Java (JDK).** JDK es un entorno de desarrollo de software utilizado para crear applets y aplicaciones Java. El nombre completo de JDK es Java Development Kit. Los desarrolladores de Java pueden usarlo en Windows, macOS, Solaris y Linux. JDK les ayuda a codificar y ejecutar programas Java. Es posible instalar más de una versión de JDK en la misma computadora.

¿Por qué usar JDK?

Estas son las principales razones para usar JDK:

- JDK contiene las herramientas necesarias para escribir programas Java para ejecutarlos.
- Incluye compilador, lanzador de aplicaciones Java, Appletviewer, etc.
- El compilador convierte el código escrito en Java en código de bytes.

- **Máquina Virtual Java (JVM).** Java Virtual Machine (JVM) es un motor que proporciona un entorno de tiempo de ejecución para controlar el código Java o las aplicaciones. Convierte el código de bytes de Java en lenguaje de máquina. JVM es una parte de Java Run Environment (JRE). En otros lenguajes de programación, el compilador produce código de máquina para un sistema en particular. Sin embargo, el compilador de Java produce código para una máquina virtual conocida como máquina virtual de Java.

¿Por qué JVM?

Estas son las razones importantes para usar JVM:

- JVM proporciona una forma independiente de la plataforma de ejecutar el código fuente de Java.
- Tiene numerosas bibliotecas, herramientas y marcos.
- Una vez que ejecuta un programa Java, puede ejecutarse en cualquier plataforma y ahorrar mucho tiempo.
- JVM viene con el compilador JIT (Just-in-Time) que convierte el código fuente de Java en un lenguaje de máquina de bajo nivel. Por lo tanto, se ejecuta más rápido que una aplicación normal..

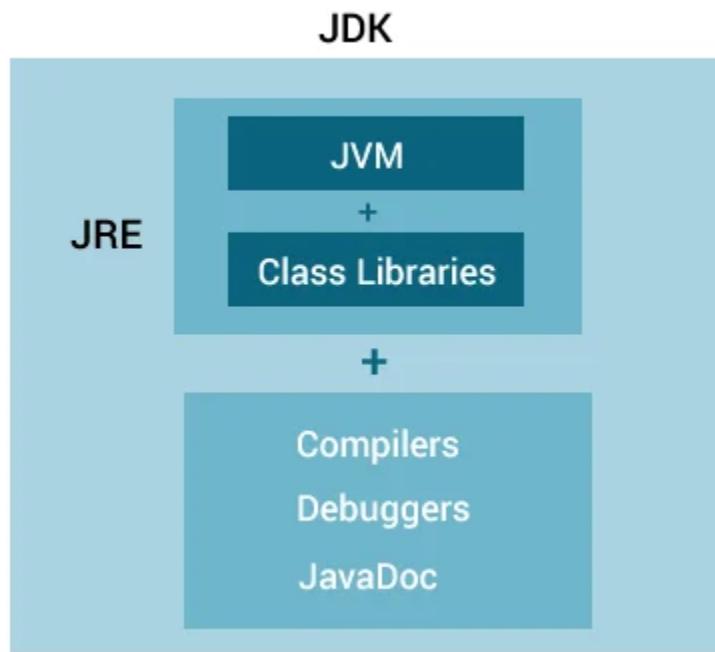
- **Entorno de tiempo de ejecución de Java (JRE).** JRE es una pieza de software que está diseñada para ejecutar otro software. Contiene las bibliotecas de clases, la clase de cargador y la JVM. En términos simples, si deseas ejecutar un programa Java, necesitas JRE. Si no eres programador, no necesitas instalar JDK, solo JRE para ejecutar programas Java.

¿Por qué usar JRE?

Estas son las principales razones para usar JRE:

- JRE contiene bibliotecas de clases, JVM y otros archivos de soporte. No incluye ninguna herramienta para el desarrollo de Java como un depurador, compilador, etc.
- Utiliza clases de paquetes importantes como matemáticas, swing, util, lang, awt y bibliotecas de tiempo de ejecución.
- Si tienes que ejecutar applets de Java, JRE debe estar instalado en su sistema.

Relación entre JVM, JRE y JDK.



Diferentes tipos de plataformas Java

Hay cuatro tipos diferentes de plataformas de lenguaje de programación Java:

- **Plataforma Java, edición estándar (Java SE):** la API de Java SE ofrece la funcionalidad principal del lenguaje de programación Java. Define toda la base de tipo y objeto para clases de alto nivel. Se utiliza para redes, seguridad, acceso a bases de datos, desarrollo de interfaz gráfica de usuario (GUI) y análisis XML.
- **Java Platform, Enterprise Edition (Java EE):** la plataforma Java EE ofrece una API y un entorno de tiempo de ejecución para desarrollar y ejecutar aplicaciones de red altamente escalables, a gran escala, de múltiples niveles, confiables y seguras.

- **Plataforma de lenguaje de programación Java, Micro Edition (Java ME):** la plataforma Java ME ofrece una API y una máquina virtual de tamaño reducido que ejecuta aplicaciones del lenguaje de programación Java en dispositivos pequeños, como teléfonos móviles.
- **Java FX:** JavaFX es una plataforma para desarrollar aplicaciones ricas de Internet utilizando una API de interfaz de usuario liviana. Usa motores gráficos y de medios acelerados por hardware que ayudan a Java a aprovechar los clientes de mayor rendimiento y una apariencia moderna y API de alto nivel para conectarse a fuentes de datos en red.

Java se ejecuta en diferentes plataformas, pero los programadores lo escriben de la misma manera. Los archivos Java tienen una extensión **.java**. ¡Algunos programas son un solo archivo, otros son cientos de archivos!

b. Instalación de Java

A continuación mostraremos cómo instalar la herramienta de desarrollo JDK. JDK incluye Java Runtime Environment, el compilador Java y las API de Java.

Algunas PC's pueden tener Java ya instalado. Para verificar si tienes Java instalado en una PC con Windows, escribe lo siguiente en el símbolo del sistema (cmd.exe):

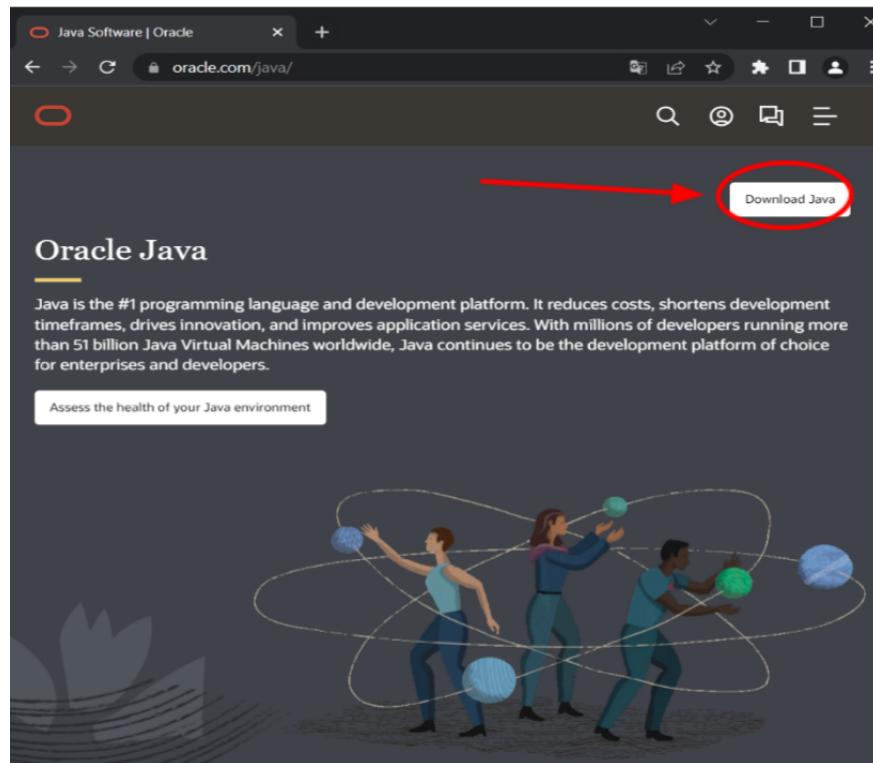
```
C:\Users\Your Name>java -version
```

Si Java está instalado, verás algo como esto (dependiendo de la versión):

```
java version "17.0.2" 2022-01-18 LTS
Java(TM) SE Runtime Environment (build 17.0.2+8-LTS-86)
Java HotSpot(TM) 64-Bit Server VM (build 17.0.2+8-LTS-86, mixed mode, sharing)
```

Si no tienes Java instalado en tu computadora, realiza lo siguiente:

Visita la página de Java en Oracle: <https://www.oracle.com/java/>



Da clic en **Download Java**, te redireccionará a la siguiente página

A screenshot of a web browser displaying the "Java Downloads" section of the Oracle website. The page title is "Java Downloads | Oracle". Below the title, there are tabs for "Java downloads", "Tools and resources", and "Java archive". A search bar at the top includes the placeholder "Looking for other Java downloads?". Below the search bar, there are buttons for "OpenJDK Early Access Builds" and "JRE for Consumers". The main content area features a heading "Java 18 and Java 17 available now". It includes a link to "Learn about Java SE Subscription". A paragraph explains that Java 17 LTS is the latest long-term support release for the Java SE platform. It also mentions that Java 18 will receive updates until September 2022 and Java 17 until at least September 2024. A horizontal navigation bar below the heading shows "Java 18" and "Java 17", with "Java 17" circled in red and a red arrow pointing to it. The section "Java SE Development Kit 17.0.2 downloads" is shown, along with a thank you message for downloading the JDK. At the bottom, there are links for "Linux", "macOS", and "Windows".

Eige la pestaña que dice **Java 17** y da clic donde dice **Windows**

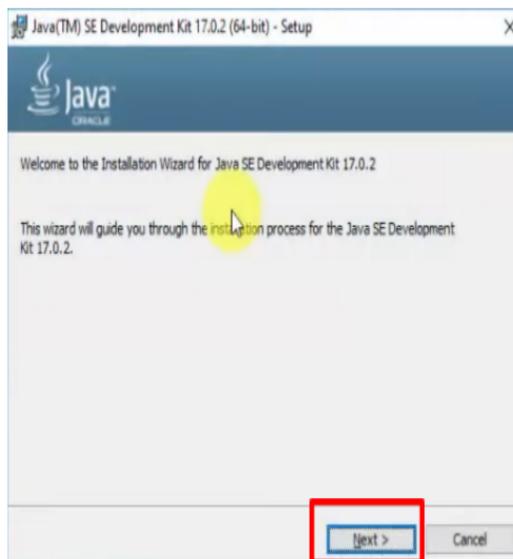
The screenshot shows the Java Downloads page for Oracle. At the top, there are tabs for Java 18 and Java 17, with Java 17 being the active tab. Below this, the title "Java SE Development Kit 17.0.2 downloads" is displayed. A message thanks users for downloading the Java Platform, Standard Edition Development Kit (JDK). The page lists three download options: x64 Compressed Archive, x64 Installer, and x64 MSI Installer, each with a download link and SHA256 hash.

Product/file description	File size	Download
x64 Compressed Archive	171.34 MB	https://download.oracle.com/java/17/latest/jdk-17_windows-x64_bin.zip (sha256)
x64 Installer	152.43 MB	https://download.oracle.com/java/17/latest/jdk-17_windows-x64_bin.exe (sha256)
x64 MSI Installer	151.32 MB	https://download.oracle.com/java/17/latest/jdk-17_windows-x64_bin.msi (sha256)

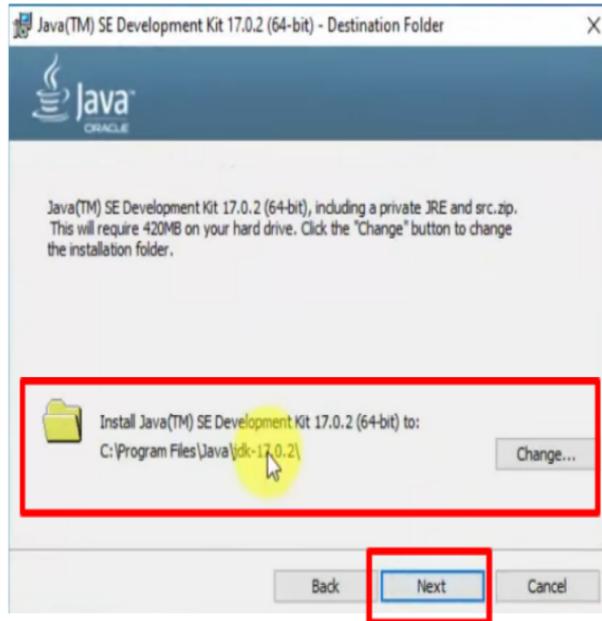
A continuación descarga el archivo ejecutable que se encuentra en el apartado de abajo.

Dirígete a la carpeta de **Descargas** y ejecuta el archivo:

jdk-17_windows-x64_bin.exe



Da clic en **Siguiente**

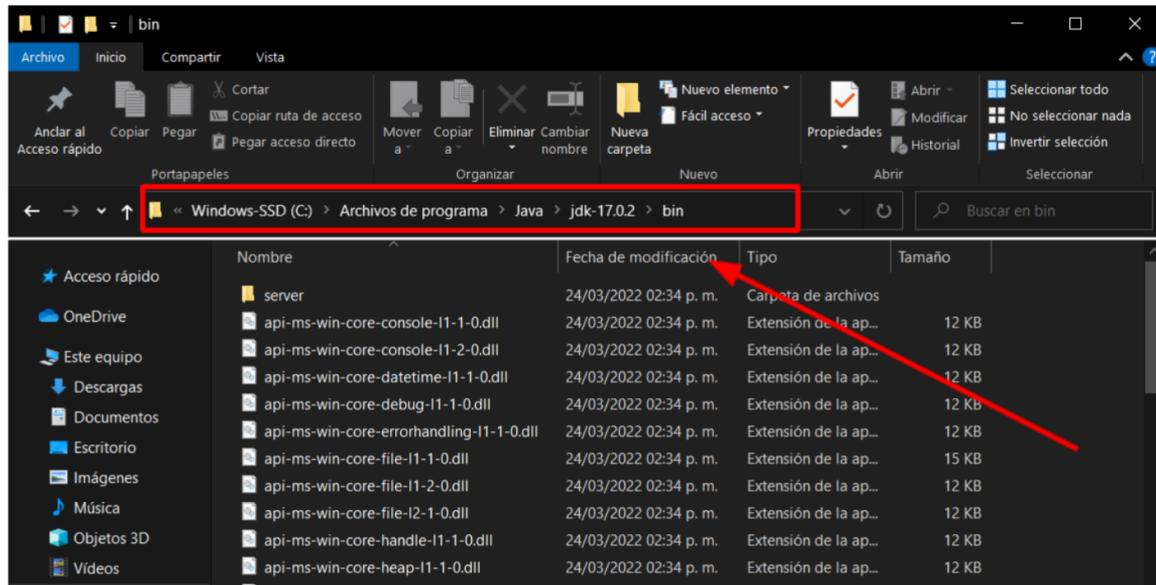


Escoge la dirección donde quieras almacenar los archivos del programa o puedes dejar el que muestra por default; da clic en **Siguiente**. Espera a que se instale

Al finalizar la instalación, cierra la ventana.



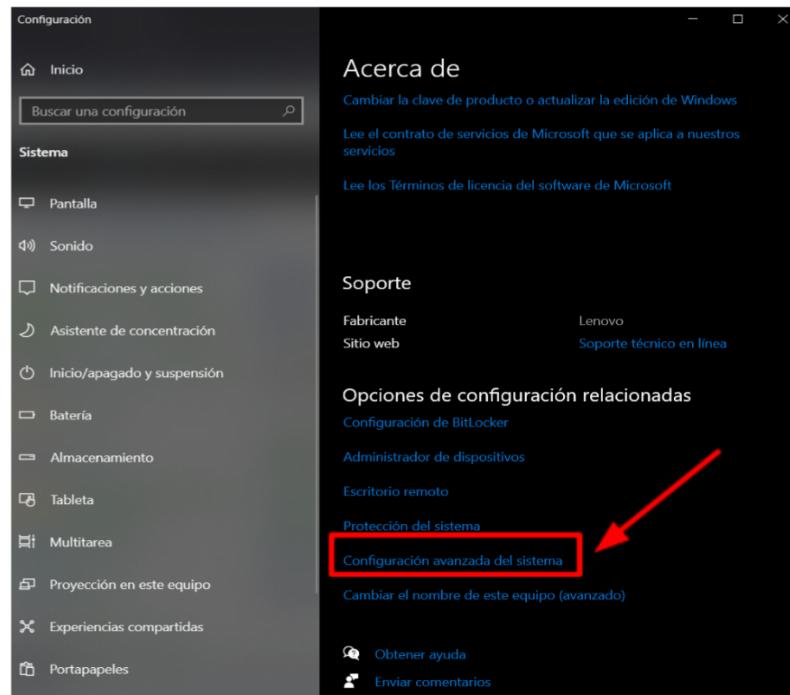
A continuación, ubica la carpeta \bin en el **Explorador de Archivos**



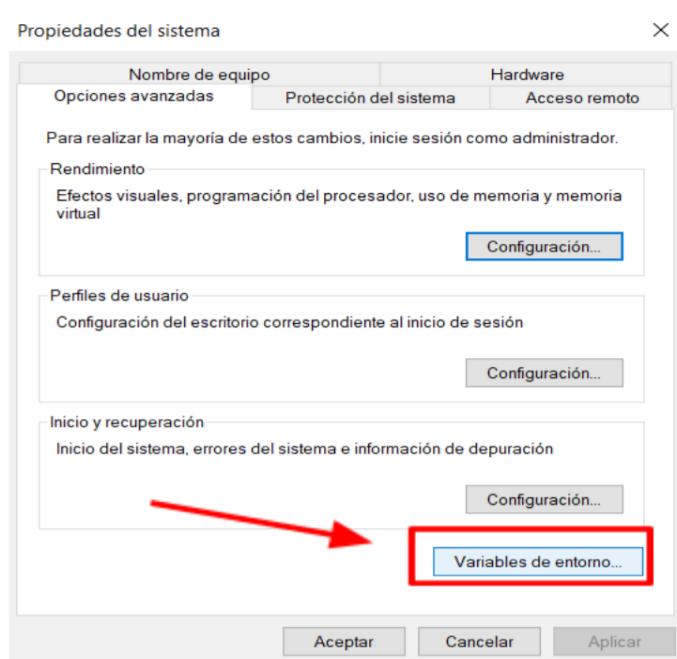
Copia la dirección de la carpeta **\bin**, en el caso del ejemplo es la siguiente:

C:\Program Files\Java\jdk-17.0.2\bin

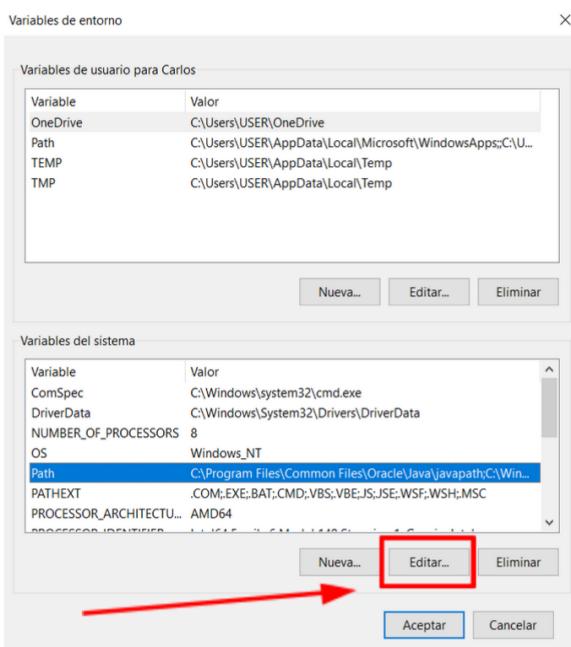
Ve a "Propiedades del sistema" (Se puede encontrar en **Panel de control > Sistema > Configuración avanzada del sistema**).



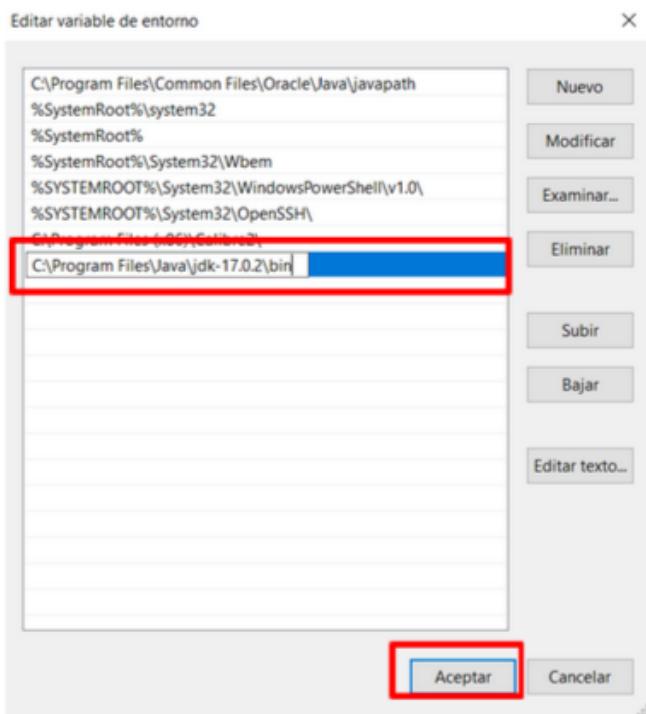
Haz clic en el botón "**Variables de entorno**" en la pestaña "**Opciones avanzadas**".



Luego, selecciona la variable "**Path**" en Variables del sistema y haz clic en el botón "**Editar**".



Haz clic en el botón "Nuevo" y agrega la ruta donde está instalado Java, que es la que habíamos copiado arriba **C:\Program Files\Java\jdk-17.0.2\bin** (si no se especificó nada más cuando lo instaló). Luego, haz clic en "Aceptar" y guarda la configuración.



Por último, abre el símbolo del sistema (cmd.exe) y escribe **java -version** para ver si Java se está ejecutando en tu máquina

```
C:\Users\Your Name>java -version
java version "17.0.2" 2022-01-18 LTS
Java(TM) SE Runtime Environment (build 17.0.2+8-LTS-86)
```

También tenemos algunos editores en línea para programar en Java. Aquí te dejamos algunos:

<https://www.jdoodle.com/online-java-compiler/>
<https://www.programiz.com/java-programming/online-compiler/>
<https://www.interviewbit.com/online-java-compiler/>

Primer programa en Java

A continuación, aprenderás a escribir el programa "Hola Mundo" en Java.

En Java, cada aplicación comienza con un nombre de clase y esa clase (class) debe coincidir con el nombre del archivo.

Vamos a crear nuestro primer archivo Java, llamado **HolaMundo.java**, que se puede hacer en cualquier editor de texto (como el Bloc de notas).

El archivo debe contener un mensaje "**Hola Mundo**", que está escrito con el siguiente código:

```
/* Mi primer programa
*/
class HolaMundo {
    // Contenido dentro de la clase
    public static void main(String[] args) {
        System.out.println("¡Hola Mundo!");
    }
}
```

No te preocupes si no entiendes el código anterior; lo discutiremos en detalle en secciones posteriores. Por ahora, concéntrate en cómo ejecutar el código anterior.

Guarda el código en el Bloc de notas como "**HolaMundo.java**". Abre el símbolo del sistema (cmd.exe), navega hasta el directorio donde guardaste tu archivo y escribe "**javac HolaMundo.java**":

```
C:\Users\USER>javac HolaMundo.java
```

Si el archivo se compila correctamente, este comando genera una clase ejecutable: **HolaMundo.class**. Ejecutable significa que podemos ejecutar este programa desde la terminal. En la terminal, escribe **dir** (**ls** si te encuentras en linux) y presiona la tecla intro o enter. **dir** (o **ls**) es un comando que enumera todos los archivos disponibles.

```
C:\Users\USER>dir
    HolaMundo.class
    HolaMundo.java
```

Ahora, escribe "**java HolaMundo**" para ejecutar el archivo:

```
C:\Users\USER>java HolaMundo
```

La salida debe leer:

¡Hola Mundo!

Ten en cuenta que omitimos la parte **.class** del nombre del archivo.

Java es un lenguaje de programación que distingue entre mayúsculas y minúsculas. Todo el código, los comandos y los nombres de archivo deben usarse en mayúsculas y minúsculas coherentes. **HolaMundo** no es lo mismo que **holamundo**.

Sintaxis básica en Java

Dentro de **HolaMundo.java**, teníamos una clase (**class**):

```
public class HolaMundo{  
}
```

Hablaremos más sobre las clases en el futuro, pero por ahora piense en ellas como un solo concepto.

Marcamos el dominio de este concepto usando llaves: {}. La sintaxis dentro de las llaves es parte de la clase.

Cada archivo tiene una clase principal que lleva el nombre del archivo. Nuestro nombre de clase: **HolaMundo** y nuestro nombre de archivo: **HolaMundo.java**. Cada palabra está en mayúscula.

Dentro de la clase teníamos un **método** main() que enumera las tareas de nuestro programa:

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

Al igual que las clases, usamos llaves para marcar el comienzo y el final de un método.

public, **static** y **void** son sintaxis que aprenderemos más adelante. **String[] args** es un marcador de posición para la información que queremos pasar a nuestro programa. Esta sintaxis es necesaria para que el programa se ejecute, pero es más avanzada de lo que necesitamos explorar en este momento.

Nuestro programa también mostraba el texto "¡Hola Mundo!" en la pantalla. Esto se logró mediante una declaración de impresión:

```
System.out.println("Hello World");
```

Desglosemos un poco más esta línea de código. No te preocupes si algunos de los términos aquí son nuevos para tí. ¡Profundizaremos en lo que son todos estos con mucho más detalle más adelante!

- **System** es una clase de Java incorporada que contiene herramientas útiles para nuestros programas.
- **out** es la abreviatura de "output" ("salida").
- **println** es la abreviatura de "print line" ("línea de impresión").
- **punto y coma** se utilizan para marcar el final de una declaración, una línea de código que realiza una sola tarea.

Podemos usar **System.out.println()** siempre que queramos que el programa cree una nueva línea en la pantalla después de generar un valor:

```
System.out.println("¡Hola Mundo!");
System.out.println("¡Hoy es un grandioso día para programar!");
```

Después de imprimir "¡Hola Mundo!", la terminal de salida crea una nueva línea para que se emita la siguiente instrucción. Este programa imprimirá cada declaración en una nueva línea así:

```
¡Hola Mundo!
¡Hoy es un gran día para programar!
```

También podemos generar información usando **System.out.print()**. Ten en cuenta que estamos usando **print()**, no **println()**. A diferencia de **System.out.println()**, este tipo de declaración de impresión genera todo en la misma línea. Por ejemplo:

```
System.out.print("Hola ");
System.out.print("Mundo");
```

El código anterior tendrá el siguiente resultado:

```
Hola Mundo
```

En este ejemplo, si usaras **print()** o **println()** nuevamente, el nuevo texto se imprimirá inmediatamente después de Mundo en la misma línea. Es importante recordar dónde dejaste el "cursor" de tu programa. Si usas **println()**, el cursor se move a la siguiente línea. Si usa print(), el cursor permanece en la misma línea.

En el futuro, todos los ejemplos usarán **System.out.println()** para generar valores.

Las personas también leen el código, y queremos que nuestras intenciones sean claras para los humanos al igual que queremos que nuestras instrucciones sean claras para la computadora.

Para esto, podemos escribir comentarios, notas para lectores humanos de nuestro código. Estos comentarios no se ejecutan, por lo que no se necesita una sintaxis válida dentro de un comentario.

Cuando los comentarios son cortos, usamos la sintaxis de una sola línea: **//**

```
// Contenido dentro de la clase
```

Cuando los comentarios son largos, usamos la sintaxis para varias líneas: **/*** y ***/**.

```
/* Mi primer programa  
*/
```

Otro tipo de opción de comentarios es el comentario de Javadoc, que se representa mediante **/**** y ***/**. Los comentarios de Javadoc se utilizan para crear documentación para las API (interfaces de programación de aplicaciones). Al escribir comentarios de Javadoc, recuerda que eventualmente se utilizarán en la documentación que tus usuarios puedan leer, así que asegúrate de ser especialmente cuidadoso al escribir estos comentarios.

Los comentarios de Javadoc generalmente se escriben antes de la declaración de campos, métodos y clases (que veremos más adelante):

```
/**  
 La siguiente clase realiza la siguiente tarea...  
*/
```

II. Variables

Digamos que necesitamos un programa que conecte a un usuario con nuevos trabajos. Necesitamos el nombre del usuario, su salario y su situación laboral. Todas estas piezas de información se almacenan en nuestro programa.

Almacenamos información en *variables*, ubicaciones con nombre en la memoria.

Nombrar una pieza de información nos permite usar ese nombre más tarde, accediendo a la información que almacenamos.

Las variables también dan contexto y significado a los datos que almacenamos. El valor **42** podría ser la edad de alguien, el peso en libras o la cantidad de pedidos realizados. Con un nombre, sabemos que el valor **42** es **edad**, **pesoenkilos** o **numerospedidoshechos**.

Así es como creamos una variable en Java,

```
int velocidadLimite = 80;
```

Aquí, **velocidadLimite** es una variable de tipo de datos **int** y le hemos asignado el valor 80.

El tipo de datos **int** sugiere que la variable solo puede contener números enteros (hablaremos de eso más adelante).

En el ejemplo, hemos asignado un valor a la variable durante la declaración. Sin embargo, esto no es obligatorio.

Puedes declarar variables y asignar un valor por separado. Por ejemplo,

```
int velocidadLimite;  
velocidadLimite = 80;
```

Algo importante que debes saber es que Java es un lenguaje de tipo estático. Significa que todas las variables deben declararse antes de que puedan usarse.

El valor de una variable se puede cambiar en el programa, de ahí el nombre **variable**. Por ejemplo,

```
int velocidadLimite;  
... ... ...
```

```
velocidadLimite = 90;
```

Aquí, inicialmente, el valor de **velocidadLimite** es 80. Luego, lo cambiamos a 90.

Sin embargo, no podemos cambiar el tipo de datos de una variable en Java dentro del mismo alcance.

¿Qué es el alcance de una variable?

No te preocupes por eso por ahora. Solo recuerda que no podemos hacer algo como esto:

```
int velocidadLimite;  
... ... ...  
float velocidadLimite;
```

El lenguaje de programación Java tiene su propio conjunto de reglas y convenciones para nombrar variables. Esto es lo que necesitas saber:

- Java distingue entre mayúsculas y minúsculas. Por lo tanto, **edad** y **EDAD** son dos variables diferentes. Por ejemplo,

```
int edad = 24;  
int EDAD = 25;  
  
System.out.println(edad); // Imprime 24  
System.out.println(EDAD); // Imprime 25
```

- Las variables pueden comenzar con una **letra** o un guión bajo, **_** o signo de pesos, **\$**. Por ejemplo,

```
int edad; // nombre válido y buena práctica  
int _edad; // válido pero mala práctica  
int $edad; // válido pero mala práctica
```

- Los nombres de variables no pueden comenzar con números. Por ejemplo,

```
int 1edad; // variable inválida
```

- Los nombres de variables no pueden usar espacios en blanco. Por ejemplo,

```
int mi edad; // variable inválida
```

En este caso, si necesitas usar nombres de variables que tengan más de una palabra, usa todas las letras en minúsculas para la primera palabra y escribe en mayúscula la primera letra de cada palabra subsiguiente. Por ejemplo, **miEdad**.

- Al crear variables, elige un nombre que tenga sentido. Por ejemplo, **puntuacion**, **numero**, **piso** tiene más sentido que nombres de variables como **s**, **n** y **p**.
- Si eliges nombres de variables de una sola palabra, utiliza todas las letras en minúsculas. Por ejemplo, es mejor usar **velocidad** en lugar de **VELOCIDAD** o **vELOCIDAD**.

Hay tres tipos de variables en Java:

1. **Variable local.** Una variable declarada dentro del cuerpo de un método (method) se llama variable local. Puedes usar esta variable solo dentro de ese método y los otros métodos en la clase (class) ni siquiera sabrán que la variable existe.
Una variable local no se puede definir con la palabra clave "static".
2. **Variable de instancia.** Una variable declarada dentro de la clase (class) pero fuera del cuerpo de un método (method) se denomina variable de instancia. No se declara como estático. Se denomina variable de instancia porque su valor es específico de la instancia y no se comparte entre instancias.
3. **Variable estática.** Una variable que se declara como **static** se denomina variable estática. No puede ser local. Puedes crear una sola copia de la variable estática y compartirla entre todas las instancias de la clase. La asignación de memoria para variables estáticas ocurre solo una vez cuando la clase se carga en la memoria.

Ejemplo.

```
class Ejemplo {  
    static int a = 1; // variable estática  
    int data = 99; // variable de instancia  
    void method() {  
        int b = 90; // variable local  
    }  
}
```

Tipos de datos en Java

Como sugiere el nombre, los tipos de datos especifican el tipo de datos que se pueden almacenar dentro de las variables en Java. Hay dos grupos de tipos datos en Java:

- **Tipos de datos primitivos:** incluye **int, double, boolean, char, byte, short, long y float**
- **Tipos de datos no primitivos:** como **String, Arrays y Classes**.

Los tipos de datos primitivos están predefinidos y disponibles en el lenguaje Java. Los valores primitivos no comparten estado con otros valores primitivos.

Tipo de dato int

El primer tipo de datos que almacenaremos es el número entero. Los números enteros son muy comunes en la programación. A menudo los ves usados para almacenar edades, o tamaños máximos, o la cantidad de veces que se ha ejecutado algún código, entre muchos otros usos.

En Java, los números enteros se almacenan en el tipo de datos primitivo **int**.

- **ints** contienen números positivos, números negativos y cero. No almacenan fracciones o números con decimales en ellos.
- El tipo de datos **int** Almacena números enteros desde -2,147,483,648 hasta 2,147,483,647
- El valor por default de **int** es 0.

Para declarar una variable de tipo **int**, usamos la palabra clave **int** antes del nombre de la variable:

```
// int variable declaración
int fechaCreacionJava;
    // Asignación
fechaCreacionJava = 1996;
    // Declaración y asignación
int numeroTiposPrimitivos = 8;
```

Tipo de dato double

Los números enteros no cumplen con lo que necesitamos para cada programa. ¿Y si quisiéramos almacenar el precio de algo? Necesitamos un punto decimal. ¿Y si quisiéramos almacenar la población mundial? Ese número sería mayor de lo que puede contener el tipo **int**.

El tipo de datos primitivo **double** puede ayudar.

- **double** puede contener decimales, así como números muy grandes y muy pequeños.
- El tipo de datos **double** es un punto flotante de precisión doble de 64 bits.
- El valor por default de **double** es 0.0d

Para declarar una variable de tipo **double**, usamos la palabra clave **double** en la declaración:

```
// los doubles pueden tener decimales:  
double precio = 8.99;  
// los doubles pueden tener valores mayores que los que podría contener un int:  
double pib = 1 285 518 000;
```

Tipo de dato boolean

A menudo, nuestros programas enfrentan preguntas que solo pueden responderse con un sí o un no.

¿Está encendido el horno? ¿La luz es verde? ¿Comí el desayuno?

Estas preguntas se responden con un **boolean**, un tipo de dato que hace referencia a uno de los siguientes dos valores: verdadero (**true**) o falso (**false**).

Declaramos variables booleanas usando la palabra clave **boolean** antes del nombre de la variable.

```
boolean javaEsUnLenguajeCompilado = true;  
boolean javaEsUnaTazaDeCafe = false;
```

- El valor por default de **boolean** es **false**.

- El tipo de datos booleano especifica un bit de información, pero su "tamaño" no se puede definir con precisión.

Tipo de dato char

¿Cómo respondemos preguntas como: ¿Cuál es la respuesta correcta, a, b, c, o d? ¿Con qué letra empieza tu nombre?

El tipo de datos **char** puede contener cualquier carácter, como una letra, un espacio o un signo de puntuación.

- Debe estar entre comillas simples, ' '.
- Es un carácter Unicode de 16 bits.
- Su valor por default es '\u0000' (Java usa el sistema Unicode, no el sistema de código ASCII. El \u0000 es el rango más bajo del sistema Unicode.).

Por ejemplo:

```
char respuesta = 'a';
char primeraLetra = 'p';
char signo = '!';
```

Tipo de dato String

El tipo de datos **String** se utiliza para almacenar una secuencia de caracteres (texto). Los valores de **String** (cadena) deben estar entre comillas dobles. Las cadenas (**String**) en Java no son tipos primitivos, sino objetos. Ya hemos visto ejemplos de **String**, por ejemplo, cuando imprimimos "¡Hola Mundo!".

```
String saludo = "¡Hola Mundo!";
System.out.println(saludo);
```

III. Operadores en Java

Los operadores son símbolos que realizan operaciones sobre variables y valores. Por ejemplo, **+** es un operador que se usa para sumar, mientras que ***** también es un operador que se usa para multiplicar.

Los operadores en Java se pueden clasificar en 5 tipos:

1. Operadores aritméticos
2. Operadores de Asignación
3. Operadores relacionales
4. Operadores lógicos
5. Operadores unarios

Operadores aritméticos

Los operadores aritméticos se utilizan para realizar operaciones aritméticas en variables y datos. Por ejemplo,

a+b;

Aquí, el operador **+** se usa para sumar dos variables **a** y **b**. De manera similar, hay varios otros operadores aritméticos en Java.

Operador	Operación
+	Suma
-	Sustracción
*	Multiplicación
/	División
%	Operación Módulo (resto/residuo después de la división)

Suma y Sustracción

Digamos que estamos escribiendo un programa que representa la cuenta bancaria de un usuario. ¡Con variables, sabemos cómo almacenar un saldo! Usaríamos un doble, el tipo de dato primitivo que puede almacenar grandes números decimales. Pero, ¿cómo depositaríamos y retiraríamos de la cuenta?

```
double saldo = 20000.99;  
double cantidadDepositada = 1000.0;  
saldo = saldo + cantidadDepositada;  
// el saldo ahora tiene 21000.99
```

Usamos el operador **+** para sumar los valores **saldo** y **cantidadDepositada**.

Si quisiéramos retirar del saldo, usaríamos el operador **-** :

```
double cantidadARetirar = 500;  
saldo = saldo - cantidadARetirar;  
// el saldo ahora tiene 19500.99
```

¡La suma y la resta también funcionan con valores de tipo **int**!

Multiplicación y División

Digamos que nuestro empleador calcula nuestro cheque de pago y lo deposita en nuestra cuenta bancaria. Si trabajamos 6 días la semana pasada, a razón de \$380 el día. Java puede calcular esto con el operador de multiplicación ***** :

```
double cantidadPagoCheque = 380 * 6;  
// cantidadPagoCheque ahora tiene $2280
```

Si queremos ver cuántas días representa nuestro saldo total, usamos el operador de división **/** :

```
double saldo = 9880;  
double pagoPorDia = 380;  
double diasTrabajados = saldo / pagoPorDia;  
// diasTrabajados ahora tiene 26
```

La división tiene resultados diferentes con números enteros. El operador **/** realiza divisiones enteras, lo que significa que se pierde cualquier resto.

```
int divisionEntera = 10 / 5;  
//divisionEntera ahora tiene 2, porque 10 dividido por 5 es 2  
int divisionNoentera= 10 / 4;  
//divisionNoentera tiene ahora 2, porque 10 dividido por 4 es 2.5
```

divisionEntera almacena lo que esperas, pero **divisionNoEntera** contiene 2 porque **int's** no pueden almacenar decimales. Es importante tener en cuenta que el **int** no redondea el decimal, sino que lo “aplasta”. ¡Java elimina el 0.5 para ajustar el resultado a un tipo **int**!

Es importante tener en cuenta que si intentamos dividir cualquier número por 0, obtendremos como resultado un error **ArithmeticException**.

Módulo

Si horneamos 10 galletas y las repartimos en tandas de 3, ¿cuántas nos sobrarán después de repartir todas las tandas completas que pudimos?

El operador módulo **%** nos da el resto después de dividir dos números.

```
int galletasHorneadas = 10;  
int galletasSobrantes = 10 % 3;  
//galletasSobrantes tiene el valor de 1
```

¡Te queda 1 galleta después de repartir todos las tandas de 3 que pudiste!

Cuando **a = 7** se divide por **b = 4**, el resto es **3**.

Ten en cuenta que el operador **%** se usa principalmente con números enteros.

Operadores de Asignación

Los operadores de asignación se utilizan en Java para asignar valores a las variables. Por ejemplo,

```
int edad;  
edad = 5;
```

Aquí, **=** es el operador de asignación. Asigna el valor de su derecha a la variable de su izquierda. Es decir, se le asigna **5** a la variable **edad**.

Veamos algunos operadores de asignación (compuesta) más usados en Java.

Operador	Ejemplo	Equivalente a
=	a = b;	a = b;
+=	a += b;	a = a + b;
-=	a -= b;	a = a - b;
*=	a *= b;	a = a * b;
/=	a /= b;	a = a / b;
%=	a %= b;	a = a % b;

Imagina que estamos trabajando para vender pasteles y queremos hacer un seguimiento de cuántos pasteles tenemos creando una variable llamada **numPasteles**:

```
int numPasteles = 12;
```

Si horneamos **8** pasteles más, sabemos que podríamos actualizar nuestra variable usando el operador **+** :

```
numPasteles = numPasteles + 8;
// El valor de numPasteles ahora es 20
```

Si bien este método funciona bien, tuvimos que escribir nuestra variable **numPasteles** dos veces. Podemos acortar esta sintaxis usando un *operador de asignación compuesta*.

Los operadores de asignación compuesta realizan una operación aritmética en una variable y luego reasignan su valor. Usando el operador de asignación compuesto **+=** , podemos reescribir nuestro código anterior así:

```
numPasteles += 8;
// El valor de numPasteles ahora es 20
```

Esta vez solo tuvimos que hacer referencia a **numPasteles** una sola vez.

Operadores relacionales en Java

Los operadores relacionales se utilizan para comprobar la relación entre dos operandos. Por ejemplo,

```
a < b;  
// comprobar si a es menor que b
```

Aquí, el operador **<** es el operador relacional el cual comprueba si **a** es menor que **b** o no.

Devuelve verdadero o falso.

Operador	Descripción	Ejemplo
==	Es igual a	3 == 5 devuelve false
!=	No igual a	3 != 5 devuelve true
>	Mayor que	3 > 5 devuelve false
<	Menor que	3 < 5 devuelve true
>=	Mayor o igual que	3 >= 5 devuelve false
<=	Menor o igual que	3 <= 5 devuelve true

Supongamos que queremos retirar dinero de nuestro programa de cuenta bancaria y queremos ver si estamos retirando menos dinero del que tenemos disponible. Los operadores relacionales nos ayudan a plantear y resolver este problema:

```
double saldo = 20000.01;  
double cantidadARetirar = 5000.01;  
System.out.print(cantidadARetirar < saldo);  
//esto imprimirá true, ya que cantidadARetirar es menor que el saldo
```

Podríamos almacenar el resultado en una variable de tipo booleana:

```
double miSaldo= 200.05;  
double costoDeUnaNuevaLaptop= 1000.05;  
boolean puedoComprarLaptop = miSaldo > costoDeUnaNuevaLaptop;  
//puedoComprarLaptop es false, ya que 200.05 no es más grande que 1000.05
```

¿Cómo validaríamos nuestro cheque de pago para ver si nos pagaron la cantidad correcta?

Podemos usar otro operador relacional para hacer esto. **==** nos dirá si dos variables son iguales:

```
double montoChequePagado = 9880;  
double calcularChequePagado = 380 * 6;  
  
System.out.print(montoChequePagado == calcularChequePagado );  
//Esto imprimirá verdadero, ya que montoChequePagado es igual calcularChequePagado
```

Observa que la verificación de igualdad son dos signos de igual seguidos, en lugar de uno solo. ¡Un signo igual, **=**, es cómo asignamos valores a las variables! Es fácil confundirse, así que asegúrate de revisar tu código para saber si estás colocando la cantidad de signos igual correcta.

Para verificar si dos variables no son iguales, podemos usar **!=**:

```
double saldo = 9880;  
double cantidadADepositar = 2280;  
double saldoActualizado = saldo + cantidadADepositar;  
  
boolean saldoHaCambiado = saldo != saldoActualizado;  
// saldoHaCambiado es true, ya que saldo no es igual a saldoActualizado
```

¿Cómo podríamos asegurarnos de que nos pagaron al menos la cantidad que esperábamos en nuestro cheque de pago? Podríamos usar mayor o igual que, **>=**, o menor o igual que, **<=**.

```
double cantidadChequePagado = 2280;  
double calcularChequePagado = 380 * 6;  
System.out.println(cantidadChequePagado >= calcularChequePagado);  
//esto imprimirá true, ya que cantidadChequePagado es igual a calcularChequePagado
```

Ten en cuenta que los operadores relacionales se utilizan en la toma de decisiones y en los bucles.

Operadores lógicos en Java

Los operadores lógicos se utilizan para comprobar si una expresión es verdadera o falsa. Se utilizan en la toma de decisiones.

Operador	Ejemplo	Significado
&& (Logical AND)	expresion1 && expresion2	verdadero (true) solo si tanto la expresion1 como la expresion2 son verdaderas
 (Logical OR)	expresion1 expresion2	Verdadera si expresion1 o expresion2 es verdadera
! (Logical NOT)	! expresion	verdadero si la expresion es falsa y viceversa

Operadores unarios en Java

Los operadores unarios se utilizan con un solo operando. Por ejemplo, **++** es un operador unario que aumenta el valor de una variable en 1. Es decir, **++5** devolverá **6**.

Los diferentes tipos de operadores unarios son:

Operador	Significado
+	Más unario: no es necesario usarlo ya que los números son positivos sin usarlo
-	Menos unario: invierte el signo de una expresión
++	Operador de incremento: incrementa el valor en 1
--	Operador de disminución: disminuye el valor en 1
!	Operador de complemento lógico: invierte el valor de un booleano

Ejemplo

```
int num = 5, numd=12;  
++num;  
// incrementa en 1 a num
```

```
--b  
// decrementa en 1 a numd
```

Aquí, el valor de **num** aumenta a 6 desde su valor inicial de 5 y el valor de **numd** disminuye a 11.

Concatenación de cadenas (String's)

Digamos que queremos imprimir una variable y queremos describirla a medida que la imprimimos. Para nuestro ejemplo de cuenta bancaria, imagina que queremos decirle al usuario:

```
Tu nombre de usuario es: <username>
```

Con el valor de la variable **username** mostrado.

El operador **+**, que usamos para sumar números, se puede usar para concatenar cadenas. En otras palabras, ¡podemos usarlo para unir dos cadenas (String's)!

```
String username = "alejandrahz";  
System.out.println("Tu nombre de usuario es: " + username);
```

Este código imprimirá:

```
Tu nombre de usuario es: alejandrahz
```

Incluso podemos usar un tipo de dato primitivo como la segunda variable para concatenar, y Java inteligentemente lo convertirá primero en una Cadena:

```
int saldo = 10000;  
String mensaje= "Tu saldo es: " + saldo;  
System.out.println(mensaje);
```

Este código imprimirá:

```
Tu saldo es: 10000
```

IV. Sentencias

El compilador de Java ejecuta el código de arriba a abajo. Las declaraciones en el código se ejecutan según el orden en que aparecen. Sin embargo, Java proporciona sentencias que se pueden usar para controlar el flujo del código Java. Estas declaraciones se denominan *declaraciones de flujo de control*. Es una de las características fundamentales de Java, que proporciona un flujo fluido de programa.

Declaración *if*

Imagina que estamos escribiendo un programa que inscribe a los estudiantes a cursos.

- *Si* un estudiante ha completado los requisitos previos, puede inscribirse en un curso.
- *De lo contrario*, deberá tomar los cursos previos que requiere.

No pueden tomar Física II sin terminar Física I.

Representamos este tipo de *toma de decisiones* en nuestro programa en Java utilizando sentencias *condicionales* o de *flujo de control*. Antes de este punto, nuestro código se ejecutaba línea por línea de arriba hacia abajo, pero las declaraciones condicionales nos permiten ser selectivos en qué partes se ejecutarán.

Las declaraciones condicionales verifican una condición booleana (**boolean**) y ejecutan un *bloque* de código según la condición. Las llaves marcan el alcance de un *bloque condicional* similar a un método o clase.

Sintaxis

```
if (condicion) {  
    // bloque de código a ejecutar si la condición es verdadera  
}
```

Ten en cuenta que **if** está en minúsculas. En letras mayúsculas (If o IF) generará un error.

Aquí hay una declaración condicional completa:

```
if (true) {  
    System.out.println("¡Hola Mundo!");  
}
```

Si la condición es verdadera (**true**), entonces se ejecuta el bloque. Así que ¡Hola Mundo! se imprime.

Pero supongamos que la condición es diferente:

```
if (false) {  
    System.out.println("¡Hola Mundo!");  
}
```

Si la condición es falsa (**false**), entonces el bloque no se ejecuta.

Ejemplo

```
int x = 20;  
int y = 18;  
if (x > y) {  
    System.out.println("x es mayor que y");  
}
```

Si un condicional es breve, podemos omitir las llaves por completo:

```
if (true) System.out.println("La brevedad es el alma del ingenio");
```

Ten en cuenta que las declaraciones condicionales no terminan en punto y coma.

Declaración *if-else*

Hemos visto cómo ejecutar un bloque de código usando un condicional, pero ¿qué pasa si hay dos posibles bloques de código que nos gustaría ejecutar?

Por ejemplo, *si* un estudiante tiene el curso previo requerido, *entonces* se inscribe en el curso seleccionado, *de lo contrario*, se inscribe en el curso previo requerido.

Creamos una rama condicional alternativa con la palabra clave **else**.

Sintaxis

```
if (condicion) {  
    //bloque de código a ejecutar si la condición es verdadera  
} else {  
    //bloque de código a ejecutar si la condición es falsa  
}
```

Ejemplo.

```
if (tienePrerequisito) {  
    // Inscríbete en el curso  
}  
else {  
    // Inscríbete en curso previo requerido  
}
```

Esta declaración condicional asegura que se ejecutará exactamente un bloque de código. Si la condición, **tienePrerequisito**, es falsa (**false**), se ejecuta el bloque que está después de **else**.

Ejemplo.

```
int numero = 10;  
// comprueba si el número es mayor que 0  
if (numero > 0) {  
    System.out.println("El número es positivo.");  
}  
// ejecutar este bloque si el número no es mayor que 0  
else {  
    System.out.println("El número no es positivo.");  
}
```

En el ejemplo anterior, tenemos una variable llamada **numero**. Aquí la expresión de prueba **numero > 0** comprueba si **numero** es mayor que 0.

Dado que el valor de **numero** es 10, la expresión de prueba se evalúa como verdadera (**true**). Por lo tanto, se ejecuta el código dentro del cuerpo de **if**.

Si cambiamos el valor del **numero** a un entero negativo, digamos **-5** y ejecutamos el programa con el nuevo valor de **numero**, la expresión de prueba se evalúa como falsa (**false**). Por lo tanto, se ejecutaría el código dentro del cuerpo de **else**.

Este código también se llama declaración if-then-else:

- Si (**if**) la condición es verdadera, entonces (**then**) haz algo.
- De lo contrario (**else**), haz algo diferente.

Declaración *if-else-if*

La declaración *if-else-if* contiene la declaración **if** seguida de varias declaraciones **else-if**. En otras palabras, podemos decir que es la cadena de sentencias **if-else** las que crean un árbol de decisión donde el programa puede entrar en el bloque de código donde la condición es verdadera. También podemos definir una sentencia **else** al final de la cadena.

Sintaxis

```
if(condicion1) {  
    statement 1; //se ejecuta cuando la condicion1 es verdadera  
}  
else if(condicion2) {  
    statement 2; //se ejecuta cuando la condicion2 es verdadera  
}  
...  
else {  
    statement 2; //se ejecuta cuando todas las condiciones son falsas  
}
```

Imagina que nuestro programa ahora está seleccionando el curso apropiado para un estudiante. Verificaremos su envío para encontrar la inscripción correcta al curso.

La declaración condicional ahora tiene múltiples condiciones que se evalúan de arriba hacia abajo:

```
String course = "Teatro";  
  
if (course.equals("Biología")) {  
    // Inscríbete en el curso de Biología.  
} else if (course.equals("Algebra")) {  
    // Inscríbete en el curso de Álgebra  
} else if (course.equals("Teatro")) {  
    // Inscríbete en el curso de Teatro  
} else {  
    System.out.println("¡Curso no encontrado!");  
}
```

La primera condición que se evalúe como verdadera (**true**) hará que se ejecute ese bloque de código. Aquí hay un ejemplo que demuestra el orden:

```
int resultadoPrueba= 72;

if (resultadoPrueba >= 90) {

    System.out.println("A");

} else if (resultadoPrueba >= 80) {

    System.out.println("B");

} else if (resultadoPrueba >= 70) {

    System.out.println("C");

} else if (resultadoPrueba >= 60) {

    System.out.println("D");

} else {

    System.out.println("F");

}

// imprime C
```

Esta declaración condicional encadenada tiene dos condiciones que se evalúan como verdaderas. Dado que **resultadoPrueba >= 70** viene antes que **resultadoPrueba >= 60**, solo se ejecuta el bloque de código anterior.

Ten en cuenta que solo se ejecutará uno de los bloques de código.

Declaraciones condicionales anidadas

Podemos crear estructuras condicionales más complejas creando *declaraciones condicionales anidadas*, que se crean colocando declaraciones condicionales dentro de otras declaraciones condicionales.

Sintaxis

```
if (condición exterior) {
    if (condición anidada) {
```

```

        Instrucción a ejecutar si ambas condiciones son verdaderas
    }
else{
    se ejecuta cuando la condición anidada es falsa
}
}

```

Cuando implementamos declaraciones condicionales anidadas, la declaración externa se evalúa primero. Si la condición externa es verdadera (**true**), entonces se evalúa la declaración anidada interna.

Vamos a crear un programa que nos ayude a decidir qué ponernos según el clima:

```

int temp = 45;
boolean lluvia = true;

if (temp < 60)
{
    System.out.println ("B!Utiliza una chamarra!");
    if (lluvia == true)
    {
        System.out.println ("Consigue una sombrilla.");
    }
    else
    {
        System.out.println ("No es necesario que lleves tu sombrilla.");
    }
}

```

En el fragmento de código anterior, nuestro compilador verificará la condición en la primera declaración **if-then**: **temp < 60**. Dado que **temp** tiene un valor de **45**, esta condición es verdadera; por lo tanto, nuestro programa imprimirá **¡Utiliza una chamarra!**

Luego, evaluaremos la condición de la declaración **if-then** anidada: **lluvia == true**. Esta condición también es verdadera (**true**), por lo que **Consigue una sombrilla** también se imprime en la pantalla.

Ten en cuenta que, si la primera condición fuera falsa (**false**), la condición anidada no se evaluaría.

Bucles en Java

En el mundo de la programación, detestamos estar repitiendo. Hay dos razones para esto:

- Escribir el mismo código una y otra vez nos lleva mucho tiempo.
- Tener menos código significa tener menos para depurar (“debugear”).

Pero a menudo necesitamos hacer el mismo proceso más de una vez. Afortunadamente, las computadoras son realmente buenas (y rápidas) para realizar tareas repetitivas. Y en Java, podemos usar bucles.

Un **bucle** (loop) es una herramienta de programación que permite a los desarrolladores repetir el mismo bloque de código hasta que se cumpla alguna condición.

El compilador primero evalúa una condición booleana. Si la condición es verdadera (**true**), se ejecuta el cuerpo del bucle. Cuando se ejecuta la última línea del cuerpo del ciclo, la condición se vuelve a evaluar. Este proceso continúa hasta que la condición es falsa (**false**). Si la condición inicial es falsa (**false**), el ciclo nunca se ejecuta.

Empleamos bucles para escalar fácilmente los programas, ahorrando tiempo y minimizando los errores.

Repasaremos dos tipos de bucles que veremos en todas partes:

- el bucle **for**
- el bucle **while**

Bucle *for* en Java

Incrementar con bucles es tan común en la programación que Java (como muchos otros lenguajes de programación) incluye una sintaxis específica para abordar este patrón: el bucle **for**.

Sintaxis

```
for (expresionInicial; condicion; actualizarExpresion)
{
    //bloque de código a ejecutar
}
```

Aquí,

1. **expresionInicial** inicializa y/o declara variables y se ejecuta solo una vez.
2. Se evalúa la **condición**. Si la condición es verdadera (**true**), se ejecuta el bloque del bucle **for**.
3. **actualizarExpresion** actualiza el valor de **expresionInicial**.
4. La **condición** se evalúa de nuevo. El proceso continúa hasta que la **condición** es falsa (**false**).

Ejemplo.

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

En el ejemplo anterior, **i** es la variable de control del bucle.

Analicemos el ejemplo anterior:

1. **i = 0**: **i** se inicializa a **0**.
2. **i < 5**: al bucle se le da una condición booleana (**boolean**) que se basa en el valor de **i**. El ciclo continuará ejecutándose hasta que **i < 5** sea falso (**false**).
3. **i++**: **i** aumentará al final de cada bucle y antes de que se vuelva a evaluar la condición.

Entonces, el código se ejecutará a través del bucle un total de cinco veces.

El término "iteración" hace referencia a los bucles. Cuando *iteramos*, solo significa que estamos repitiendo el mismo bloque de código.

Es importante tener en cuenta que, si no creamos el encabezado de bucle **for** correcto, podemos hacer que la iteración se repita demasiadas o muy pocas veces; esta ocurrencia se conoce como *error por uno* o *error por un paso* ("off by one" error).

Por ejemplo, imagina que queremos encontrar la suma de los primeros diez números y escribimos el siguiente código:

```
int suma = 0;  
for (int i = 0; i < 10; i++)  
{  
    suma += i  
}
```

Este código produciría un valor incorrecto de 45. Omitimos agregar 10 a la **suma** porque nuestra variable de control de ciclo comenzó con un valor de 0 y detuvo la iteración después de que tenía un valor de 9. ¡Nos equivocamos por uno! Podríamos arreglar esto cambiando la condición de nuestro bucle para que sea **i <= 10;** o **i < 11;**

Estos errores pueden ser engañosos porque, si bien no siempre producen un error en la terminal, pueden causar algunos errores de cálculo en nuestro código. Estos se denominan *errores lógicos*: el código funciona bien, pero no hizo lo que se esperaba que hiciera.

Bucle while en Java

El bucle **while** se usa para ejecutar un código específico hasta que se cumpla una condición determinada.

Sintaxis

```
while (condición) {  
    // bloque de código a ejecutar  
}
```

Aquí,

1. Un ciclo **while** evalúa la **condición** dentro del paréntesis () .
2. Si la **condición** se evalúa como verdadera (**true**), se ejecuta el código dentro del ciclo **while**.
3. La **condición** es evaluada de nuevo.
4. Este proceso continúa hasta que la **condición** es falsa (**false**).
5. Cuando la **condición** se evalúa como falsa (**false**), el bucle se detiene.

Ejemplo.

```
// Programa para mostrar números del 1 al 5  
  
// declarar variables  
int i = 1, n = 5;  
  
// bucle while de 1 a 5  
while (i <= n)  
{  
    System.out.println (i);  
    i++;  
}
```

Los bucles **while** son extremadamente útiles cuando desea ejecutar algún código hasta que ocurra un cambio específico. Sin embargo, si no estás seguro de que se producirá un cambio, ¡ten cuidado de entrar en un ciclo (bucle) infinito!

Los *bucles infinitos* ocurren cuando la condición nunca se evalúa como falsa (**false**). Esto puede hacer que todo tu programa quiebre.

Ejemplo.

```
int gatitos= 5;  
  
// Esto producirá un bucle infinito:  
while (gatitos < 6) {  
  
    System.out.println("¡No ha suficientes gatitos!");  
  
}
```

En el ejemplo anterior, **gatitos** sigue siendo igual a 5, que es menor que 6. Entonces obtendremos un bucle infinito.

Al recorrer el código, es común usar una variable de contar. Un *contador* (también conocido como *iterador*) es una variable utilizada en la lógica condicional del bucle y (generalmente) incrementa su valor durante cada iteración a través del código. Por ejemplo:

```
// el contador se inicializa  
int deseos= 0;  
  
// la condicional lógica usa el contador  
while (deseos < 3)  
{  
  
    System.out.println ("Deseo cumplido.");  
    //el contador incrementa  
    wishes++;  
  
}
```

En el ejemplo anterior, el contador de **deseos** se inicializa antes del bucle con un valor de **0**, luego el programa seguirá imprimiendo **Deseo cumplido.** y sumando 1 a **deseos**

siempre que **deseos** tengan un valor de menor que 3. Una vez que **deseos** alcancen un valor de 3 o más, el programa saldrá del bucle.

Así que la salida se vería así:

```
Deseo cumplido.  
Deseo cumplido.  
Deseo cumplido.
```

También podemos decrementar contadores así:

```
int sentadillasPorHacer = 10;  
  
while (sentadillasPorHacer > 0)  
{  
  
    System.out.println ("Te quedan por hacer: " + sentadillasPorHacer);  
    sentadillasPorHacer--;  
  
}
```

En el código anterior, el contador, **sentadillasPorHacer**, comienza en 10 y decrementa de uno en uno. Cuando llega a 0, el bucle sale.

El bucle **do/while** es una variante del bucle **while**. Este bucle ejecutará el bloque de código una vez, antes de verificar si la condición es verdadera (**true**), luego repetirá el ciclo mientras la condición sea verdadera(**true**).

Sintaxis

```
do  
{  
    // bloque de código a ejecutar  
}  
while (condición);
```

El siguiente ejemplo usa un bucle **do/while**. El bucle siempre se ejecutará al menos una vez, incluso si la condición es falsa (**false**), porque el bloque de código se ejecuta antes de que se verifique la condición:

```
int i = 0;
do
{
    System.out.println (i);
    i++;
}
while (i < 5);
```

Break y Continue en Java

Si alguna vez queremos salir de un bucle antes de que finalice todas sus iteraciones o queremos omitir una de las iteraciones, podemos usar las palabras clave **break** y **continue**.

La palabra clave **break** se utiliza para salir o interrumpir un bucle. Una vez que se ejecuta **break**, el bucle dejará de iterar. Por ejemplo:

```
for (int i = 0; i < 10; i++)
{
    System.out.println (i);
    if (i == 4)
    {
        break;
    }
}
```

A pesar de que el ciclo se configuró para iterar hasta que la condición **i < 10** sea falsa (**false**), el código anterior generará lo siguiente porque usamos **break**:

```
0
1
2
3
4
```

La palabra clave **continue** se puede colocar dentro de un bucle si queremos omitir una iteración. Si se ejecuta **continue**, la iteración del bucle actual finalizará inmediatamente y

comenzará la siguiente iteración. Podemos usar la palabra clave **continue** para omitir incluso cualquier iteración valorada:

Ejemplo.

```
for (int i = 0; i < 10; i++) {  
    if (i == 4) {  
        continue;  
    }  
    System.out.println(i);  
}
```

Este ejemplo omite el valor de 4.

A continuación veremos ejemplo de uso de las sentencias if, while, for en series de números.

Series

Una serie es una colección numerada de objetos en la que se permiten repeticiones y el orden es importante. Las series pueden ser **finitas**, por ejemplo, C, A, R, L, O, S es una serie de letras con la letra 'C' primero y 'S' al final, o **infinitas**, como la serie de todos los números pares positivos 2, 4, 6, 8, 10, 12, 14

Serie de impares positivos

La *serie de números impares positivos* es una serie formada por todos aquellos números enteros que no son múltiplos del número 2, es decir, no se puede escribir como producto del número 2 con cualquier otro número. Los primeros términos de la serie de números impares positivos son 1, 3, 5, 7, 9, 11, 13, 15

Serie de números impares positivos: 1, 3, 5, 7, 9, 11, 13, 15, 17, ... así sigue sucesivamente.

Para construir esta serie podemos hacer lo siguiente:

1. Toma un valor para **n**. Este es nuestro límite superior para los números impares que se imprimirán.
2. Inicializa la variable **i** con **1**.
3. Comprueba si **i** es menor o igual que **n**, de lo contrario (**else**) ve al paso 7.

4. Si la condición anterior es verdadera (**true**). Comprueba si **i** deja un residuo de 1 cuando se divide por 2.
5. Si la condición anterior es verdadera (**true**), continua con el siguiente paso, de lo contrario (**false**), incrementa **i** en **1** y vuelve al paso 3.
6. Imprime **i**, e incrementa **i** en **1** y vuelve al paso 3.
7. Termina.

Apliquemos ahora esta lógica en nuestro programa.

Mostremos la serie de números impares positivos usando el bucle **while**

```
//asigna un valor a n
int n = 20;

    //imprimir todos los números impares <=n
int i = 1;
while (i <= n)
{
    if (i % 2 == 1)
    {
        System.out.print (i + ", ");
    }
    i++;
}
```

Salida

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34,
```

También podemos usar un bucle **for** para generar la serie de números impares positivos en Java.

```
//asigna un valor a n

int n = 20;

    //imprimir todos los números impares <=n

for (int i = 1; i <= n; i++)
{
    if (i % 2 == 1)
    {
```

```
System.out.print (i + ", ");
    }
}
```

Salida

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34,
```

Serie de Fibonacci

La serie de Fibonacci es una serie donde el siguiente término es la suma de los dos términos anteriores. Los dos primeros términos de la serie de Fibonacci son **0** y **1**.

Serie de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... así sigue sucesivamente.

Primer término	0	
Segundo término	1	
Siguiente término	1 = 0 + 1	término 3
	2 = 1 + 1	término 4
	3 = 2 + 1	término 5
	5 = 3 + 2	término 6
	8 = 5 + 3	término 7
	13	término 8
	21	término 9
	34	término 10
	...así continúa sucesivamente	

Declaremos que nuestros dos primeros términos sean:

```
primerTermino = 0
segundoTermino = 1
```

Los siguientes términos de la serie de Fibonacci se calcularían como:

```
siguienteTermino = primerTermino + segundoTermino ;
primerTermino = segundoTermino ;
segundoTermino = siguienteTermino;
```

```
1 = 0 + 1
1
1
```

siguienteTermino = primerTermino + segundoTermino ;	2 = 1 + 1
---	-----------

Apliquemos ahora esta lógica en nuestro programa.

Mostremos la serie de Fibonacci usando el bucle **for**

```
int n = 10, primerTermino = 0, segundoTermino = 1;  
  
for (int i = 1; i <= n; ++i)  
{  
    System.out.print (primerTermino + ",");  
  
    // calcula el siguiente termino  
  
    int siguienteTermino = primerTermino + segundoTermino;  
    primerTermino = segundoTermino;  
    segundoTermino = siguienteTermino;  
}
```

Salida

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34,
```

En el programa anterior, **primerTermino** y **segundoTermino** se inicializan con **0** y **1** respectivamente (los dos primeros dígitos de la serie de Fibonacci).

Aquí, hemos usado el bucle **for** para:

- imprimir el **primerTermino** de la serie,
- calcular el **siguienteTermino** realizando la suma de **primerTermino** y el **segundoTermino**,
- asignar el valor del **segundoTermino** al **primerTermino** y el **siguienteTermino** al **segundoTermino**.

También podemos usar un bucle **while** para generar la serie de Fibonacci en Java.

```
int i = 1, n = 10, primerTermino = 0, segundoTermino= 1;  
  
while (i <= n)
```

```
{  
    System.out.print (primerTermino + ",");  
  
    int siguienteTermino = primerTermino + segundoTermino;  
    primerTermino = segundoTermino;  
    segundoTermino = siguienteTermino ;  
  
    i++;  
}
```

Salida

0, 1, 1, 2, 3, 5, 8, 13, 21, 34,

El funcionamiento de este programa es el mismo que el programa anterior.

Y, aunque ambos programas son técnicamente correctos, es mejor usar un bucle **for** en este caso. Ésto es porque se conoce el número de iteraciones (de **1** a **n**).

V. Instrucciones

Una alternativa a encadenar condiciones **if-then-else** juntas es usar la sentencia **switch**. Este condicional verificará un valor dado contra cualquier número de condiciones y ejecuta el bloque de código donde haya una coincidencia.

Sintaxis

```
switch (expresión) {  
  
    case valor1:  
        // código  
        break;  
  
    case valor2:  
        // código  
        break;  
  
    ...  
    ...  
  
    default:  
        // declaraciones predeterminadas  
}
```

La **expresión** se evalúa una vez y se compara con los valores de cada caso.

- Si **expresión** coincide con **valor1**, se ejecuta el **código** del caso **valor1**. De manera similar, el **código** del caso **valor2** se ejecuta si la expresión coincide con **valor2**.
- Si no hay coincidencia, se ejecuta el código del caso predeterminado (o por **default**).

Ten en cuenta que el funcionamiento de la instrucción **switch-case** es similar al del condicional **if-else-if**. Sin embargo, la sintaxis de la declaración **switch** es más limpia y mucho más fácil de leer y escribir.

Ejemplo.

```

String curso= "Historia";

switch (curso)
{
case "Algebra":
// Inscríbete en Álgebra
break;
case "Biología":
// Inscríbete en Biología
break;
case "Historia":
//Inscríbete en Historia
break;
case "Teatro":
// Inscríbete en Teatro
break;
default:System.out.println ("Curso no encontrado");
}

```

Este ejemplo inscribe al estudiante en la clase de **Historia** al verificar el valor contenido entre paréntesis, **curso**, contra cada una de las etiquetas de casos (**cases**). Si el valor después de la etiqueta del caso (**case**) coincide con el valor entre paréntesis, se ejecuta el bloque de cambio.

En el ejemplo anterior, **curso** hace referencia a la cadena "**Historia**", que coincide con **case "Historia":** .

Cuando ningún valor coincide, se ejecuta el bloque predeterminado (**default**). Piensa en esto como el equivalente a **else**.

Los bloques **switch** son diferentes a otros bloques de código porque no están marcados con llaves y usamos la palabra clave **break** para salir de la instrucción **switch**.

Sin **break**, se ejecuta el código debajo de la etiqueta **case** coincidente, *incluido el código debajo de otras etiquetas case*, lo que rara vez es el comportamiento deseado.

```

String curso= "Biología";

switch (curso)
{
case "Algebra":
// Inscríbete en Álgebra

```

```

case "Biología":
    // Inscríbete en Biología
case "Historia":
    // Inscríbete en Historia
case "Teatro":
    // Inscríbete en Teatro
default:
    System.out.println ("Curso no encontrado");
}

// ¡Inscribe al estudiante en Biología... Historia y Teatro!

```

Ejemplo.

```

// Programa Java para comprobar el tamaño (size) usando la sentencia switch...case

int numero= 44;
String size;

// declaración switch para comprobar el tamaño
switch (numero) {

    case 29:
        size = "Pequeño";
        break;

    case 42:
        size = "Mediano";
        break;

    // coincide con el valor
    case 44:
        size = "Grande";
        break;

    case 48:
        size = "Extra Grande";
        break;

    default:
        size = "Desconocido";
        break;
}

System.out.println("Tamaño: " + size);

```

En el ejemplo anterior, hemos utilizado la declaración **switch** para encontrar el tamaño (**size**). Aquí, tenemos una variable **numero**. La variable se compara con el valor de cada **case**.

Como el valor coincide con 44, se ejecuta el código de **case 44**.

Aquí, a la variable **size** se le asigna el valor **Grande**.

VI. Arreglos

Hemos visto cómo almacenar piezas individuales de datos en variables. ¿Qué sucede cuando necesitamos almacenar un grupo de datos? ¿Qué pasaría si tuviéramos una lista de estudiantes en un salón de clases? ¿O tuviéramos un ranking de los 10 mejores caballos que terminan una carrera de caballos?

Arreglos unidimensionales

Si estuviéramos almacenando 5 números de boletos de lotería, por ejemplo, tendríamos que crear una variable diferente para cada valor:

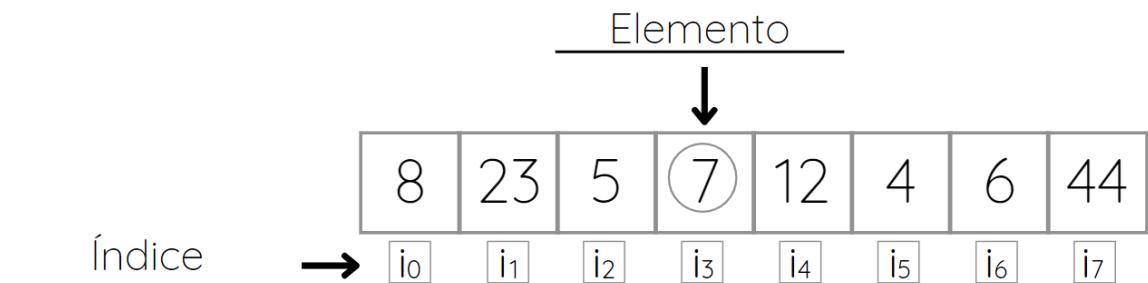
```
int primerNumero = 4;  
int segundNumero = 8;  
int tercerNumero = 15;  
int cuartoNumero = 16;  
int quintoNumero = 23;
```

Eso es un montón de código repetido fútilmente. ¿Y si tuviéramos cien números de lotería? Es más limpio y conveniente usar un *arreglo* de Java para almacenar los datos como una lista.

Una arreglo puede almacenar un número fijo de valores de tipo de datos. Los arreglos almacenan **double**, **int**, **boolean** o cualquier otro tipo de dato primitivo. ¡Los arreglos también pueden contener cadenas (**String's**), así como referencias a objetos!

```
tipoDeDatos [ ] nombreDelArreglo;
```

En un arreglo, cada ubicación de memoria está asociada con un número. El número se conoce como un índice del arreglo. Cada índice de un arreglo se corresponde con un valor diferente. Aquí hay un diagrama de un arreglo lleno de valores enteros:



Observa que los índices comienzan en 0. El elemento en el índice 0 es 8, mientras que el elemento en el índice 1 es 23. Este arreglo tiene una longitud de 8, ya que contiene ocho elementos, pero el índice más grande del arreglo es 7.

Imagina que estamos usando un programa para realizar un seguimiento de los precios de diferentes prendas de vestir que queremos comprar. Querríamos una lista de los precios y una lista de los artículos a los que corresponden cada uno de los precios. Para crear un arreglo, proporcionamos un nombre y declaramos el tipo de datos que contiene:

```
double[ ] precios;
```

Sin embargo, al igual que con las variables, podemos declarar e inicializar en la misma línea. Esto nos permite inicializar explícitamente el arreglo para que contenga los datos que queremos almacenar:

```
double[ ] precios = {13.15, 15.87, 14.22, 16.66};
```

Podemos usar arreglo para almacenar cadenas (**String's**) y otros objetos, como datos primitivos:

```
String[ ] articulosDeRopa = {"Camiseta sin mangas", "Gorra", "Calcetines", "Pantalón"};
```

Ahora que tenemos un arreglo declarado e inicializado, queremos poder obtener valores de él.

Usamos corchetes, [y], para acceder a los datos de un índice determinado:

```
double[ ] precios = {13.1, 15.87, 14.22, 16.66};  
System.out.println(precios[1]);
```

Este comando imprimirá:

```
15.87
```

Esto sucede porque **15.87** es el elemento en el índice **1** del arreglo. Recuerda, el índice de un arreglo comienza en 0 y termina en un índice de uno menos que el número de elementos del arreglo.

Si intentamos acceder a un elemento fuera de su rango de índice correcto, recibiremos el error **ArrayIndexOutOfBoundsException**.

Por ejemplo, si tuviéramos que ejecutar el comando `System.out.println(precios[5])`, obtendríamos el siguiente resultado:

```
java.lang.ArrayIndexOutOfBoundsException: 5
```

También podemos crear arreglos vacíos y luego llenar sus elementos uno por uno. Los arreglos vacíos deben inicializarse con un tamaño fijo:

```
String[] menuComestible = new String[5];
```

¡Una vez que declara este tamaño, no se puede cambiar! Este arreglo siempre será de tamaño 5.

Después de declarar e inicializar, podemos configurar cada índice del arreglo para que sea un elemento diferente:

```
menuComestible[0] = "Hot-Dog vegetariano";
menuComestible[1] = "Ensalada de papa";
menuComestible[2] = "Pan de maíz";
menuComestible[3] = "Brócoli asado";
menuComestible[4] = "Helado de café";
```

Este grupo de comandos tiene el mismo efecto que asignar toda el arreglo a la vez:

```
String[] menuComestible= {"Hot-Dog vegetariano", "Ensalada de papa", "Pan de maíz",
"Brócoli asado", "Helado de café"};
```

¡También podemos cambiar un artículo después de que haya sido asignado! Digamos que este restaurante está cambiando su plato de **Brócoli asado** por uno de **Coliflor al horno**:

```
menuComestible[3] = "Coliflor al horno";
```

Ahora, el arreglo se ve así:

```
["Hot-Dog vegetariano", "Ensalada de papa", "Pan de maíz", "Coliflor al horno", "Helado de
café"]
```

¿Qué sucede si tenemos un arreglo que almacena todos los nombres de usuario de nuestro programa y queremos ver rápidamente cuántos usuarios tenemos? Para obtener la longitud de un arreglo, podemos acceder al campo **length** del objeto del arreglo:

```
String[ ] menuArticulos = new String[5];
System.out.println(menuArticulos.length);
```

Este comando imprimiría **5**, ya que el arreglo **menuArticulos** tiene **5** espacios, aunque todos están vacíos.

Si imprimimos la longitud del arreglo de **precios**:

```
double[ ] precios = {13.1, 15.87, 14.22, 16.66};

System.out.println(precios.length);
```

¡Veríamos **4**, ya que hay cuatro elementos en el arreglo de **precios**!

Una aplicación directa de lo anterior es implementar esto en un bucle recorriendo todo los elementos de la matriz con el bucle, por ejemplo, **for** y usar la propiedad de longitud (**length**) para especificar cuántas veces debe ejecutarse el bucle.

```
// crear un arreglo
int[ ] edad = { 12, 4, 5 };

// recorre el arreglo utilizando el bucle for
for (int i = 0; i < edad.length; i++)
{
    System.out.println(edad[i]);
}
```

También hay un bucle "**for-each**", que se usa exclusivamente para recorrer elementos en arreglos:

Sintaxis

```
for (tipoDeVariable : nombreDelArreglo)
{
    ...
}
```

El siguiente ejemplo genera todos los elementos en el arreglo **autos**, usando un bucle "**for-each**":

```

String[ ] autos = {"Volvo", "BMW", "Ford", "Mazda"};
for (String i : autos)
{
    System.out.println (i);
}

```

El ejemplo anterior se puede leer así: para cada (**for each**) elemento **String** (llamado **i**) en el arreglo **autos**, imprime el valor de **i**.

Si comparas el bucle **for** y el bucle **for-each**, verás que el procedimiento **for-each** es más fácil de escribir, no requiere un contador (usando la propiedad de longitud) y es más legible.

Arreglos bidimensionales

Antes de continuar mencionaremos un concepto básico que nos será de utilidad para lo siguiente.

Una *matriz bidimensional* es una tabla rectangular de números, símbolos o expresiones, dispuestos en filas (renglones) y columnas,

Por ejemplo, la matriz bidimensional **A** tiene dos renglones y tres columnas.

$$A = \begin{bmatrix} -2 & 5 & 6 \\ 5 & 2 & 7 \end{bmatrix}$$

3 columnas

2 renglones

También se puede decir que esta es una matriz bidimensional 2×3 , dos renglones (filas) y tres columnas.

De vuelta a la programación. Como hemos aprendido hasta ahora, un arreglo es un grupo de datos que son del mismo tipo de datos. Esto significa que podemos tener un arreglo de datos de tipo primitivo (por ejemplo de números enteros):

```
[1, 2, 3, 4, 5]
```

Sin embargo, podemos declarar arreglos multidimensionales en Java.

Un arreglo multidimensional es un arreglo de arreglos. Es decir, cada elemento de un arreglo multidimensional es un arreglo por sí mismo. Por ejemplo,

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Al tipo de arreglo del ejemplo anterior se denomina *arreglo bidimensional*, ya que podemos verlos como una matriz bidimensional de valores que contienen tanto filas como columnas.

```
[1, 2, 3]  
[4, 5, 6]  
[7, 8, 9]
```

Adicionalmente, podemos tener arreglos bidimensionales que no tienen forma rectangular. A estos arreglos se les llama arreglos irregulares (o escalonados):

```
[['a', 'b', 'c', 'd'], ['e', 'f'], ['g', 'h', 'i', 'j'], ['k']]
```

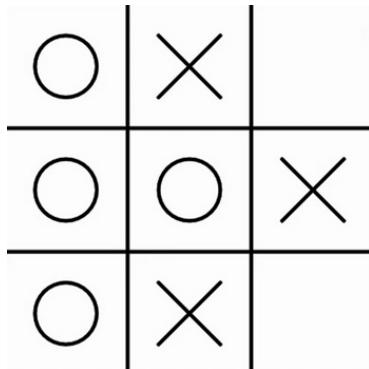
```
['a', 'b', 'c', 'd']  
['e', 'f']  
['g', 'h', 'i', 'j']  
['k']
```

Ten en cuenta que los arreglos bidimensionales no siempre tienen que tener la misma cantidad de subarreglos en cada arreglo. Esto haría que la forma de la matriz bidimensional no fuera rectangular.

¿Por qué usar arreglos bidimensionales?

- Es útil usar arreglos bidimensionales para situaciones en las que se necesita almacenar y organizar datos por filas y columnas. Por ejemplo, exportar datos para utilizarlos en una hoja de cálculo.
- Puedes condensar varios arreglos en una sola variable utilizando arreglos bidimensionales. Por ejemplo, si tienes 10 estudiantes y cada uno tiene 10 calificaciones diferentes en los exámenes, puedes representar las calificaciones generales de los exámenes del grupo a través de un arreglo bidimensional de 10 x 10 haciendo que cada fila represente a un estudiante y cada columna represente una de las pruebas que han realizado.

- Los arreglos bidimensionales se pueden usar para mapear datos. Por ejemplo, si deseas crear el juego del gato (tic-tac-toe), puedes representar el estado del juego mediante un arreglo bidimensional de 3×3 .



Hay muchas otras formas de usar arreglos bidimensionales dependiendo de la aplicación que se quiera dar. El único inconveniente es que, una vez inicializados, no se pueden agregar ni eliminar nuevas filas o columnas sin copiar los datos en un arreglo bidimensional recién inicializado. Esto se debe a que la longitud de las matrices en Java es inmutable (no se puede cambiar después de la creación).

En la declaración de arreglos bidimensionales, el formato es similar al de los arreglos unidimensionales normales, excepto que incluye un conjunto adicional de corchetes después del tipo de datos. En el siguiente ejemplo, **int** representa el tipo de datos, el primer conjunto de corchetes **[]** quiere decir que estamos declarando arreglos y el segundo conjunto de corchetes **[]** quiere decir que estamos declarando un arreglo de arreglos.

```
int[][] arregloBidimensionalEnteros;
```

Puedes pensar en esto como crear un arreglo (**[]**) de arreglos de enteros (**int[]**). Es así como terminamos con **int[][]**.

Ahora que hemos declarado un arreglo bidimensional, veamos cómo comenzamos a asignarle valores iniciales. Al inicializar arreglos (unidimensionales), definimos su tamaño. Inicializar un arreglo bidimensional es diferente porque, en lugar de solo incluir la cantidad de elementos en el arreglo, también debemos indicar cuántos elementos habrá en los sub-arreglos. Esto también se puede considerar como el número de filas y columnas de una matriz bidimensional.

```
int[][] intArreglo1;
intArreglo1 = new int[fila][columna];
```

Aquí tenemos un ejemplo de cómo inicializar un arreglo bidimensional vacío con 3 filas y 5 columnas.

```
int[][] intArreglo2;  
intArreglo2 = new int[3][5];
```

Esto da como resultado una matriz que se ve así:

```
[0,0,0,0,0]  
[0,0,0,0,0]  
[0,0,0,0,0]
```

Si ya sabes qué valores van a estar en el arreglo bidimensional, puedes inicializarlo y escribir todos los valores a la vez. Podemos lograr esto a través de listas de inicializadores.

- En Java, las listas de inicializadores son una forma de inicializar arreglos (unidimensionales) y asignarles valores al mismo tiempo.
- También podemos usar esto para arreglos bidimensionales creando una lista de inicializadores de listas de inicializadores

Un ejemplo de una lista de inicializadores para un arreglo unidimensional sería:

```
char[] arregloChar = {'a', 'b', 'c', 'd'};
```

De manera similar a cómo una lista de inicializadores regulares define el tamaño y los valores de un arreglo, las listas de inicializadores anidados definirán la cantidad de filas, columnas y valores para un arreglo bidimensional.

Hay tres situaciones en las que podemos usar listas de inicializadores para arreglos bidimensionales:

1. En el caso de que la variable aún no haya sido declarada, podemos proporcionar una forma abreviada ya que Java inferirá el tipo de datos de los valores en las listas de inicializadores:

```
double[][] valoresDouble = {{1.5, 2.6, 3.7}, {7.5, 6.4, 5.3}, {9.8, 8.7, 7.6}, {3.6, 5.7, 7.8}};
```

- Si la variable ya ha sido declarada, puede inicializarla creando un nuevo (**new**) arreglo bidimensional con los valores de la lista de inicializadores:

```
String[ ][ ] valoresString;  
valoresString = new String[ ][ ] {{"trabajar", "con"}, {"arreglos", "bidimensionales"}, {"es",  
"divertido"}};
```

- El método anterior también se aplica a la asignación de un nuevo arreglo bidimensional a un arreglo bidimensional existente almacenado en una variable.

Veamos cómo acceder a elementos en un arreglo bidimensional pero primero recordemos cómo acceder a elementos en arreglos unidimensionales.

Para un arreglo unidimensional, todo lo que necesitamos proporcionar es un índice (que puede comenzar en **0**) que representa la posición del elemento al que queremos acceder. ¡Veamos un ejemplo!

Dada una matriz de cinco cadenas (**String's**):

```
String [ ] palabras= {"gato", "perro", "manzana", "oso", "águila"};
```

Podemos acceder al primer elemento usando el índice **0**, al último elemento usando la longitud del arreglo menos uno (en este caso, 4) y a cualquiera de los elementos intermedios. Para esto, proporcionamos el índice del elemento al que queremos acceder dentro de un conjunto de corchetes. Veamos estos ejemplos en código:

```
// Almacenar el primer elemento del arreglo de String palabras  
String primeraPalabra = palabras[0];  
  
// Almacenar el último elemento del arreglo de String  
String ultimaPalabra = palabras[palabras.length-1];  
  
// Almacenar un elemento desde una posición diferente en la matriz  
String palabraDeEnmedio = palabras[2];
```

Ahora, para arreglos bidimensionales, la sintaxis es ligeramente diferente. Esto se debe a que, en lugar de proporcionar un solo índice, debemos proporcionar dos índices. Veamos el siguiente ejemplo:

```
// Damos un arreglo bidimensional de datos enteros  
int[ ][ ] data = {{2,4,6}, {8,10,12}, {14,16,18}};
```

```
// Accedemos y almacenamos un elemento deseado  
int almacenado = data[0][2];
```

Hay dos formas de pensar al acceder a un elemento específico en un arreglo bidimensional.

- La primera forma de pensar es que el primer valor representa una fila y el segundo valor representa una columna en la matriz bidimensional.
- La segunda forma de pensar es que el primer valor representa a qué subarreglo se accede desde el arreglo principal y el segundo valor representa a qué elemento del subarreglo se accede.

El ejemplo anterior del arreglo bidimensional llamado **data** se puede visualizar así. Los índices (**índex**) están etiquetados fuera de la matriz:

Index	0	1	2
0	[2 , 4 , 6]		
1	[8 , 10 , 12]		
2	[14 , 16 , 18]		

Usando este hecho, ahora sabemos que el resultado de **int almacenado = data[0][2];** almacenaría el número entero **6**. Esto se debe a que el valor **6** se encuentra en la primera fila (**índice 0**) y en la tercera columna (**índice 2**). Aquí hay una plantilla que se puede usar para acceder a elementos en matrices 2D:

```
tipoDatos nombreVariable = arregloBiDExistente[fila][columna];
```

Aquí hay otra forma de visualizar el sistema de indexación para nuestra arreglo de enteros del ejemplo de arriba. Podemos ver qué valores de fila y columna se utilizan para acceder a cada uno de los elementos en cada posición.

```
[data[0][0],data[0][1],data[0][2]]  
[data[1][0],data[1][1],data[1][2]]  
[data[2][0],data[2][1],data[2][2]]
```

Al acceder a estos elementos, si el valor de la fila o la columna está fuera de los límites, la aplicación generará el error **ArrayIndexOutOfBoundsException**.

Ahora recordemos cómo modificar elementos en un arreglo unidimensional.

Para un arreglo unidimensional, proporcionas el índice del elemento que deseas modificar dentro de un conjunto de corchetes junto al lado del nombre de la variable y almacena en él un valor aceptable:

```
arregloAlmacenado[5] = 10;
```

Para arreglos bidimensionales, el formato es similar, pero proporcionaremos el índice del arreglo externo en el primer conjunto de corchetes y el índice del subarreglo en el segundo conjunto de corchetes. También podemos pensarla como que se proporciona la fila en el primer conjunto de corchetes y el índice de la columna en el segundo conjunto de corchetes, como si se tratase de una matriz bidimensional:

```
arregloBidimensional[1][3] = 150;
```

Para asignar un nuevo valor a un determinado elemento, asegúrate de que el nuevo valor que se está utilizando sea del mismo tipo o se pueda convertir al tipo que ya está definido en el arreglo bidimensional.

Digamos que queremos reemplazar cuatro valores de un nuevo arreglo bidimensional llamado **intBidim**. Mira en este código de ejemplo para ver cómo elegir elementos individuales y asignarles nuevos valores.

```
int[ ][ ] intBidim = new int[4][3];  
  
intBidim[3][2] = 16;  
intBidim[0][0] = 4;  
intBidim[2][1] = 12;  
intBidim[1][1] = 8;
```

Aquí hay una imagen de antes y después que muestra cuándo se inicializó por primera vez el arreglo bidimensional en comparación con cuándo se accedió y modificó los cuatro elementos.

[0, 0, 0]	[4, 0, 0]
[0, 0, 0]	[0, 8, 0]
[0, 0, 0]	[0, 12, 0]
[0, 0, 0]	[0, 0, 16]

Estamos a punto de ver cómo podemos usar bucles para hacernos la vida más fácil cuando trabajamos con arreglos bidimensionales. Pero antes de hacer eso, tomemos un momento para refrescar la idea sobre cómo funcionan los bucles anidados.

Los bucles anidados consisten de dos o más bucles colocados uno dentro del otro.

La forma en que funciona es que, para cada iteración del bucle externo, el bucle interno finaliza todas sus iteraciones.

Aquí hay un ejemplo usando bucles **for**:

```
for (int externo= 0; externo< 3; externo++)
{
    System.out.println ("El índice externo es: " + externo);
    for (int interno = 0; interno < 4; interno++)
    {
        System.out.println ("\t El índice interno es: " + interno);
    }
}
```

La salida de los bucles anidados anteriores se ve así:

```
El índice externo es: 0
    El índice interno es: 0
    El índice interno es: 1
    El índice interno es: 2
    El índice interno es: 3
El índice externo es: 1
    El índice interno es: 0
    El índice interno es: 1
    El índice interno es: 2
    El índice interno es: 3
El índice externo es: 2
    El índice interno es: 0
    El índice interno es: 1
    El índice interno es: 2
    El índice interno es: 3
```

A partir de este ejemplo, podemos ver cómo cada vez que el ciclo externo itera una vez, el ciclo interno itera por completo.

Este es un concepto importante para el proceso de acceder a los elementos de arreglos bidimensionales, porque para cada fila en una matriz bidimensional, quisiéramos estar iterando a través de cada columna.

Los bucles anidados pueden consistir de cualquier tipo de bucle y de cualquier combinación de bucles. Veamos algunos ejemplos interesantes.

Aquí hay un ejemplo de bucles **while** anidados:

```
int contadorExterno = 0;
int contadorInterno = 0;
while (contadorExterno < 5)
{
    contadorExterno++;
    contadorInterno = 0;
    while (contadorInterno < 7)
    {
        contadorInterno++;
    }
}
```

Incluso podemos tener algunas combinaciones interesantes. Aquí hay un bucle **each-for** dentro de un bucle **while**:

```
int contadorExterno = 0;
int[ ] arregloInterno = { 1, 2, 3, 4, 5 };

while (contadorExterno < 7)
{
    System.out.println ();
    for (int numero : arregloInterno)
    {
        System.out.print (numero * contadorExterno + " ");
    }
    contadorExterno++;
}
```

El resultado del ejemplo anterior crea una tabla de multiplicar:

```
0 0 0 0 0
1 2 3 4 5
2 4 6 8 10
3 6 9 12 15
```

```
4 8 12 16 20  
5 10 15 20 25  
6 12 18 24 30
```

Este es un ejemplo interesante, porque para cada iteración del bucle **while**, iteramos a través de cada elemento de un arreglo usando un bucle **each-for**. Esto es similar al patrón de iteración que usamos para el proceso de acceder a los elementos de un arreglo bidimensional.

Realizar el proceso de acceder a los elementos de un arreglo bidimensional (**traversing arrays**) usando bucles es importante porque nos permite acceder a muchos elementos rápidamente, acceder a elementos en arreglos bidimensionales muy grandes e incluso acceder a elementos en arreglos bidimensionales de tamaños desconocidos.

Consideremos el siguiente arreglo bidimensional

```
char[ ][ ] bloqueDeLetras= {{'a','b','c'},{'d','e','f'},{'g','h','i'},{'j','k','l'}};
```

Veamos qué sucede cuando accedemos a elementos de la matriz externa.

```
System.out.println(bloqueDeLetras[0]);  
System.out.println(bloqueDeLetras[1]);  
System.out.println(bloqueDeLetras[2]);  
System.out.println(bloqueDeLetras[3]);
```

Ésta es la salida del código anterior:

```
abc  
def  
ghi  
jkl
```

Como se puede ver, podemos recuperar el subarreglo completo de cada uno de los elementos del arreglo externo. Notemos que para acceder a estos subarreglos, solo necesitamos ir aumentando el índice. ¡Esto significa que podemos acceder a cada subarreglo en el arreglo bidimensional usando un bucle!

Aquí tenemos un ejemplo que produce el mismo resultado, pero tiene la ventaja que puede manipular arreglos bidimensionales de cualquier tamaño.

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

Éste es el resultado:

```
abc  
def  
ghi  
jkl
```

Si quisiéramos recuperar del arreglo de arriba la letra 'f', usamos:

```
char letraAlmacenada = bloqueDeLetras[1][2];
```

Sin embargo podríamos usar un bucle para recuperar cada uno de los subarreglos almacenados en el arreglo externo, podríamos usar un bucle anidado para acceder a cada uno de los elementos del subarreglo.

Quizás te preguntes cómo podemos calcular la cantidad de iteraciones necesarias para acceder a los elementos del arreglo bidimensional completamente.

- Para encontrar la cantidad de elementos en la matriz externa, solo necesitamos obtener la longitud del arreglo bidimensional.
 - **int longitudDelArregloExterno = bloqueDeLetras.length;**
 - Al pensar en el arreglo bidimensional en forma de matriz, este valor es la altura de la matriz (el número de filas)
- Para encontrar el número de elementos en el subarreglo, podemos obtener la longitud del subarreglo después de que se haya recuperado del arreglo externo.
 - para obtener la longitud del primer subarreglo en el arreglo bidimensional **bloqueDeLetras** podemos usar:
int longitudDelSubarreglo = bloqueDeLetras[0].length;
 - Al pensar en el arreglo bidimensional en forma de matriz, este valor es el ancho de la matriz (el número de columnas)
- En la mayoría de los casos, obtener la longitud del primer subarreglo en el arreglo bidimensional será igual para el resto de los subarreglos (si tiene

forma rectangular), pero hay raras ocasiones en las que la longitud de los subarreglos podría ser diferente.

Veamos un ejemplo.

```
for (int a = 0; a < bloqueDeLetras.length; a++)
{
    for (int b = 0; b < bloqueDeLetras[a].length; b++)
    {
        System.out.print ("Accesando a: " + bloqueDeLetras[a][b] + "\t");
    }
    System.out.println ();
}
```

Puedes pensar en la variable **a** como el índice del bucle externo y en la variable **b** como el índice del bucle interno.

Aquí está la salida:

```
Accesando a: a  Accesando a: b  Accesando a: c
Accesando a: d  Accesando a: e  Accesando a: f
Accesando a: g  Accesando a: h  Accesando a: i
Accesando a: j  Accesando a: k  Accesando a: l
```

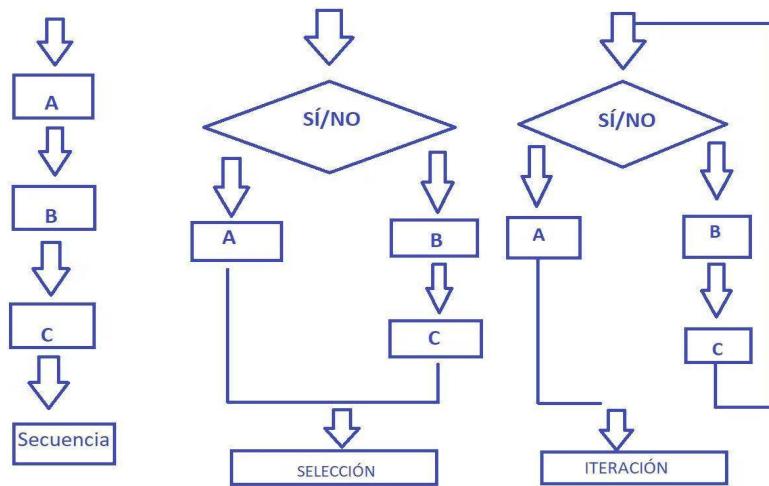
Dentro del bucle **for** anidado, podemos ver que se accede a cada uno de los elementos del subarreglo utilizando el índice de bucle externo para el arreglo externo y el índice de bucle interno para el subarreglo.

No tenemos que usar solo bucles regulares **for** para acceder a los elementos de arreglos bidimensionales. Podemos usar bucles **each-for** si no necesitamos realizar un seguimiento de los índices. Dado que los bucles **each-for** solo usan el elemento de las matrices, es un poco más engorroso hacer un seguimiento de en qué índice estamos. Esta misma idea se aplica también a los bucles **while** y **do-while**. Esta es la razón por la que generalmente usamos bucles **for** regulares, excepto cuando queremos hacer algo tan simple como imprimir.

VII. Programación Orientada a Objetos

En la programación existe algo llamado *paradigmas de programación*. Un **paradigma de programación** es un estilo o forma de programación.

Uno de los paradigmas que usualmente se enseñan es el secuencial o estructurado, es decir, las instrucciones van de arriba a abajo, una después de otra. Es la manera más sencilla de aprender a programar. No tenemos que abstraer cosas complejas. Si no solamente damos una orden, damos otra orden, leemos un dato, lo manipulamos con alguna operación, ponemos una condicional para validar ese resultado, y según el resultado mandamos una cosa u otra.

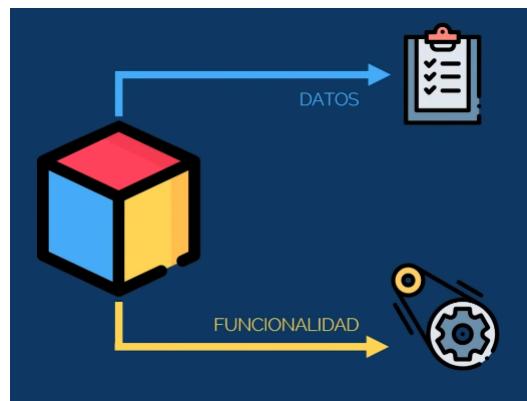


Cuando se empieza a trabajar con proyectos más grandes, este tipo de estilo de programación no ayuda en mucho al punto que pueden complicar mucho las cosas.

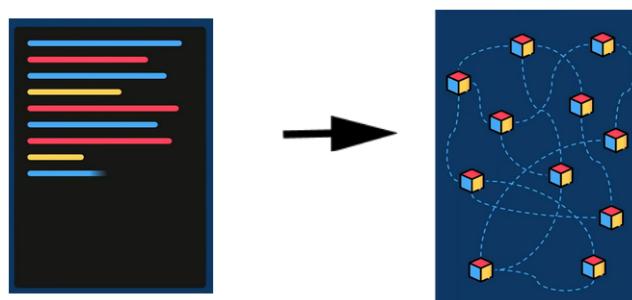
Por ejemplo, un cliente pide una tienda en línea, donde él va a vender zapatos. ¿Cómo vas a hacer de arriba a hacia abajo ese programa? Están como artículos de venta los zapatos que pueden tener como características: un precio, un color, una talla, una marca de tal manera que cada una de estas características puede filtrarse en la búsqueda realizada por el cliente final. Además de esto necesitamos que después de que el cliente haya realizado la selección del artículo, puedan enviarlo al carrito de compras para que después proceda a conectarse a una pasarela de pago, verificar ciertos datos, los productos, la fecha aproximada de entrega, así sucesivamente hasta confirmar la compra del artículo. Son muchas cosas que pueden hacer que se compliquen mucho las cosas si únicamente nos restringimos al uso de la programación estructurada, es decir, un programa que lea instrucciones de arriba hacia abajo.

Aquí es donde necesitamos cambiar de paradigma de programación.

El paradigma más usado en el mundo es la Programación Orientada a Objetos, porque cada uno de estos elementos que necesita el sistema, la aplicación o el programa como lo son en nuestro ejemplo anterior, el carrito, el producto, el usuario, etc., es un "objeto" en este paradigma; y esos objetos tienen sus propios datos y tienen su propio comportamiento (funcionalidad). Por ejemplo, los productos tienen precio, tienen marca, tienen nombre pero también tienen funcionalidad, por ejemplo, pueden ser comprados, pueden ser agregados al carrito, etc. Los usuarios también tienen datos, por ejemplo, su nombre, su número de tarjeta de crédito, su dirección, etc. Pero también tiene acciones que ellos pueden hacer como por ejemplo, comprar productos. El carrito de compras también sería un "objeto" que también tiene datos como por ejemplo, qué productos han sido agregados o si no tiene productos agregados, qué usuario está haciendo uso de ese carrito de compras. Además de que tiene su funcionalidad, por ejemplo, mandar una orden de compra a la pasarela de pago, saber si el pago fue exitoso o no. Cada uno de esos elementos en los que vamos dividiendo el sistema, la aplicación o el programa, es un "objeto" que tiene datos y tienen funcionalidad.



Así, pasamos de tener un programa con código de arriba hacia abajo (programación estructurada) donde las funcionalidades están todas metidas y que es muy difícil de separar y de escalar a un sistema (programación orientada a objetos) donde tenemos los objetos (elementos) separados y que se comunican entre ellos: como el usuario que se comunica con el producto para comprarlo y a la vez el producto que se comunica con el carrito de compras, y el carrito de compras se comunica con la pasarela de pagos y a la vez con el usuario, etc.

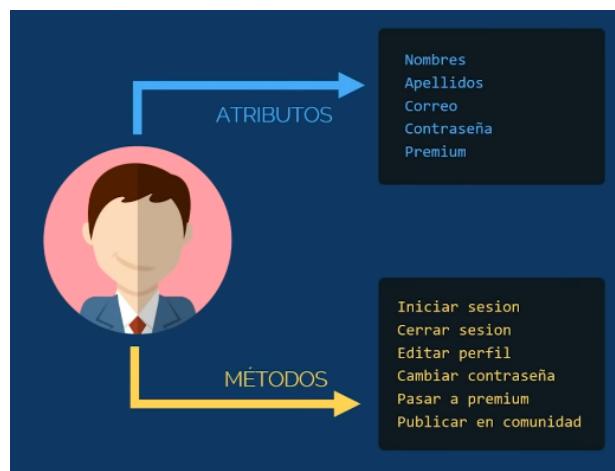


Los objetos se comunican entre ellos.

De esta manera es más fácil manejar y mantener un sistema además de hacerlo crecer, por ejemplo, si más adelante necesitamos otra funcionalidad, podemos agregar otro objeto, o agregar atributos o funcionalidad a los objetos que ya existen.

Como dijimos antes los objetos tienen datos y funcionalidades. Los datos en Programación Orientada a Objetos se llaman **atributos** y las funcionalidades se llaman **métodos**, así, cada objeto tiene sus atributos y tiene sus métodos.

Supongamos que tenemos que programar una aplicación donde se ofrecen cursos en línea. Primeramente, debemos crear usuarios. Para esto usamos un proceso llamado abstracción.

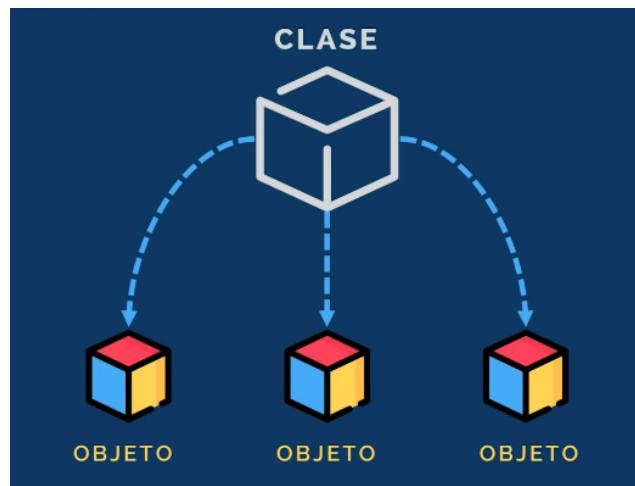


Todo lo anteriormente mencionado tendría que estar establecido en algún lenguaje de programación, por ejemplo java. Pero, ¿qué pasaría si cada cierto tiempo debemos registrar un nuevo usuario? Sería tedioso estar escribiendo bloques de código para cada usuario. Para resolver esto usamos el concepto de **clase** que es como una plantilla o un molde el cual tiene la estructura básica del objeto, sus atributos que son sus datos y su método que es su funcionalidad.

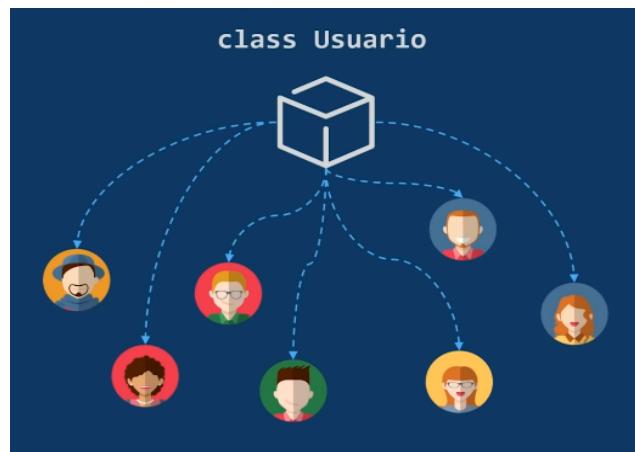


En nuestro ejemplo de arriba no creamos el objeto usuario sino creamos la plantilla, es decir la clase (**class**) **Usuario** de tal manera que cada vez que una persona o usuario nuevo llega y se registra a los cursos en línea y crea su cuenta, lo que está sucediendo es que está usando la clase que con anterioridad se había creado dentro del código de la

aplicación de manera tal que cada usuario nuevo es considerado como un nuevo objeto dentro del código de la aplicación.



Al proceso de crear objetos a partir de una clase se llama **instanciar**. De esta manera, con una sola clase podemos crear decenas, cientos o miles de usuarios, sin tener que escribir código nuevamente.



Hasta aquí, hemos dejado claro que un objeto tiene datos, que son los atributos y tiene funcionalidad, que son los métodos. Y que a través de una clase, que es la plantilla, podemos generar (instanciar) varios objetos.

Como paradigma la Programación Orientada a Objetos se basa en cuatro pilares: abstracción, encapsulamiento, polimorfismo y herencia.

- ABSTRACCIÓN
- ENCAPSULAMIENTO
- POLIMORFISMO
- HERENCIA

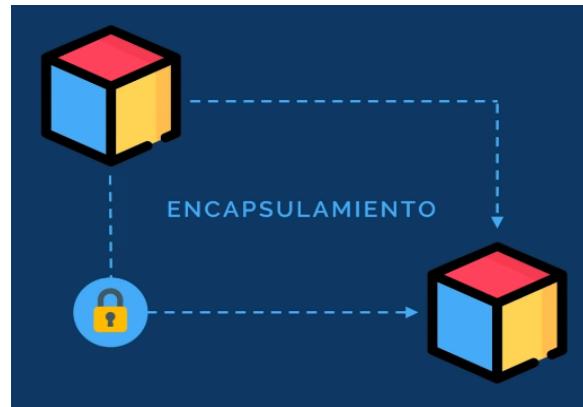
La **abstracción** es un proceso de interpretación y diseño que implica reconocer y enfocarse en las características importantes de una situación o elemento, y filtrar o ignorar todas las particularidades no esenciales.

- Dejar a un lado los detalles y definir características específicas.
- Centrarse en lo que es y lo que hace.
- Hacer más énfasis en qué hace que cómo lo hace

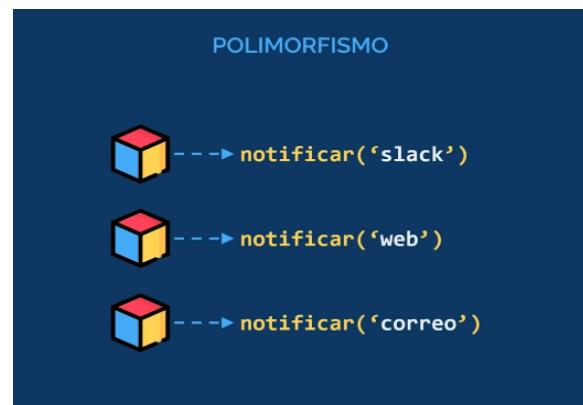
El proceso de abstracción es identificar qué atributos y qué métodos va a tener una clase.



Como vimos anteriormente los objetos se comunican entre ellos. Esto podría tener problemas de seguridad si un objeto pudiera modificar los datos de cualquier otro, para esto necesitamos proteger la información de manipulaciones no autorizadas de tal manera que cuando se comunican los objetos hay cierto tipo de acceso a información que puede tener uno del otro a través de métodos para acceder, lo que evita cambios indebidos. A esto se le llama el **encapsulamiento**.



Polimorfismo es poder dar la misma instrucción a diferentes objetos de tal manera que cada uno de ellos responda a su propia manera. En el ejemplo, de la realización del programa para los cursos en línea, podemos ver la aplicación de polimorfismo cuando, por ejemplo, uno de los usuarios se convierte en usuario Premium. Cuando el usuario se convierte en Premium, se notifica a través de 3 canales: slack, web y correo. Slack es una notificación para quien controla, maneja o tiene acceso a la aplicación. Vía Web, es una notificación que se hace a través de la misma aplicación al usuario. Vía correo, es la notificación enviada a través de correo electrónico al usuario sobre que ha adquirido la suscripción Premium. La notificación a través de los 3 canales es la misma aunque cada uno de esos métodos funciona diferente.



En la Programación Orientada a Objetos, tenemos una clase padre y las clases hijas que **heredan** funcionalidades y atributos de esa clase padre pero sin embargo no son idénticas, solamente aprovechan eso que ya existe para después añadir nuevas cosas. En el ejemplo de la creación de una aplicación para cursos en línea, tenemos una clase para crear usuarios genéricos. Pero qué pasa si necesitamos un usuario que sea Staff que tenga diferentes funcionalidades y atributos al usuarios común. Entonces, lo que hacemos es que creamos una nueva clase Staff que herede ciertas funcionalidades y atributos de la clase padre.

Clases en POO

Una **clase** (**class**) es un prototipo o modelo para el objeto. Antes de crear un objeto, primero necesitamos definir la clase.

Podemos pensar en la clase (**class**) como un boceto (prototipo) de una casa. Contiene todos los detalles sobre los pisos, puertas, ventanas, etc., basados en estas descripciones es que construimos la casa. La casa es el objeto.

Dado que muchas casas se pueden hacer desde la misma descripción, podemos crear muchos objetos de una clase.

Podemos crear una clase en Java usando la palabra clave clase (**class**). Por ejemplo,

```
class nombreDeClase{  
    // atributos  
    // métodos  
}
```

Aquí, los atributos y los métodos representan los **datos** y la **funcionalidad** del objeto, respectivamente.

- Los atributos se utilizan para almacenar datos.
- Los métodos se utilizan para realizar algunas operaciones.

Para nuestro artículo **bicicleta**, podemos crear la clase como

```
class bicicleta  
{
```

```
// atributos o datos  
private int velocidades = 5;  
  
// método o funcionalidad  
public void frenado ()  
{  
    System.out.println ("Los frenos están funcionando");  
}  
}
```

En el ejemplo anterior, hemos creado una clase llamada **bicicleta**. Contiene un atributo llamado **velocidades** y un método llamado **frenado ()**.

Aquí, **bicicleta** es un prototipo. Ahora, podemos crear cualquier número de bicicletas utilizando el prototipo. Y todas las bicicletas compartirán los atributos y los métodos del prototipo.

Objetos en POO

Un objeto se llama una **instancia** de una clase. Por ejemplo, supongamos que **bicicleta** es una clase como antes, entonces **bicicletaMontana**, **bicicletaDeportiva**, **bicicletaTurismo**, etc. pueden considerarse como objetos provenientes de la clase **bicicleta**.

Sintaxis

```
nombreClase objeto = new nombreClase();
```

Ejemplo.

```
// Para la clase bicicleta  
bicicleta bicicletaDeportiva= new bicicleta();  
  
bicicleta bicicletaTurismo = new bicicleta();
```

Hemos utilizado la palabra clave **new** junto con el constructor de la clase para crear un objeto. Los constructores son similares a los métodos y tienen el mismo nombre que la clase. Por ejemplo, la **bicicleta()** es el constructor de la clase **bicicleta**.

Aquí, **bicicletaDeportiva** y **bicicletaTurismo** son los nombres de los objetos. Podemos usarlos para acceder a los atributos y métodos de la clase bicicleta.

Como puedes ver, hemos creado dos objetos de la clase **bicicleta**. Sin embargo, podemos crear múltiples objetos de una sola clase en Java.

Accediendo a las partes de una clase

Para acceder a las partes de una clase podemos usar el nombre de los objetos junto con el operador . (punto) para acceder a las partes de una clase. Por ejemplo,

```
class bicicleta
{
    // atributo o datos
    int velocidades = 5;

    // método o funcionalidad
    void frenado()
    {
        ...
    }

    // Crear un objeto
    bicicleta bicicletaDeportiva = new bicicleta();

    // campo de acceso y método
    bicicletaDeportiva.velocidades;
    bicicletaDeportiva.frenado();
}
```

En el ejemplo anterior, hemos creado una clase llamada **bicicleta**. Incluye un atributo llamado **velocidades** y un método llamado **frenado()**. Observe la declaración,

```
bicicleta bicicletaDeportiva = new bicicleta();
```

Aquí, hemos creado un objeto de **bicicleta** llamado **bicicletaDeportiva**. Luego usamos el objeto para acceder a los **atributos** y al **método** de la clase.

- **bicicletaDeportiva.velocidades** - accede a atributo **velocidades**.
- **bicicletaDeportiva.frenado()** - accede al método **frenado()**.

Veamos el siguiente ejemplo.

```
class lampara
{
```

```

//almacena el valor de la luz
// true si la luz está prendida
// false si la luz está apagada
boolean estaPrendida;

// método para prender la luz
void prendida ()
{
    estaPrendida = true;
    System.out.println ("¿La luz está prendida? " + estaPrendida);

}

// método para apagar la luz
void apagada ()
{
    estaPrendida = false;
    System.out.println ("¿La luz está prendida? " + estaPrendida);
}

}

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

        // se crean los objetos led y halógeno
        lampara led = new lampara();
        lampara halogeno = new lampara();

        // enciende la luz llamando al método prendida()
        led.prendida ();

        // Apaga la luz llamando al método apagada()
        halogeno.apagada ();
    }
}

```

Salida:

```

¿La luz está prendida? true
¿La luz está prendida? false

```

En el programa anterior, hemos creado una clase de nombre **lampara**. Contiene una variable: **estaPrendida**, y dos métodos: **prendida()** y **apagada()**.

Dentro de la clase **main**, hemos creado dos objetos: **led** y **halogeno** de la clase **lampara**. Luego usamos los objetos para llamar a los métodos de la clase.

- **led.prendida ()** - Establece la variable **estaPrendida** como verdadera (**true**) e imprime la salida.
- **halogeno.apagada ()** - Establece la variable **estaPrendida** en falso (**false**) e imprime la salida.

La variable **estaPrendida** definida dentro de la clase **lampara** también es llamada *variable de instancia*. Esto es porque al objeto creado a partir de una clase se le llama *instancia (instance)*. Y, cada instancia tendrá su propia copia de la variable.

Es decir, los objetos **led** y **halogeno** tendrán su propia copia de la variable **estaPrendida**.

Métodos en POO

Un método es un bloque de código que realiza una tarea en específico.

Supongamos que necesitas escribir un programa que cree un círculo y lo coloree. Puedes crear dos métodos para resolver este problema:

- Un método para dibujar el círculo.
- Un método para colorear el círculo.

Dividiendo un problema complejo en situaciones más manejables hará que tu programa sea más fácil de entender y reutilizable.

En Java, hay dos tipos de métodos:

- **Métodos definidos por el usuario:** Podemos crear nuestro propio método en función de nuestras necesidades.
- **Métodos estándar de la biblioteca:** Estos son métodos incorporados en Java que están disponibles para usarse.

Primero aprendamos sobre los métodos definidos por el usuario.

Sintaxis

```
tipoDatoDevolucion nombreMetodo() {  
    // bloque de código del método
```

```
}
```

donde tenemos que,

- **tipoDatoRegresa**: especifica qué tipo de valor regresa un método, por ejemplo, si un método tiene un tipo de dato de devolución (**tipoDatoDevolucion**) **int**, entonces devuelve un valor entero.

Si el método no devuelve un valor, su tipo de devolución es vacío (**void**).

- **nombreMetodo**: es un identificador que se usa para referirse a un método particular en un programa.
- **Bloque de código del método**: incluye las declaraciones en el programa que se utilizan para realizar algunas tareas. El bloque de código del método está delimitado por llaves {}.

Por ejemplo,

```
int sumarNumeros() {  
    // código  
}
```

En el ejemplo anterior, el nombre del método es **sumarNumeros()**. Y, el tipo de devolución de dato es **int**.

Esta sintaxis es la forma más simple de declarar un método. Sin embargo, la sintaxis completa para declarar un método es

Sintaxis

```
modificador static tipoDatoDevolucion nombreDeMetodo (parametro1, parametro2, ...) {  
    // bloque de código del método  
}
```

donde tenemos que,

- **modificador**: define los tipos de acceso si el método es público (**public**), privado (**private**), etc.
- **static**: si usamos la palabra clave **static**, se puede acceder sin crear objetos.

Por ejemplo, el método **sqrt()** de la clase Matemáticas estándar es estática. Por lo tanto, podemos llamar directamente **Math.sqrt** sin crear una instancia de clase **Math**.

- **parametro1 / parametro2** - Estos son valores que se transfieren al método. Podemos pasar cualquier número de argumentos a un método.

En el ejemplo anterior, hemos declarado un método llamado **sumarNumeros()**. Ahora, para usar el método, debemos invocarlo.

Aquí está cómo podemos invocar al método **sumarNumeros()**.

```
// invocación del método  
sumarNumeros();
```

Ejemplo.

```
class Main {  
  
    // creación de un método  
    public int sumarNumeros(int a, int b) {  
        int suma = a + b;  
        // valor de devolución  
        return suma;  
    }  
  
    public static void main(String[] args) {  
  
        int num1 = 25;  
        int num2 = 15;  
  
        // creación de un objeto de Main  
        Main obj = new Main();  
        // invocando el método  
        int resultado = obj.sumarNumeros(num1, num2);  
        System.out.println("La suma es: " + resultado);  
    }  
}
```

Salida:

```
La suma es: 40
```

En el ejemplo anterior, hemos creado un método llamado **sumarNumeros()**. El método toma dos parámetros **a** y **b**. Observa la línea,

```
int resultado = obj.sumarNumeros(num1, num2);
```

Aquí, hemos invocado al método haciéndole pasar los dos argumentos **num1** y **num2**. Dado que el método está devolviendo algún valor, hemos almacenado el valor en la variable **resultado**.

Ten en cuenta que el método no es estático (**static**). Por lo tanto, estamos llamando al método usando el objeto de la clase.

Un método en Java puede o no puede devolver un valor cuando se invoca la función. Utilizamos la *declaración de devolución* (**return**) para devolver cualquier valor. Por ejemplo,

```
int sumaNumeros() {  
    ...  
    return suma;  
}
```

Aquí, estamos devolviendo la variable **suma**. Dado que el tipo de devolución de la función es **int**. La variable **suma** debe ser de tipo **int**. De lo contrario, generará un error.

Ejemplo.

```
class Main {  
  
    // creamos un método  
    public static int cuadrado(int num) {  
  
        // return statement  
        return num * num;  
    }  
  
    public static void main(String[] args) {  
        int resultado;  
  
        // se invoca al método  
        // se almacena el valor devuelto en resultado  
        resultado = cuadrado(10);  
  
        System.out.println("El valor del cuadrado de 10 es: " + resultado);  
    }  
}
```

Salida:

El valor del cuadrado de 10 es: 100

En el programa anterior, hemos creado un método llamado **cuadrado()**. El método toma un número como su parámetro y devuelve el cuadrado del número.

Aquí, hemos mencionado el tipo de devolución del método como **int**. Por lo tanto, el método siempre deberá devolver un valor entero.

Nota que si el método no devuelve ningún valor, usamos la palabra clave **void** como el tipo de devolución del método. Por ejemplo,

```
public void cuadrado(int a) {  
    int cuadrado = a * a;  
    System.out.println("El cuadrado es: " + a);  
}
```

Por otro lado, un parámetro de un método es un valor aceptado por el método. Como se mencionó anteriormente, un método también puede tener cualquier número de parámetros. Por ejemplo,

```
// método con dos parámetros  
int sumarNumeros(int a, int b) {  
    // código  
}  
  
// método sin parámetros  
int sumarNumeros(){  
    // código  
}
```

Si se crea un método con los parámetros, debemos pasar los valores correspondientes al llamar al método. Por ejemplo,

```
// invocando el método con dos parámetros  
sumarNumeros(25, 15);  
  
// invocando el método sin parámetros  
sumarNumeros()
```

Aquí otro ejemplo del uso de parámetros en métodos.

```
class Main {  
  
    // método sin parámetros  
    public void visualizacion1() {  
        System.out.println("Método sin parámetro");  
    }  
  
    // método con un único parámetro  
    public void visualizacion2(int a) {  
        System.out.println("Método con un único parámetro: " + a);  
    }  
  
    public static void main(String[] args) {  
  
        //crea un objeto en main  
        Main obj = new Main();  
  
        // invocando un método sin parámetros  
        obj.visualizacion1();  
  
        // invocando un método con un único parámetro  
        obj.visualizacion2(24);  
    }  
}
```

Salida:

```
Método sin parámetro  
Método con un único parámetro: 24
```

Aquí, el parámetro del método es **int**. Por lo tanto, si pasamos cualquier otro tipo de datos en lugar de **int**, el compilador mostrará un error. Ésto es porque Java es un lenguaje *fuertemente tipado*.

Nota que el argumento **24** que pasa al método **visualización2()** durante la invocación del método se llama el argumento real.

Los métodos anteriormente definidos son los métodos definidos por el usuario pero también, como mencionamos, existen los métodos de la librería estándar.

Los métodos de librería estándar son métodos integrados en Java que están fácilmente disponibles para su uso. Estas librerías estándar vienen junto con la Biblioteca Java Class (JCL) en un archivo Java Archive (*.JAR) con JVM y JRE.

Por ejemplo,

- **print()** es un método de **java.io.PrintSteam**. El método **print("...")** imprime la cadena dentro de las comillas.
- **sqrt()** es un método de la clase **Math**. Devuelve la raíz cuadrada de un número.

Ejemplo.

```
public class Main {  
    public static void main(String[] args) {  
  
        // usando el método sqrt()  
        System.out.print("La raíz cuadrada de 4 es: " + Math.sqrt(4));  
    }  
}
```

Salida:

```
La raíz cuadrada de 4 es: 2.0
```

Ventajas de usar métodos:

- La principal ventaja es la reutilización del código. Podemos escribir un método una vez, y usarlo varias veces. No tenemos que volver a escribir todo el código cada vez. Piensa en ello como, "escribe una vez, reutiliza varias veces".
- Los métodos hacen que el código sea más legible y más fácil de depurar.

Herencia en POO

La herencia es una de las características clave de Programación Orientada a Objetos que nos permite crear una nueva clase de una clase existente.

La nueva clase que se crea se conoce como **subclase** (clase hija o derivada) y la clase existente desde donde se deriva la clase hija se conoce como **superclase** (clase padre o base).

La palabra clave **extends** se utiliza para realizar la herencia en Java. Por ejemplo,

```
class animal {  
    // Aquí irían definidos los métodos y atributos de la clase animal  
}  
  
// uso de la palabra clave extends para llevar acabo la herencia  
class perro extends animal {  
  
    // métodos y atributos de perro  
  
}
```

En el ejemplo anterior, la clase **perro** se crea heredando los métodos y los atributos de la clase **animal**.

Aquí, **perro** es la subclase y **animal** es la superclase.

Más detalladamente tenemos:

```
class animal {  
  
    // atributo y método de la clase padre  
    String nombre;  
    public void comer() {  
        System.out.println("Puedo comer");  
    }  
}  
  
// hereda de animal  
class perro extends animal {  
  
    // nuevo método dentro de la subclase  
    public void visualizar() {  
        System.out.println("Mi nombre es " + nombre);  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
  
        // crear un objeto de la subclase  
        perro labrador = new perro();  
  
        // acceder a los atributos de la superclase  
    }  
}
```

```

labrador.nombre = "Gus";
labrador.visualizar();

//invocar el método de la superclase usando un objeto de la subclase
labrador.comer();

}
}

```

Salida:

```

Mi nombre es Gus
Puedo comer

```

En el ejemplo anterior, hemos derivado una subclase **perro** de la superclase **animal**. Observa las declaraciones,

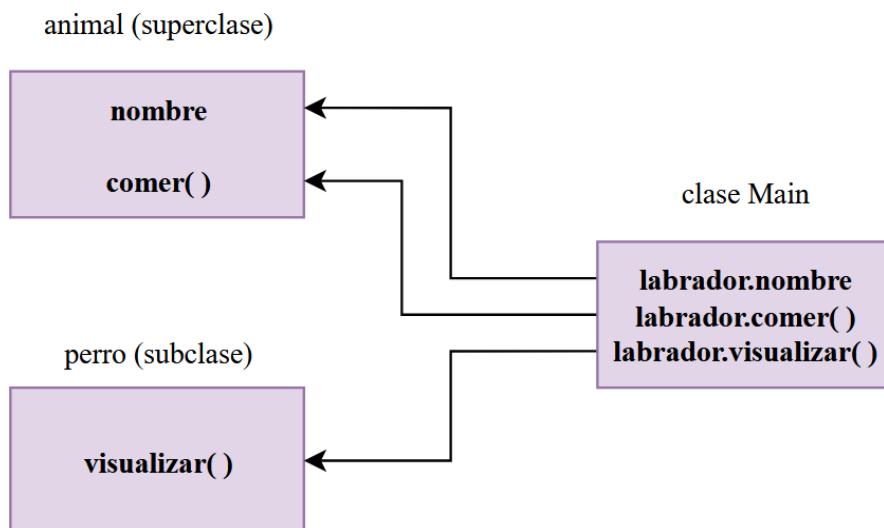
```

labrador.nombre = "Gus";
labrador.comer();

```

Aquí, **labrador** es un objeto de la subclase **perro**. Sin embargo, **nombre** y la **comer()** son miembros de la clase **animal**.

Dado que el **perro** hereda los atributos y el método de **animal**, podemos acceder al atributo y método usando el objeto **perro**.



¿Qué pasaría si el mismo método está presente tanto en la superclase como en la subclase?

En este caso, el método en la subclase anula al método en la superclase. Este concepto se conoce como *método overriding* en Java.

Ejemplo.

```
class animal{  
  
    // método en la superclase  
    public void comer() {  
        System.out.println("Puedo comer");  
    }  
}  
  
// perro hereda animal  
class perro extends animal {  
  
    // overriding el método comer()  
  
    @Override  
    public void comer() {  
        System.out.println("Como comida para perros");  
    }  
  
    // nuevo método en la subclase  
    public void ladrar() {  
        System.out.println("Puedo ladrar");  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
  
        // crea un objeto en la subclase  
        perro labrador = new perro();  
  
        // invoca el método comer()  
        labrador.comer();  
        labrador.ladrar();  
    }  
}
```

Salida:

```
Como comida para perros  
Puedo ladrar
```

En el ejemplo anterior, el método **eat()** está presente tanto en la superclase **animal** como en subclase **perro**.

Aquí, hemos creado un objeto **labrador** a partir de la subclase **perro**.

Ahora, cuando invocamos **comer()** utilizando el objeto **labrador**, el método dentro de la subclase **perro** es invocado. Esto se debe a que el método dentro de la clase derivada anula el método dentro de la clase base.

@Override es una anotación para decirle al compilador que estamos anulando un método. Sin embargo, la anotación no es obligatoria.

Hay cinco tipos de herencia:

1. **Herencia Única.** En este tipo de herencia, una única subclase se extiende desde una única superclase.
2. **Herencia Multinivel.** En la herencia multinivel, una subclase se extiende desde una superclase y luego la misma subclase actúa como una superclase para otra clase.
3. **Herencia Jerárquica.** En la herencia jerárquica, múltiples subclases se extienden desde una sola superclase.
4. **Herencia Múltiple.** En múltiples herencias, una sola subclase se extiende desde múltiples superclases.
5. **Herencia Híbrida.** La herencia híbrida es una combinación de dos o más tipos de herencia.

¿Por qué es importante usar la herencia en Java?

- El uso más importante de la herencia en Java es la reutilización del código. El código que está presente en la clase principal puede ser utilizado directamente por la clase hija.
- El método overriding es también conocido como polimorfismo de tiempo de ejecución. Por lo tanto, podemos lograr el polimorfismo en Java con la ayuda de la herencia.

Polimorfismo en POO

El **polimorfismo** es un concepto importante en Programación Orientada a Objetos. Simplemente significa más de una forma.

Es decir, la misma entidad (método, operador u objeto) puede realizar diferentes operaciones en diferentes escenarios.

Aquí un ejemplo:

```
class Poligono {  
  
    // método para realizar una figura  
    public void render() {  
        System.out.println("Hacer polígono...");  
    }  
}  
  
class Cuadrado extends Poligono {  
  
    // método para hacer un cuadrado  
    public void render() {  
        System.out.println("Hacer un cuadrado...");  
    }  
}  
  
class Circulo extends Poligono {  
  
    // método para hacer un círculo  
    public void render() {  
        System.out.println("hacer un círculo...");  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
  
        // crear el objeto s1 a partir de Cuadrado  
        Cuadrado s1 = new Cuadrado();  
        s1.render();  
  
        // crear el objeto c1 a partir de Círculo  
        Circulo c1 = new Circulo();  
        c1.render();  
    }  
}
```

Salida:

```
Hacer un cuadrado...  
hacer un círculo...
```

En el ejemplo anterior, hemos creado una superclase: **Polígono** y dos subclases: **Cuadrado** y **Círculo**. Observa el uso del método **render()**.

El objetivo principal del método **render()** es realizar una figura. Sin embargo, el proceso de realizar un cuadrado es diferente al proceso de realizar un círculo.

Por lo tanto, el método **render()** se comporta de manera diferente en diferentes clases. O podemos decir también que **render()** es **polimórfico**.

El polimorfismo nos permite crear un código consistente. En el ejemplo anterior, también pudimos haber creado diferentes métodos: **renderCuadrado()** y **renderCírculo()** para realizar un cuadrado o un círculo, respectivamente.

Esto funcionará perfectamente. Sin embargo, para cada figura (círculo, cuadrado, etc.), necesitamos crear diferentes métodos. Esto hará que nuestro código sea inconsistente.

Para resolver esto, el polimorfismo en Java nos permite crear un solo método **render()** que se comportará de manera diferente para diferentes figuras.

Nota: que el método **print()** también es un ejemplo de polimorfismo. Se usa para imprimir valores de diferentes tipos como **char**, **int**, **string**, etc.

Podemos llevar a cabo el **polimorfismo** en Java usando las siguientes maneras:

1. **Overriding el Método**
2. **Sobrecarga del Método**
3. **Operador sobrecargado**

Overriding el Método

Cuando hablamos antes de la herencia en Java, dijimos que si el mismo método está presente tanto en la superclase como en la subclase. Entonces, el método en la subclase anula el mismo método en la superclase. Esto se llama método overriding.

En este caso, el mismo método realizará una operación en la superclase y otra operación en la subclase. Por ejemplo,

```
class Lenguaje{  
    public void visualizarInfo() {  
        System.out.println("Un lenguaje común en inglés");  
    }  
  
    class Java extends Lenguaje{  
        @Override
```

```

public void visualizarInfo() {
    System.out.println("Lenguaje de programación Java");
}

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

        // creando un objeto de la clase Java
        Java j1 = new Java();
        j1.visualizarInfo();

        // creando un objeto de la clase Lenguaje
        Lenguaje l1 = new Lenguaje();
        l1.visualizarInfo();
    }
}

```

Salida:

Lenguaje de programación Java
Un lenguaje común en inglés

En el ejemplo anterior, hemos creado una superclase llamada **Lenguaje** y una subclase llamada **Java**. Aquí, el método **visualizarInfo()** está presente tanto en **Lenguaje** como en **Java**.

Aquí usamos **visualizarInfo()** para imprimir la información. Sin embargo, imprime diferentes cosas en **Lenguaje** y **Java**.

Dependiendo del objeto utilizado para llamar al método, se imprime la información correspondiente.

Sobrecarga del Método

Dentro de una clase en Java, es posible crear varios métodos con el mismo nombre si difieren en los parámetros. Por ejemplo,

```

void func() { ... }
void func(int a) { ... }
float func(double a) { ... }
float func(int a, float b) { ... }

```

Esto se conoce como *sobrecarga del método* en Java. Aquí, el mismo método realizará diferentes operaciones dependiendo de sus parámetros definidos. Por ejemplo,

```
class Patron {  
  
    // método sin parámetro  
    public void visualizar() {  
        for (int i = 0; i < 10; i++) {  
            System.out.print("*");  
        }  
    }  
  
    // método con un único parámetro  
    public void visualizar(char simbolo) {  
        for (int i = 0; i < 10; i++) {  
            System.out.print(simbolo);  
        }  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
        Patron d1 = new Patron();  
  
        // invocar al método sin parámetro  
        d1.visualizar();  
        System.out.println("\n");  
  
        //invocar al método con un único parámetro  
        d1.visualizar('#');  
    }  
}
```

Salida:

```
*****  
  
#####
```

En el ejemplo anterior, hemos creado una clase llamada **Patron**. La clase contiene un método llamado **visualizar()** que está sobrecargado.

```
// método sin parámetros
```

```
visualizar() {...}  
  
// método con un único parámetro de tipo char  
visualizar(char simbolo) {...}
```

Aquí, la función principal de **visualizar()** es imprimir el patrón. Sin embargo, dependiendo de los argumentos que se introduzcan, el método realiza diferentes operaciones:

- Imprime un patrón de *, si no se introduce ningún argumento o
- Imprime el patrón del parámetro, si se pasa un único argumento de tipo **char**.

Operador sobrecargado

Algunos operadores en Java se comportan de manera diferente con diferentes operandos. Por ejemplo,

- El operador **+** está sobrecargado para realizar una suma numérica, así como para realizar una concatenación de cadena, y
- operadores como **&**, **|**, y **!** están sobrecargados para operaciones lógicas y bits.

Veamos cómo podemos lograr polimorfismo utilizando la sobrecarga del operador.

El operador **+** se utiliza para sumar dos entidades. Sin embargo, en Java, el operador **+** realiza dos tareas.

1. Cuando se usa **+** con números (números enteros y punto flotante), realiza una suma matemática. Por ejemplo,

```
int a = 5;  
int b = 6;  
  
// + con números  
int suma = a + b; // salida = 11
```

2. Cuando usamos el operador **+** con cadenas, en este caso realiza la concatenación de cadenas (unión dos cadenas). Por ejemplo,

```
String primero = "Java";  
String segundo = "Programación en";  
  
// + con cadenas  
Nombre = primero + segundo; // Salida = Programación en Java
```

De esta manera, podemos ver que el operador **+** está sobrecargado en Java para realizar dos operaciones: adición y concatenación.

Encapsulación en POO

La **encapsulación** es una de las características clave de la Programación Orientada a Objetos. La encapsulación se refiere a la agrupación de atributos y métodos dentro de una sola clase.

Evita que las clases externas accedan y cambien los atributos y métodos de una clase. Esta también ayuda a ocultar datos.

Anteriormente vimos que solo se puede acceder a las variables privadas (**private**) dentro de la misma clase (una clase externa no tiene acceso a ellas). Sin embargo, es posible acceder a ellas si proporcionamos métodos públicos (**public**) **get** y **set**.

El método **get** devuelve el valor de la variable y el método **set** establece un valor.

La sintaxis de ambos es que comienzan con **get** o **set**, seguido del nombre de la variable, con la primera letra en mayúscula:

```
class Persona {  
    private String nombre; // private = acceso restringido  
  
    // Getter  
    public String getNombre() {  
        return nombre;  
    }  
  
    // Setter  
    public void setNombre (String nuevoNombre) {  
        this.nombre = nuevoNombre;  
    }  
}
```

En el ejemplo anterior, el método **get** devuelve el valor de la variable **nombre**.

El método **set** toma un parámetro (**nuevoNombre**) y lo asigna a la variable **nombre**. La palabra clave **this** se utiliza para referirse al objeto actual.

Sin embargo, ya que la variable **nombre** es declarada privada (**private**), no podemos acceder a ella desde fuera de esta clase:

```
class Persona {  
    private String nombre; // private = acceso restringido  
  
    // Getter  
    public String getNombre() {  
        return nombre;  
    }  
  
    // Setter  
    public void setNombre (String nuevoNombre) {  
        this.nombre = nuevoNombre;  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Persona miObj = new Persona();  
        miObj.nombre = "Carlos";  
        System.out.println(miObj.nombre);  
    }  
}
```

Si la variable se hubiese declarado como pública (**public**), la salida sería:

```
Carlos
```

Sin embargo, al intentar acceder a la variable privada (**private**) **nombre**, obtendremos un error:

```
Main.java:19: error: nombre has private access in Persona  
    miObj.nombre = "Carlos";  
           ^  
Main.java:20: error: nombre has private access in Persona  
    System.out.println(miObj.nombre);
```

En su lugar, usamos los métodos **getNombre()** y **setNombre()** para acceder y actualizar la variable:

```

class Persona {
    private String nombre; // private = acceso restringido

    // Getter
    public String getNombre() {
        return nombre;
    }

    // Setter
    public void setNombre (String nuevoNombre) {
        this.nombre = nuevoNombre;
    }
}

public class Main {
    public static void main(String[] args) {
        Persona miObj = new Persona();
        // Estable el valor de la variable nombre a "Carlos"
        miObj.setNombre("Carlos");
        System.out.println(miObj.getNombre());
    }
}

```

¿Cómo es usada la encapsulación?

- En Java, la encapsulación nos ayuda a mantener juntos los campos y métodos relacionados, lo que hace que nuestro código sea más limpio y fácil de leer.
- Ayuda a tener control de los valores de nuestros atributos.
- Los métodos **getter** y **setter** proporcionan acceso de solo lectura o solo escritura a nuestros atributos de la clase.
- Ayuda a desacoplar componentes de un sistema. Por ejemplo, podemos encapsular el código en varios paquetes. Estos componentes desacoplados (paquetes) se pueden desarrollar, probar y depurar de forma independiente y simultánea. Y cualquier cambio en un componente en particular no tiene ningún efecto en otros componentes.
- También podemos ocultar datos mediante la encapsulación. Estos se mantienen ocultos de las clases externas.

VIII. Variables de Texto

Leer datos de entrada

La clase **Scanner** que se encuentra dentro del paquete **java.util** se usa para leer datos de entrada de diferentes fuentes como flujos de entrada, usuarios, archivos, etc. Tomemos un ejemplo.

```
import java.util.Scanner;

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

        // Crea un objeto a partir de la clase Scanner
        Scanner entrada = new Scanner(System.in);

        System.out.print("Ingresa tu nombre: ");

        // captura la entrada del teclado
        String nombre = entrada.nextLine();

        //imprime el nombre
        System.out.println("Mi nombre es " + nombre);

        // cierra la clase Scanner
        entrada.close();
    }
}
```

Salida:

```
Ingresa tu nombre: Guadalupe
Mi nombre es Guadalupe
```

En el ejemplo anterior, observa la línea

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

donde hemos creado un objeto llamado **entrada** a partir de la clase **Scanner**.

El parámetro **System.in** lo usamos para tomar el dato de entrada a través de la entrada estándar. Funciona de la misma manera en que se capturan datos de entrada del teclado.

Luego usamos el método **nextLine()** de la clase **Scanner** para leer/capturar una línea de texto del usuario.

Ahora que tienes una noción vaga de lo que se puede hacer con la clase Scanner, veamos algunas cosas más a fondo.

Como vimos en el ejemplo anterior, necesitamos importar el paquete **java.util.Scanner** antes de poder usar la clase **Scanner**.

Una vez que importamos el paquete, podremos crear objetos a partir de la clase **Scanner**:

```
// leer entrada de un flujo de entrada  
Scanner sc1 = new Scanner(InputStream entrada);  
  
// lee entrada de archivos  
Scanner sc2 = new Scanner(File archivo);  
  
// leed entrada de una cadena (string)  
Scanner sc3 = new Scanner(String cadena);
```

En el ejemplo anterior, hemos creado objetos de la clase **Scanner** que leerán la entrada de **InputStream** (flujos de entrada), **File** (archivos) y **String** (cadenas) respectivamente.

La clase **Scanner** proporciona varios métodos que nos permiten leer entradas de diferentes tipos.

Método	Función
nextBoolean()	Lee valores lógicos booleanos introducidos por el usuario.
nextByte()	Lee valores byte introducidos por el usuario.
nextDouble()	Lee valores double introducidos por el usuario.
nextFloat()	Lee valores float introducidos por el usuario.
nextInt()	Lee valores int introducidos por el usuario.
nextLine()	Lee valores String introducidos por el usuario.

nextLong()	Lee valores long introducidos por el usuario.
nextShort()	Lee valores short introducidos por el usuario.

Concatenación

El método **concat()** concatena (junta) dos cadenas (**string's**) y las devuelve.

Ejemplo.

```
class Main {
    public static void main(String[] args) {
        String str1 = "Programación en ";
        String str2 = "Java";

        // concatenación de str1 y str2
        System.out.println(str1.concat(str2));
    }
}
```

Salida:

Programación en Java

También puede ser:

```
class Main {
    public static void main(String[] args) {
        String str1 = "Programación en ";
        String str2 = "Java";

        // concatenación de str2 y str1
        System.out.println(str2.concat(str1));
    }
}
```

Salida:

JavaProgramación en

La sintaxis del método de concatenación de cadenas (**String's**) **concat()** es:

```
cadena.concat(String str)
```

donde **cadena** es un objeto de la clase **String**

Mayúsculas

El método **toUpperCase()** convierte todos los caracteres de una cadena (**String**) en caracteres en mayúsculas.

La sintaxis del método **toUpperCase()** es:

```
cadena.toUpperCase()
```

Ejemplo.

```
class Main {  
    public static void main(String[] args) {  
        String str1 = "Aprender Java es divertido";  
        String str2 = "Java123";  
  
        // convierte en mayúsculas los caracteres de la cadena  
        System.out.println(str1.toUpperCase());  
        System.out.println(str2.toUpperCase());  
    }  
}
```

Salida:

```
APRENDER JAVA ES DIVERTIDO  
JAVA123
```

Minúsculas

El método **toLowerCase()** convierte todos los caracteres de la cadena (**String**) en caracteres en minúsculas.

La sintaxis del método **toLowerCase()** es:

```
cadena.toLowerCase()
```

Ejemplo.

```
class Main {  
    public static void main(String[] args) {  
        String str1 = "PROGRAMACIÓN EN JAVA";  
  
        // convierte en minúsculas los caracteres de la cadena  
        System.out.println(str1.toLowerCase());  
  
    }  
}
```

Salida:

```
programación en java
```

Longitud

El método **length()** devuelve la longitud de la cadena (**string**).

La sintaxis del método **length()** es:

```
cadena.length()
```

Ejemplo.

```
class Main {  
    public static void main(String[] args) {  
        String str1 = "Programar en Java es divertido";  
        //cuenta número de caracteres en una cadena contando los espacios en blanco  
        System.out.println(str1.length());  
  
    }  
}
```

Salida:

30

Conversión

Podemos convertir un tipo de dato **String** en un tipo de dato **int** en java usando el método **Integer.parseInt()**.

Generalmente se usa si tenemos que realizar operaciones matemáticas en una cadena (**string**) que contiene un número. Cada vez que recibimos datos de TextField o TextArea, los datos ingresados se reciben como una cadena. Si los datos ingresados están en formato de número, necesitaríamos convertir la cadena a un dato de tipo **int**. Para hacerlo, usamos el método **Integer.parseInt()**.

Ejemplo.

```
public class Main
{
    public static void main (String args[])
    {
        //Declarando una variable de tipo String
        String s = "200";
        System.out.println (s + " pesos");
        //Convirtiendo la variable de tipo String en una de tipo int usando Integer.parseInt()
        int i = Integer.parseInt (s);
        //Imprimiendo el valor de i
        System.out.println (i);
        System.out.println (i + 300);
    }
}
```

Salida:

200 pesos
200
500

Podemos convertir un tipo de dato **int** en un tipo de dato **String** en java usando el método **String.valueOf()**

Generalmente se usa cuando tenemos que mostrar un número en un campo de texto **Textfield** debido a que todo es visualizado como una cadena (**string**).

Ejemplo.

```
public class Main
{
    public static void main (String args[])
    {
        int i = 200;
        String s = String.valueOf (i);
        System.out.println (i + 100); // imprimirá 300 porque i tiene un valor numérico
        System.out.println (s + 100); //imprimirá 200100 porque s es una cadena
    }
}
```

Salida:

```
300
200100
```

Leer textos en Java

La clase **Reader** del paquete **java.io** es una clase abstracta para leer transmisión de caracteres.

Dado que **Reader** es una clase abstracta, no es útil por sí misma. Sin embargo, sus subclases se pueden usar para leer datos.

Para usar la funcionalidad de **Reader**, podemos hacer uso de su subclase **FileReader**.

Para crear un **Reader**, primero debemos importar el paquete **java.io.Reader**. Una vez que importamos el paquete, así es como podemos crear el lector.

```
// Crea un Reader
Reader entrada = new FileReader();
```

Veamos un ejemplo de cómo podemos implementar **Reader** utilizando la clase **FileReader** usando un arreglo con un límite de caracteres.

Supongamos que tenemos un archivo llamado **archivotexto.txt** con el siguiente contenido.

Java is a high-level, class-based, object-oriented programming language that is designed to have as few implementation dependencies as possible. It is a general-purpose programming language intended to let programmers write once, run anywhere (WORA), meaning that compiled Java code can run on all platforms that support Java without the need to recompile.

Entonces el código en Java dentro de un archivo llamado **Main.java** para leer este archivo sería:

```
import java.io.Reader;
import java.io.FileReader;

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

        // Crea un arreglo de caracteres
        char[] arreglo= new char[400];

        try {
            // Crea un lector de texto usando FileReader
            Reader entrada = new FileReader("archivotexto.txt");

            // Revisa si el lector de texto está listo
            System.out.println("Hay datos en la transmision? " + entrada.ready());

            // Lee los caracteres
            entrada.read(arreglo);
            System.out.println("Datos en la transmision:");
            System.out.println(arreglo);

            // Cierra el lector de texto
            entrada.close();
        }

        catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Salida:

```
Hay datos en la transmision? true
```

```
Datos en la transmision:
```

```
Java is a high-level, class-based, object-oriented programming language that is designed to have as few implementation dependencies as possible. It is a general-purpose programming language intended to let programmers write once, run anywhere (WORA), meaning that compiled Java code can run on all platforms that support Java without the need to recompile.
```

Para leer datos de **archivotexto.txt**, hemos implementado estos métodos.

```
entrada.read();      // Para comenzar a leer datos  
entrada.close();   //Para cerrar el lector
```

Hay muchas clases disponibles en la API de Java que se pueden usar para leer y escribir archivos en Java: **FileReader**, **BufferedReader**, **Files**, **Scanner**, **FileInputStream**, **FileWriter**, **BufferedWriter**, **FileOutputStream**, etc., que se debe usar dependiendo de la versión Java con la que estés trabajando, si necesitas leer bytes o caracteres, y el tamaño del archivo/líneas con el que trabajes, etc.

LinkedList en Java

La clase **LinkedList** del *marco de colecciones Java* proporciona la funcionalidad en la estructura de datos de listas enlazadas.

Sintaxis

```
LinkedList<Type> listaEnlazada = new LinkedList<>();
```

Aquí, **type** indica el tipo de una lista enlazada. Por ejemplo,

```
// crea una lista enlazada de tipo de datos Entero  
LinkedList<Integer> listaEnlazada = new LinkedList<>();
```

```
// crea una lista enlazada de tipo de datos Cadena o String  
LinkedList<String> listaEnlazada = new LinkedList<>();
```

Hagamos un ejemplo concreto:

```
import java.util.LinkedList;

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

        // crea una lista enlazada
        LinkedList<String> animales = new LinkedList<>();

        // Add elements to LinkedList
        animales.add("Perro");
        animales.add("Gato");
        animales.add("Vaca");
        System.out.println("Lista enlazada: " + animales);
    }
}
```

Salida:

```
Lista enlazada: [Perro, Gato, Vaca]
```

En el ejemplo anterior, hemos creado una lista enlazada llamada **animales**.

Aquí, hemos usado el método **add()** para agregar elementos a la lista enlazada.

LinkedList proporciona varios métodos que nos permiten realizar diferentes operaciones en las listas enlazadas. Veremos cuatro operadores de **LinkedList** de uso común:

Agregar elementos a una lista enlazada

Podemos usar el método **add()** para agregar un elemento (nodo) al final de una lista enlazada. Por ejemplo,

```
import java.util.LinkedList;

class Main {
    public static void main(String[] args){
        // crea una lista enlazada
        LinkedList<String> animales = new LinkedList<>();

        // Uso del método add() sin el parámetro de índice
    }
}
```

```

animales.add("Perro");
animales.add("Gato");
animales.add("Vaca");
System.out.println("Lista enlazada: " + animales);

// Uso del método add() con el parámetro índice
animales.add(1, "caballo");
System.out.println("Actualización de la lista enlazada: " + animales);
}
}

```

Salida:

```

Lista enlazada: [Perro, Gato, Vaca]
Actualización de la lista enlazada: [Perro, caballo, Gato, Vaca]

```

Observa que en la declaración,

```
animales.add(1, "Caballo");
```

hemos utilizado el parámetro de número de índice. Es un parámetro opcional que especifica la posición donde se agrega el nuevo elemento.

Acceso a los elementos de una lista enlazada

El método **get()** de la clase **LinkedList** se usa para acceder a un elemento de la lista enlazada. Por ejemplo,

```

import java.util.LinkedList;

class Main {
    public static void main(String[] args) {
        LinkedList<String> lenguajes = new LinkedList<>();

        // agrega elementos de la lista enlazada
        lenguajes.add("Python");
        lenguajes.add("Java");
        lenguajes.add("JavaScript");
        System.out.println("Lista enlazada: " + lenguajes);

        // accede a un elemento de la lista enlazada
        String str = lenguajes.get(1);
    }
}

```

```
        System.out.print("Elemento en el índice 1: " + str);
    }
}
```

Salida:

```
Lista enlazada: [Python, Java, JavaScript]
Elemento en el índice 1: Java
```

En el ejemplo anterior, hemos utilizado el método **get()** con el parámetro **1**. Aquí, el método devuelve el elemento en el índice **1**.

Cambiar elementos de una lista enlazada

El método **set()** de la clase **LinkedList** se usa para cambiar elementos de una lista enlazada. Por ejemplo,

```
import java.util.LinkedList;

class Main {
    public static void main(String[] args) {
        LinkedList<String> lenguajes = new LinkedList<>();

        // agregar elementos a la lista enlazada
        lenguajes.add("Java");
        lenguajes.add("Python");
        lenguajes.add("JavaScript");
        lenguajes.add("Java");
        System.out.println("Lista enlazada: " + lenguajes);

        // cambiar el elemento en el índice 3
        lenguajes.set(3, "Kotlin");
        System.out.println("Actualización de la lista enlazada: " + lenguajes);
    }
}
```

Salida:

```
Lista enlazada: [Java, Python, JavaScript, Java]
Actualización de la lista enlazada: [Java, Python, JavaScript, Kotlin]
```

Observa que en la siguiente línea,

```
lenguajes.set(3, "Kotlin");
```

el método **set()** cambia el elemento en el índice 3 a **Kotlin**.

Eliminar elemento de una lista enlazada

El método **remove()** de la clase **LinkedList** se usa para eliminar un elemento de una lista enlazada. Por ejemplo,

```
import java.util.LinkedList;

class Main {
    public static void main(String[] args) {
        LinkedList<String> lenguajes = new LinkedList<>();

        //agregar elementos a una lista enlazada
        lenguajes.add("Java");
        lenguajes.add("Python");
        lenguajes.add("JavaScript");
        lenguajes.add("Kotlin");
        System.out.println("LinkedList: " + lenguajes);

        //quitar elemento del índice 1
        String str = lenguajes.remove(1);
        System.out.println("Quitar elemento: " + str);

        System.out.println("Actualización de lista enlazada: " + lenguajes);
    }
}
```

Salida:

```
LinkedList: [Java, Python, JavaScript, Kotlin]
Quitar elemento: Python
Actualización de lista enlazada: [Java, JavaScript, Kotlin]
```

Donde, el método **remove()** toma el número de índice como parámetro, y elimina el elemento especificado por el número de índice.

Escribir archivo de texto en Java

La clase **Writer** del paquete **java.io** es una clase abstracta que representa un flujo de caracteres.

Dado que **Writer** es una clase abstracta, no es útil por sí misma. Sin embargo, sus subclases se pueden usar para escribir datos.

Para crear un **Writer**, primero debemos importar el paquete **java.io.Writer**. Una vez que importamos el paquete, así es como podemos crear el **Writer**.

```
// Crear un Writer  
Writer salida = new FileWriter();
```

Aquí tenemos un ejemplo de cómo podemos implementar el Writer utilizando la clase **FileWriter**.

```
import java.io.FileWriter;  
import java.io.Writer;  
  
public class Main {  
  
    public static void main(String args[]) {  
  
        String datos = "Ésta es la información que tendrá el archivo de salida";  
  
        try {  
            // Crear un Writer usando la clase FileWriter  
            Writer salida = new FileWriter("archivosalida.txt");  
  
            // Escribe la cadena en el archivo  
            salida.write(datos);  
  
            // cierra el Writer  
            salida.close();  
        }  
  
        catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

En el ejemplo anterior, hemos creado un **writer** usando la clase **FileWriter**. El writer está vinculado con **archivosalida.txt**.

```
Writer salida = new FileWriter("archivosalida.txt");
```

Para escribir datos en **archivosalida.txt**, hemos implementado estos métodos.

```
salida.write();      // escribe los datos en el archivo  
salida.close();    //cierra el writer
```

Cuando ejecutamos el programa, **archivosalida.txt** se llena con el siguiente contenido.

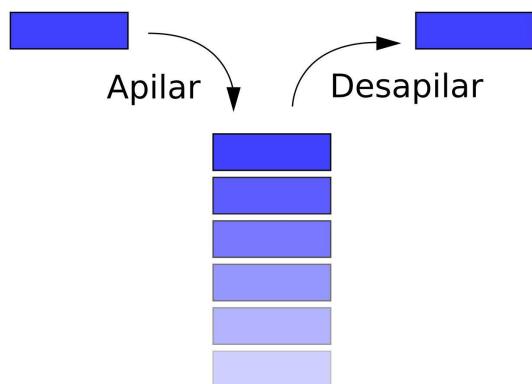
```
Ésta es la información que tendrá el archivo de salida
```

IX. Pilas

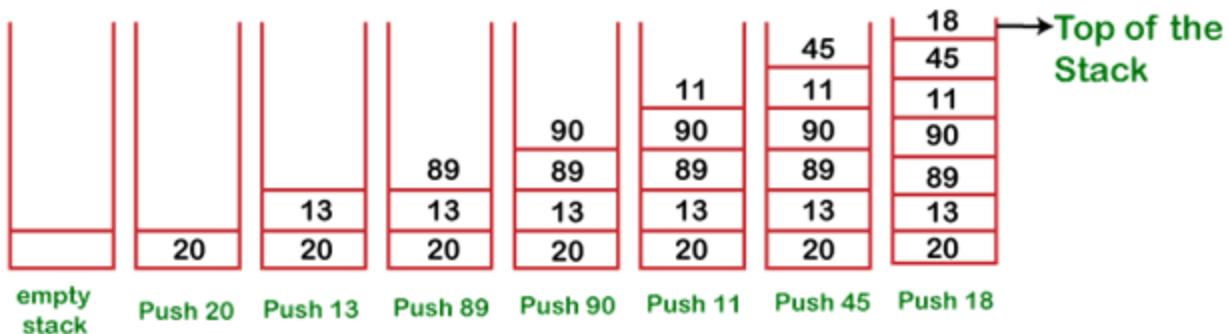
La **pila (stack)** es una estructura de datos lineal que se utiliza para almacenar la recopilación de objetos, siendo el modo de acceso a sus elementos de tipo LIFO (del inglés Last In, First Out, «último en entrar, primero en salir»).

Java Collection Framework proporciona muchas interfaces y clases para almacenar la colección de objetos. Una de ellas es la clase de pila (**stack**) que proporciona diferentes operaciones, como **push**, **pop**, **peek**, etc.

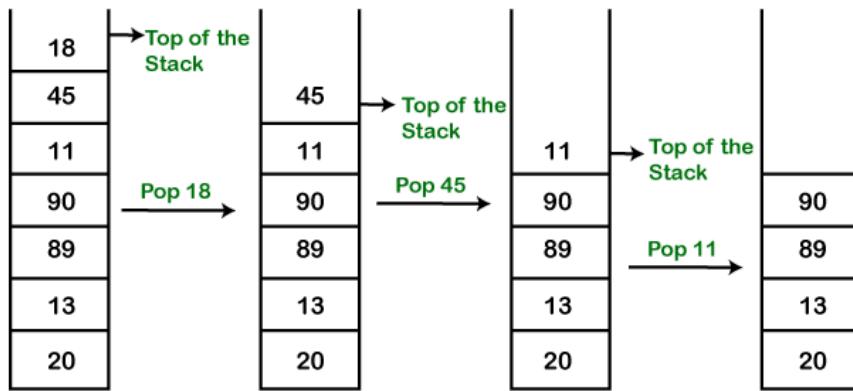
Las dos operaciones más importantes de la estructura de datos de pila son **push** (apilar, colocar) y **pop** (retirar, desapilar). La operación **push** inserta un elemento en la pila (**stack**) y **pop** elimina un elemento de la parte superior de la pila (**top of the stack**). Veamos cómo funcionan en las pilas.



Apilemos (**push**) 20, 13, 89, 90, 11, 45, 18, respectivamente en la pila de datos.



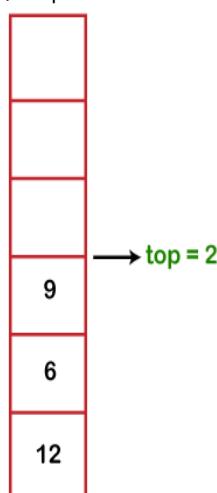
Desapilemos (**pop**) 18, 45 y 11 de la pila de datos.



Pila vacía (**empty stack**): una pila que no tiene ningún elemento se le conoce como una pila vacía (**empty stack**). Cuando la pila está vacía, el valor de la variable de la parte superior es **-1**.



Cuando apilamos (**push**) un elemento en la pila, la parte superior aumenta en **1**. Por ejemplo, en la siguiente figura tenemos que para:
push 12, top=0, push 6, top=1 y push 9, top=2



Java collections framework tiene una clase llamada **Stack** que proporciona la funcionalidad de la estructura de datos apilados.

Para crear una pila, primero debemos importar el paquete **java.util.stack**. Una vez que importamos el paquete, así es como podemos crear una pila en Java:

```
Stack<Type> pila = new Stack<>();
```

Aquí, el **type** indica el tipo de datos que se almacenan en la pila. Por ejemplo,

```
// crea una pila de datos de tipo entero  
Stack<Integer> pila = new Stack<>();  
  
// crea una pila de datos de tipo cadena o String  
Stack<String> pila = new Stack<>();
```

El método push()

Para apilar un elemento a la parte superior de la pila, usamos el método `push ()`. Por ejemplo,

```
import java.util.Stack;  
  
class Main {  
    public static void main(String[] args) {  
        Stack<String> animales= new Stack<>();  
  
        //agrega elementos a la pila  
        animales.push("Perro");  
        animales.push("Caballo");  
        animales.push("Gato");  
  
        System.out.println("Pila: " + animales);  
    }  
}
```

Salida:

```
Pila: [Perro, Caballo, Gato]
```

*El método **pop()***

Para eliminar un elemento de la parte superior de la pila (**top of the stack**), usamos el método **pop()**. A continuación tenemos un ejemplo de su aplicación:

```
import java.util.Stack;

class Main {
    public static void main(String[] args) {
        Stack<String> animales= new Stack<>();

        //Aregar elementos a la pila
        animales.push("Perro");
        animales.push("Caballo");
        animales.push("Gato");
        System.out.println("Pila inicial: " + animales);

        // Quitar elementos de la pila
        String element = animales.pop();
        System.out.println("Quitar elemento: " + element);
        System.out.println("Pila actualizada: " + animales);
    }
}
```

Salida:

```
Pila inicial: [Perro, Caballo, Gato]
Quitar elemento: Gato
Pila actualizada: [Perro, Caballo]
```

*El método **peek()***

El método **peek()** devuelve el objeto desde la parte superior de la pila. Por ejemplo,

```
import java.util.Stack;

class Main {
    public static void main(String[] args) {
        Stack<String> animales= new Stack<>();

        // Agrega elementos a la pila
```

```

animales.push("Perro");
animales.push("Caballo");
animales.push("Gato");
System.out.println("Pila: " + animales);

// Accede al elemento de la parte superior de la pila
String elemento = animales.peek();
System.out.println("Elemento de la parte superior de la pila: " + elemento);

}
}

```

Salida:

```

Pila: [Perro, Caballo, Gato]
Elemento de la parte superior de la pila: Gato

```

El método search()

Para buscar un elemento en la pila, usamos el método **search()**. Devuelve la posición del elemento contando desde la parte superior de la pila. Por ejemplo,

```

import java.util.Stack;

class Main {
    public static void main(String[] args) {
        Stack<String> animales= new Stack<>();

        // Add elements to Stack
        animales.push("Perro");
        animales.push("Caballo");
        animales.push("Gato");
        animales.push("Conejo");
        System.out.println("Pila: " + animales);

        // Busca en la pila al elemento
        int posicion = animales.search("Caballo");
        System.out.println("Posición del Caballo: " + posicion);
    }
}

```

Salida:

```
Pila: [Perro, Caballo, Gato, Conejo]
Posición del Caballo: 3
```

El método empty()

Para verificar si una pila está vacía o no, usamos el método **empty()**. Por ejemplo,

```
import java.util.Stack;

class Main {
    public static void main(String[] args) {
        Stack<String> animales= new Stack<>();

        // Agrega elementos a la pila
        animales.push("Perro");
        animales.push("Caballo");
        animales.push("Gato");
        System.out.println("Pila: " + animales);

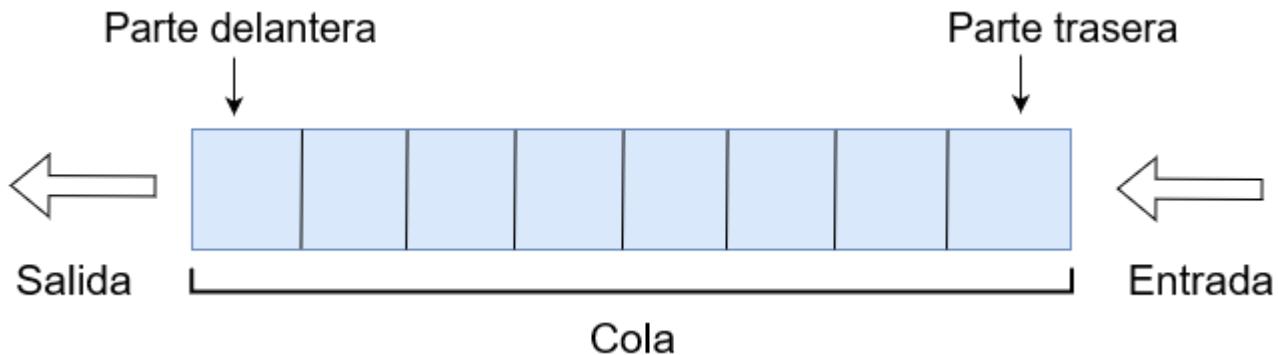
        // Revisa si la pila está vacía
        boolean resultado = animales.empty();
        System.out.println("¿La pila está vacía? " + resultado);
    }
}
```

Salida:

```
Pila: [Perro, Caballo, Gato]
¿La pila está vacía? false
```

X. Colas

Una **cola (queue)** es una estructura de datos, caracterizada por ser una secuencia de elementos en la que la operación de inserción **push** se realiza por un extremo y la operación de extracción **pull** por el otro. También se le llama estructura **FIFO** (del inglés First In First Out), debido a que el primer elemento en entrar será también el primero en salir.



Sintaxis

```
import java.util.Queue;  
Queue<TipoDato> cola = new LinkedList<TipoDato>();
```

Donde el **TipoDato** es el tipo de datos que se almacenará en la cola, y la clase de cola es una clase que implementa la interfaz de cola.

En Java, debemos importar el paquete **java.util.queue** para usar la cola (**queue**).

Algunos de los métodos comúnmente utilizados por la interfaz cola (**queue**) son:

- **add()**: inserta el elemento especificado en la cola. Si el proceso es exitoso, **add()** devuelve verdadero (**true**), si no, lanza una excepción.
- **offer()**: inserta el elemento especificado en la cola. Si el proceso es exitoso, **offer()** devuelve verdadero (**true**), si no devuelve falso (**false**).
- **element()**: devuelve la parte delantera de la cola. Lanza una excepción si la cola está vacía.
- **peek()**: devuelve la parte delantera de la cola. Devuelve **null** si la cola está vacía.
- **remove()**: devuelve y elimina la parte delantera de la cola. Lanza una excepción si la cola está vacía.
- **poll()**: devuelve y elimina la parte delantera de la cola. Devuelve **null** si la cola está vacía.

```
import java.util.Queue;
import java.util.LinkedList;

class Main {

    public static void main(String[] args) {
        // Crea una cola usando la LinkedList
        Queue<Integer> numeros = new LinkedList<>();

        // agrega elementos a la cola
        numeros.offer(1);
        numeros.offer(2);
        numeros.offer(3);
        System.out.println("Cola: " + numeros);

        // accede a los elementos de la cola
        int numeroAccedido = numeros.peek();
        System.out.println("Elemento accedido: " + numeroAccedido);

        // quita elementos de la cola
        int numeroRemovido = numeros.poll();
        System.out.println("Elemento removido: " + numeroRemovido);

        System.out.println("Cola actualizada: " + numeros);
    }
}
```

Salida:

```
Cola: [1, 2, 3]
Elemento accedido: 1
Elemento removido: 1
Cola actualizada: [2, 3]
```

XI. Métodos de Ordenamiento

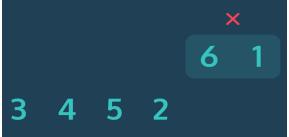
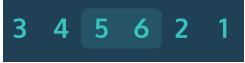
Ordenamiento de burbuja

El **ordenamiento de burbuja** es un algoritmo simple para ordenar una lista de N números en orden ascendente. El ordenamiento de burbuja compara dos elementos adyacentes y los intercambia si están en el orden ascendente equivocado.

Este algoritmo consiste en una iteración externa y una iteración interna. En la iteración interna, el primer y el segundo elementos se comparan y se intercambian de tal manera que el segundo elemento tiene un valor más grande que el primero. Esto se repite para los subsecuentes, segundo y tercero, par de elementos, y así sucesivamente hasta que se compara el último par de elementos (N-2, N-1). Al final de la iteración interna, el elemento más grande aparece al último. Esto se repite para todos los elementos de la lista en la iteración externa.

Ejemplo de la iteración interna para el primer ciclo externo:

1.		9.	
2.		10.	
3.		11.	
4.		12.	
5.		13.	
6.		14.	

7.			
8.		16.	

El mismo proceso continúa para las iteraciones restantes.

Después de cada iteración, el elemento más grande entre los elementos no organizados se coloca al final.

Algoritmo de ordenamiento de burbujas

OrdenamientoBurbuja(arreglo)

```
para i ← 1 hasta indiceDelUltimoElementoDesordenado – 1
    si elementoIzquierda > elementoDerecha
        permutar elementoIzquierda y elementoDerecha
fin ordenamientoBurbuja
```

Ejemplo.

```
// Ordenamiento de burbuja en Java

//paquete para darle cierta presentación a los arreglos
import java.util.Arrays;

class Main {

    // realización del ordenamiento de burbuja
    static void ordenamientoBurbuja(int arreglo[]) {
        int longitud = arreglo.length;

        // iteración que accede a cada elemento del arreglo
        for (int i = 0; i < longitud - 1; i++)

            // iteración que compara los elementos del arreglo
            for (int j = 0; j < longitud - i - 1; j++)

                // compara dos elementos adyacentes
                // cambia > por < para ordenar de manera ascendente
                if (arreglo[j] > arreglo[j + 1]) {
```

```

// la permutación ocurre si los elementos
// no se encuentran en el orden adecuado
int variableAuxiliar = arreglo[j];
arreglo[j] = arreglo[j + 1];
arreglo[j + 1] = variableAuxiliar;
}
}

public static void main(String args[]) {

    int[] datos = { -2, 45, 0, 11, -9 };
    System.out.println("Arreglo desordenado:");
    System.out.println(Arrays.toString(datos));
    // invocando el método usando el nombre de la clase
    Main.ordinamientoBurbuja(datos);

    System.out.println("Arreglo ordenado de manera ascendente:");
    System.out.println(Arrays.toString(datos));
}
}

```

Salida:

```

Arreglo desordenado:
[-2, 45, 0, 11, -9]
Arreglo ordenado de manera ascendente:
[-9, -2, 0, 11, 45]

```

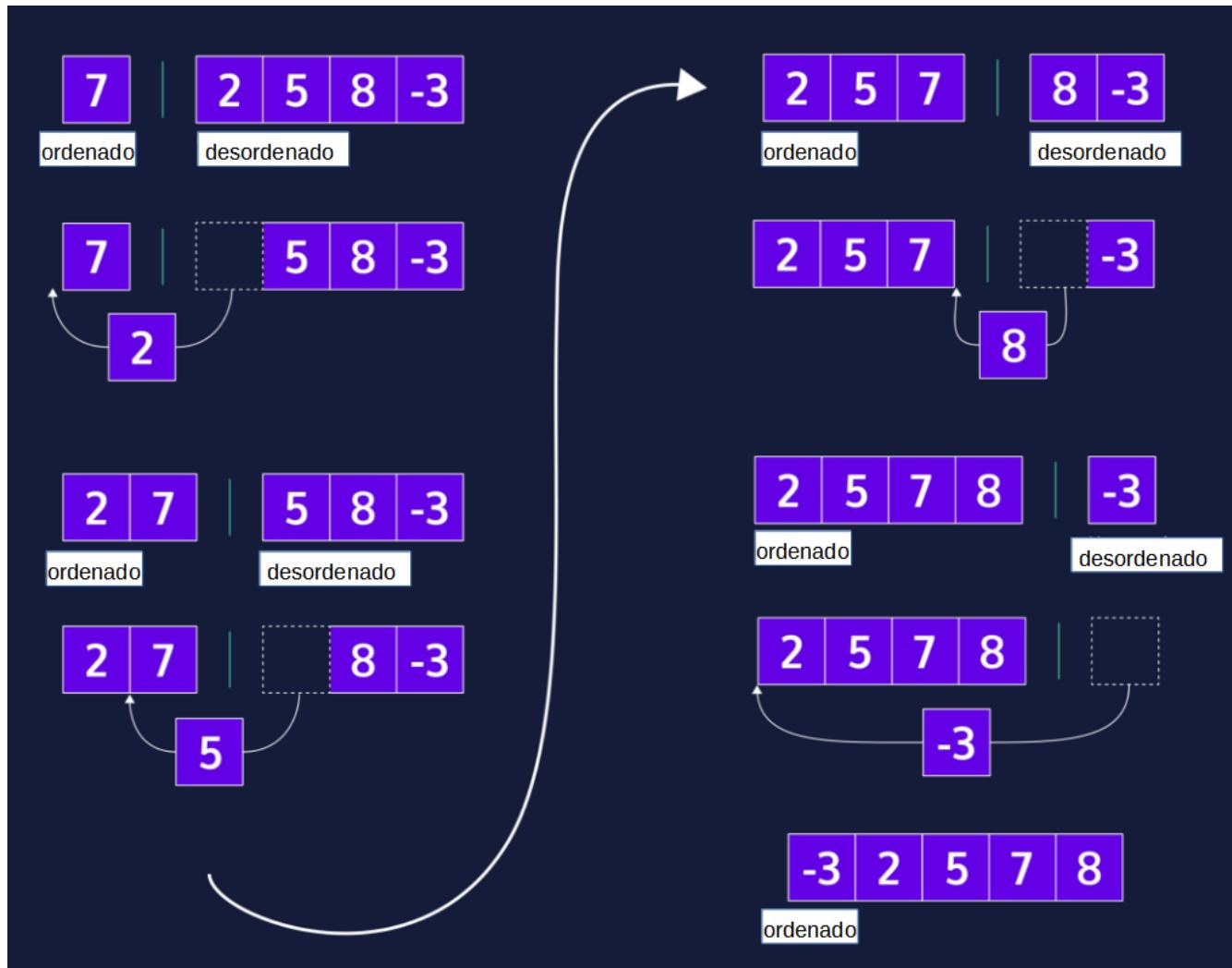
Ordenamiento por inserción

El **ordenamiento por inserción** es un algoritmo de ordenamiento que construye un arreglo ordenado al finalizar procediendo de a un elemento a la vez. En cada iteración a través de un arreglo dado, el algoritmo de inserción toma un elemento y encuentra la ubicación al que pertenece y lo inserta.

El ordenamiento por inserción funciona dividiendo el arreglo de entrada en dos listas virtuales: una sub lista ordenada y una sub lista desordenada.

La sub lista ordenada contiene inicialmente el primer elemento del arreglo de entrada. El resto de los elementos constituyen la sub lista desordenada.

El ordenamiento por inserción irá a través de los elementos de la sub lista desordenada de a uno por uno, esencialmente eliminando un elemento de la sub lista desordenada e insertando este en la posición correcta en la sub lista ordenada cambiando todos los elementos en la sub lista ordenada que sean más grandes que el elemento que se está ordenando.



Algoritmo de ordenamiento por inserción

```

ordenamientoInsercion(arreglo)
  marcar primer elemento como ordenado
  para cada elemento desordenado X
    'extraer' el elemento X
    para j ←ultimoIndiceOrdenado hasta 0
      si el actual elemento j > X
        mueve al elemento ordenado 1 a la derecha
      rompe el bucle e inserta X aquí
  fin ordenamientoInsercion

```

Ejemplo.

```
// Ordenamiento por inserción en Java

import java.util.Arrays;

class OrdenamientoInsercion{

    void ordenamientoInsercion(int arreglo[]) {
        int longitud = arreglo.length;

        for (int paso = 1; paso < longitud; paso++) {
            int key = arreglo[paso];
            int j = paso - 1;

            // Compara key con cada uno de los elementos a su izquierda
            // hasta que un elemento más pequeño que él es encontrado.
            // Para un orden descendiente, cambiar key<arreglo[j] por key>arreglo[j].
            while (j >= 0 && key < arreglo[j]) {
                arreglo[j + 1] = arreglo[j];
                j--;
            }

            // Coloca a key después del elemento más pequeño que este
            arreglo[j + 1] = key;
        }
    }

    public static void main(String args[]) {
        int[] datos = { 9, 5, 1, 4, 3 };
        System.out.println("Arreglo desordenado: ");
        System.out.println(Arrays.toString(datos));
        OrdenamientoInsercion is = new OrdenamientoInsercion();
        is.ordenamientoInsercion(datos);
        System.out.println("Arreglo ordenado de manera ascendente: ");
        System.out.println(Arrays.toString(datos));
    }
}
```

Salida:

```
Arreglo desordenado:
[9, 5, 1, 4, 3]
```

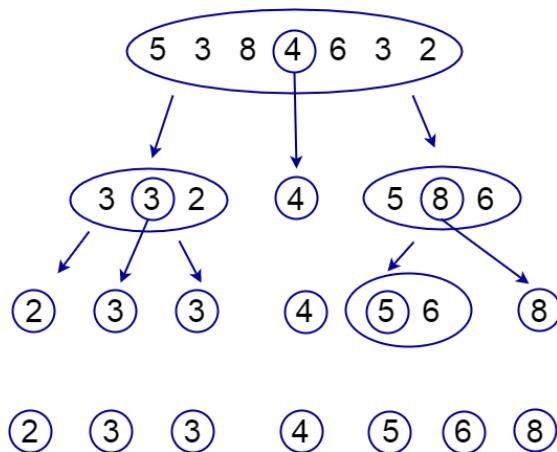
Arreglo ordenado de manera ascendente:
[1, 3, 4, 5, 9]

Ordenamiento rápido

El **ordenamiento rápido** es un algoritmo recursivo eficiente para ordenar arreglos o listas de valores. El algoritmo es un tipo de ordenamiento, donde los valores se ordenan mediante una operación de comparación como $>$ o $<$.

El ordenamiento rápido es un método para ordenar un arreglo al dividirlo repetidamente en sub-arreglos por medio de:

1. Seleccionando un elemento del arreglo inicial. Este elemento se llama *elemento pivote*.
2. Comparando cada elemento en el arreglo con el elemento pivote, dividimos los elementos en dos sub arreglos, los que son mayores y los que son menores al elemento pivote.
3. Este proceso continúa hasta que cada subarreglo contiene un solo elemento.
4. En este punto, los elementos ya están ordenados. Finalmente, los elementos se combinan para formar un arreglo ordenado.



Algoritmo de ordenamiento rápido

```
ordenamientoRapido(arreglo, indiceMasIzquierda, indiceMasDerecha)
if (indiceMasIzquierda < indiceMasDerecha)
    indicePivote ← particion(arreglo,indiceMasIzquierda, indiceMasDerecha)
    ordenamientoRapido(arreglo, indiceMasIzquierda , indicePivote - 1)
    ordenamientoRapido(arreglo, indicePivote, indiceMasDerecha)

particion(arreglo, indiceMasIzquierda, indiceMasDerecha)
set indiceMasDerecha as indicePivote
indiceAlmacenado ← indiceMasIzquierda - 1
for i ← indiceMasIzquierda + 1 to indiceMasDerecha
if elemento[i] < elementoPivote
    permuta elemento[i] y elemento[indiceAlmacenado ]
    indiceAlmacenado++
permuta elementoPivote y elemento[indiceAlmacenado +1]
return indiceAlmacenado + 1
```

Ejemplo.

```
//Método de ordenamiento rápido en Java

import java.util.Arrays;

class OrdenamientoRapido{

    // Método para encontrar la posición en que se hará la partición
    static int particion(int arreglo[], int bajo, int alto) {

        // Elige el elemento más a la derecha como pivote
        int pivot = arreglo[alto];

        // Puntero hacia un elemento mayor
        int i = (bajo - 1);

        //recorre todos los elementos
        //compara cada elemento con el pivote
        for (int j = bajo; j < alto; j++) {
            if (arreglo[j] <= pivot) {

                // si se encuentra un elemento más pequeño que pivote
                // lo permuta por el elemento más grande apuntado por i
                i++;

                // permutando el elemento en i con el elemento en j
                int temporal = arreglo[i];
```

```

arreglo[i] = arreglo[j];
arreglo[j] = temporal;
}

}

// permuta el elemento pivote con el elemento más grande especificado por i
int temporal = arreglo[i + 1];
arreglo[i + 1] = arreglo[alto];
arreglo[alto] = temporal;

//Devuelve la posición donde la partición es realizada
return (i + 1);
}

static void ordenamientoRapido(int arreglo[], int bajo, int alto) {
    if (bajo < alto) {

        // halla el elemento pivote de tal manera que
        // los elementos más pequeños que el pivote están a la izquierda
        // y los elementos más grandes que pivote están a la derecha
        int pi = particion(arreglo, bajo, alto);

        // invocación recursiva a la izquierda del pivote
        ordenamientoRapido(arreglo, bajo, pi - 1);

        // invocación recursiva a la derecha del pivote
        ordenamientoRapido(arreglo, pi + 1, alto);
    }
}
}

// Clase Main
class Main {
    public static void main(String args[]) {

        int[] datos = { 8, 7, 2, 1, 0, 9, 6 };
        System.out.println("Arreglo sin ordenar");
        System.out.println(Arrays.toString(datos));

        int longitud = datos.length;

        // invocación de ordenamientoRapido sobre los datos del arreglo
        OrdenamientoRapido.ordenamientoRapido(datos, 0, longitud - 1);

        System.out.println("Arreglo ordenado de manera ascendente: ");
    }
}

```

```
        System.out.println(Arrays.toString(datos));  
    }  
}
```

Salida:

```
Arreglo sin ordenar  
[8, 7, 2, 1, 0, 9, 6]  
Arreglo ordenado de manera ascendente:  
[0, 1, 2, 6, 7, 8, 9]
```

XII. Entorno de Desarrollo para crear aplicaciones gráficas

¿Qué es exactamente un IDE?

Un **IDE**, o **entorno de desarrollo integrado**, es un programa que te ayudará a escribir software. Un IDE lo ayuda a organizar sus proyectos de software, escribir código y luego probarlo y depurarlo (debug). Los IDE populares incluyen Netbeans, Eclipse, IntelliJ y Microsoft Visual Studio.

Un IDE generalmente incluye características que te ayudarán a:

- Escribir código
- Código de compilación o de paquete
- Depura tu aplicación

¿Necesitas un **IDE** para escribir código Java?

No, no necesitas escribir código Java en un IDE. Puedes usar cualquier editor de texto que deseas y luego compilar el código con **javac**.

Cuando eres un programador principiante de Java, es mejor que aprendas a escribir código usando un editor de texto sin formato.

Te ayudará a ver los conceptos básicos del lenguaje y te ayudará en el futuro, porque sabrás exactamente cómo se crea un programa Java desde cero.

Pero una vez que hayas pasado de la etapa de principiante, probablemente sería bueno hacer uso de algún tipo de programa de escritorio gráfico que te ayude a escribir código.

Un **IDE** te ayudará a organizar tus proyectos, ejecutar pruebas, compilar el código, formatearlo correctamente y mucho más.

Algunos de los principales **IDE** de Java que se utilizan hoy en día son:

- IntelliJ IDEA
- Eclipse
- NetBeans
- Oracle JDeveloper
- Visual Studio Code

Interfaces gráficas con Swing

Swing es una biblioteca de interfaces gráficas de usuario (GUI) para Java. Viene incluida con el entorno de desarrollo de Java (JDK). Extiende a otra librería gráfica más antigua llamada AWT.

Paquetes que contiene:

- **javax.swing**
- **java.awt**
- **java.awt.event**

Creación de ventanas

La clase **JFrame** proporciona operaciones para manipular ventanas.

Constructores:

- **JFrame()**
- **JFrame(String titulo)**

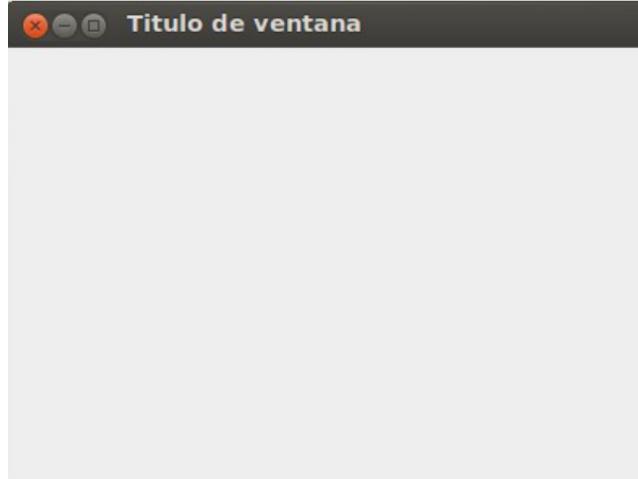
Una vez creado el objeto de ventana, hay que:

- Establecer su tamaño.
- Establecer la acción de cierre.
- Hacerla visible.

```
import javax.swing.*;  
  
public class VentanaTest  
{  
    public static void main (String[] args)  
    {  
        JFrame f = new JFrame ("Titulo de ventana");  
        f.setSize (400, 300);  
        f.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);  
        f.setVisible (true);  
    }  
}
```

Acciones de cierre:

- **JFrame.EXIT_ON_CLOSE**: Abandona aplicación.
- **JFrame.DISPOSE_ON_CLOSE**: Libera los recursos asociados a la ventana.
- **JFrame.DO NOTHING ON CLOSE**: No hace nada.
- **JFrame.HIDE ON CLOSE**: Cierra la ventana, sin liberar sus recursos.



Es usual extender la clase JFrame, y realizar las operaciones de inicialización en su constructor.

```
public class MiVentana extends JFrame
{
    public MiVentana ()
    {
        super ("Titulo de ventana");
        setSize (400, 300);
        setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
    }
}
```

```
public class VentanaTest
{
    public static void main (String[]args)
    {
        MiVentana v = new MiVentana ();
        v.setVisible (true);
    }
}
```

Componentes de una nueva ventana

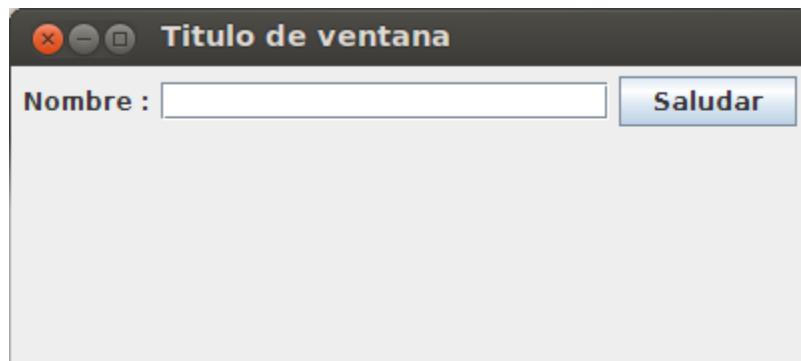
	JButton
	JLabel
	JTextField
	JCheckBox
	JRadioButton

Tras crear uno de estos componentes con **new**, ha de añadirse al **contentPane** de la ventana correspondiente mediante su método **add**.

Ejemplo. Añadiendo componentes

```
public class MiVentana extends JFrame
{
    public MiVentana ()
    {
        super ("Titulo de ventana");
        setSize (400, 300);
        setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
        Container cp = getContentPane ();
        cp.setLayout (new FlowLayout ());
        JLabel etiqueta = new JLabel ("Nombre: ");
        JTextField texto = new JTextField (20);
        JButton boton = new JButton ("Saludar");
        cp.add (etiqueta);
        cp.add (texto);
        cp.add (boton);
    }
}
```

Resultado



Layouts de una Aplicación Gráfica

En Java no es habitual indicar explícitamente la posición de los componentes de la interfaz dentro de la ventana.

Los layout managers se encargan de colocar los componentes de la interfaz de usuario en la ventana contenedora.

Especifican la posición y el tamaño de dichos componentes.

- FlowLayout
- GridLayout
- BorderLayout
- GridBagConstraints
- ...

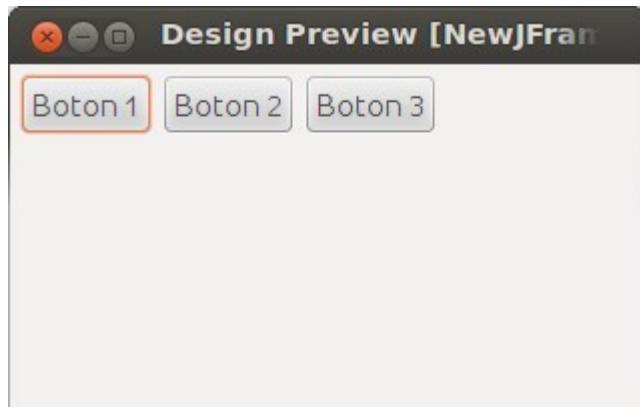
FlowLayout

Coloca los elementos uno a continuación de otro, de manera similar a la colocación de palabras en un procesador de textos.

Métodos:

- **setAlignment(int alineacion)**
- **setHgap(int separacion)**

- **setVgap(int separacion)**



GridLayout

Coloca los componentes de la interfaz en forma de rejilla.

El orden en que se añadan los componentes determina su posición en la rejilla.

Constructor:

- **GridLayout(int filas, int columnas)**

Métodos:

- **setHgap(int separacion)**
- **setVgap(int separacion)**



```
public class MiVentana2 extends JFrame
{
    public MiVentana2 ()
    {
        super ("Titulo de ventana");
        setSize (400, 300);
    }
}
```

```

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
Container cp = getContentPane ();
GridLayout gl = new GridLayout (4, 3);
gl.setHgap (5);
gl.setVgap (5);
cp.setLayout (gl);
for (int i = 1; i <= 9; i++)
{
    cp.add (new JButton (String.valueOf (i)));
}
cp.add (new JButton ("*"));
cp.add (new JButton ("0"));
cp.add (new JButton ("#"));
}
}

```



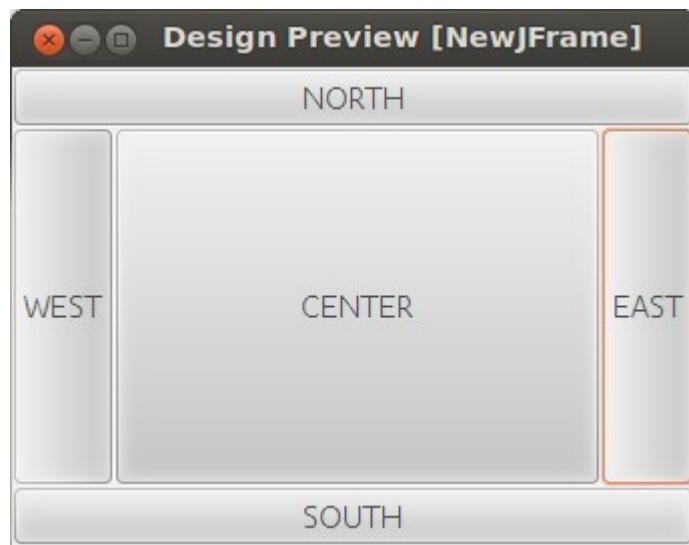
BorderLayout

Coloca y cambia de tamaño sus componentes para que se ajusten a los bordes y parte central de la ventana.

Métodos:

- **setHgap(int separacion)**
- **setVgap(int separacion)**

Al añadir un elemento a la ventana, hay que especificar su colocación:



```
JButton b = new JButton(...);  
getContentPane().add(b, BorderLayout.EAST)
```

Interfaces complejas: JPanel

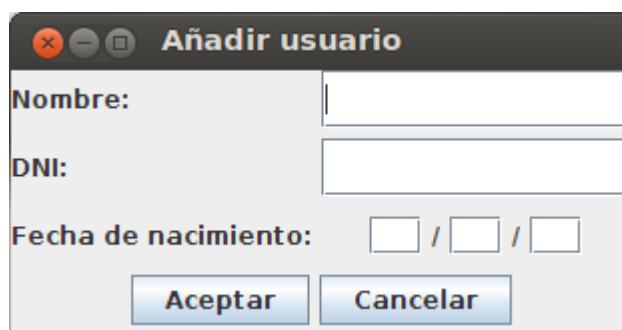
Un panel es un componente con un **layout manager** propio, y que puede contener varios componentes en su interior.

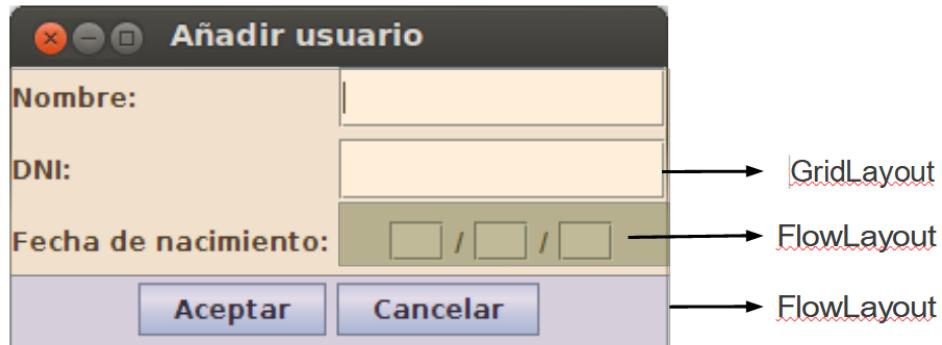
Constructor:

- **JPanel()**

Métodos:

- **void setLayout(LayoutManager lm)**
- **void add(JComponent componente)**





```

public MiVentana3 (){
    super ("AC1adir usuario");
    setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
    // Panel de fecha
    JPanel panelFecha = new JPanel ();
    panelFecha.setLayout (new FlowLayout ());
    panelFecha.add (new JTextField (2));
    panelFecha.add (new JLabel ("/"));
    panelFecha.add (new JTextField (2));
    panelFecha.add (new JLabel ("/"));
    panelFecha.add (new JTextField (2));
    // Panel de datos
    JPanel panelDatos = new JPanel ();
    GridLayout gl = new GridLayout (3, 2, 0, 5);
    panelDatos.setLayout (gl);
    panelDatos.add (new JLabel ("Nombre:"));
    panelDatos.add (new JTextField (10));
    panelDatos.add (new JLabel ("DNI:"));
    panelDatos.add (new JTextField (10));
    panelDatos.add (new JLabel ("Fecha de nacimiento: "));
    panelDatos.add (panelFecha);
    // Panel de botones
    JPanel panelBotones = new JPanel ();
    panelBotones.setLayout (new FlowLayout ());
    panelBotones.add (new JButton ("Aceptar"));
    panelBotones.add (new JButton ("Cancelar"));
    Container cp = getContentPane ();
    cp.add (panelDatos, BorderLayout.CENTER);
    cp.add (panelBotones, BorderLayout.SOUTH);
}

```

Interfaces complejas: GridBagLayout

Más flexible que **GridLayout**.

Cada componente ha de tener asociado un objeto de la clase **GridBagConstraints**. La asociación se producirá en el método **add**.



```
JButton b = new JButton("Aceptar");
GridBagConstraints gbc = new GridBagConstraints(...);
getContentPane().add(b, gbc);
```

GridBagConstraints

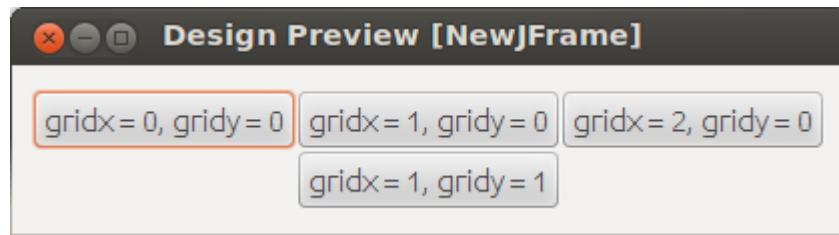
Atributos públicos:

- int **gridx**, **gridy**
- int **gridwidth**, **gridheight**
- double **weightx**, **weighty**
- int **fill**
- int **anchor**
- Insets **insets**
- int **ipadx**, **ipady**

Pueden ser inicializados en el constructor.

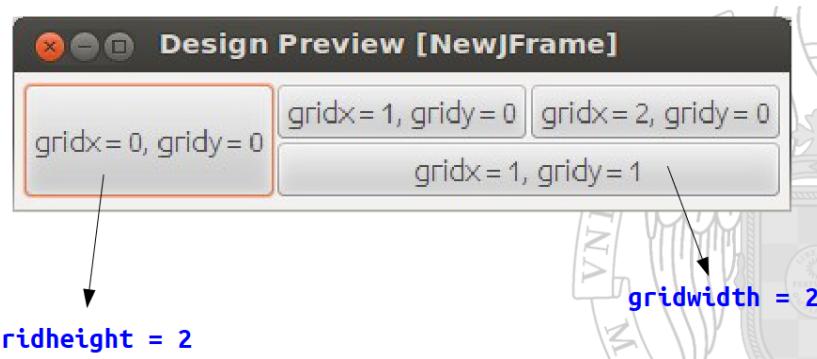
Atributos públicos:

- int gridx, gridy



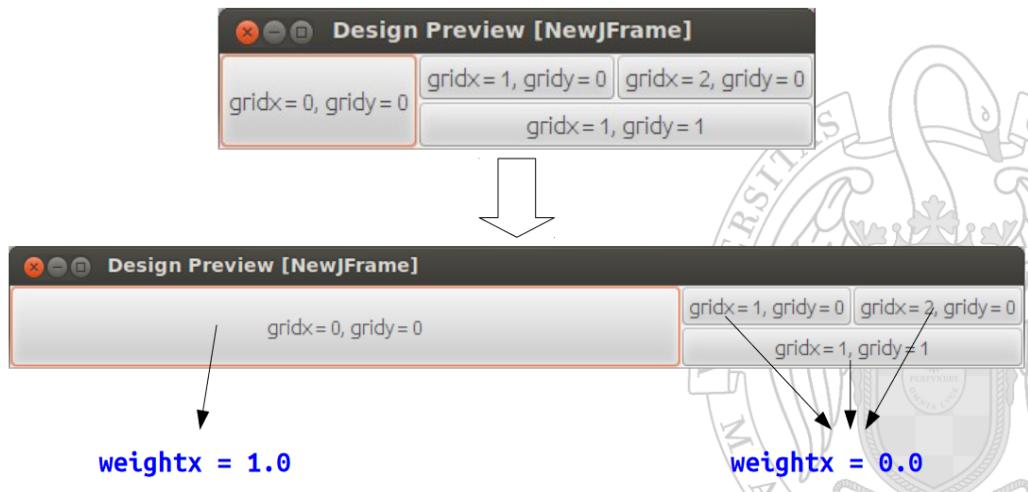
Atributos públicos:

- **int gridwidth, gridheight**



Atributos públicos:

- **double weightx, weighty**

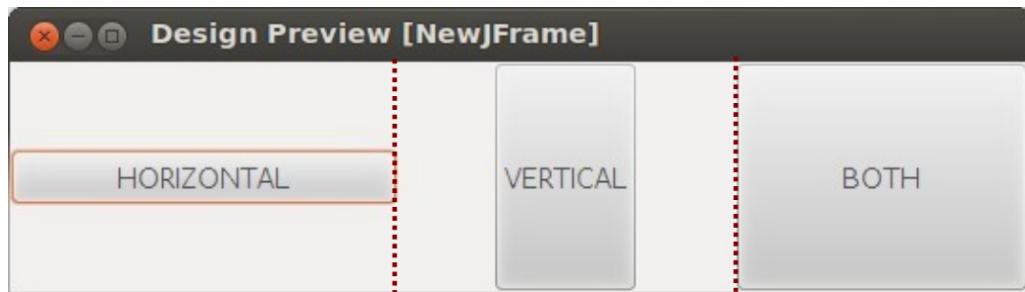


Atributos públicos:

- int fill

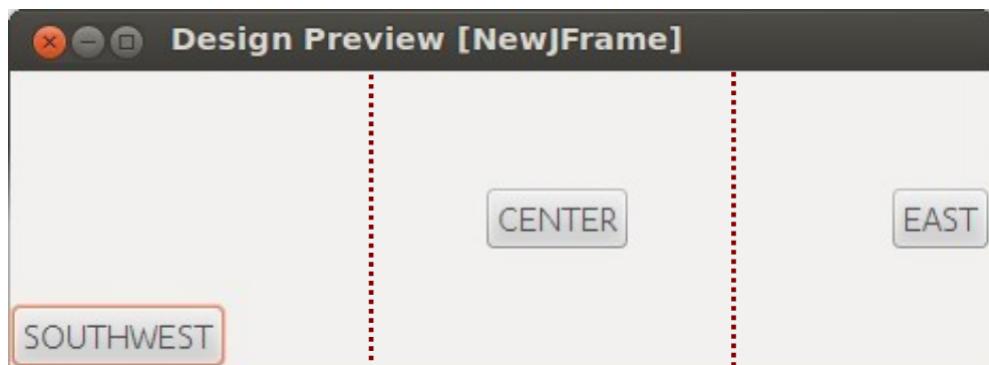
Puede ser:

- GridBagConstraints.HORIZONTAL
- GridBagConstraints.VERTICAL
- GridBagConstraints.BOTH



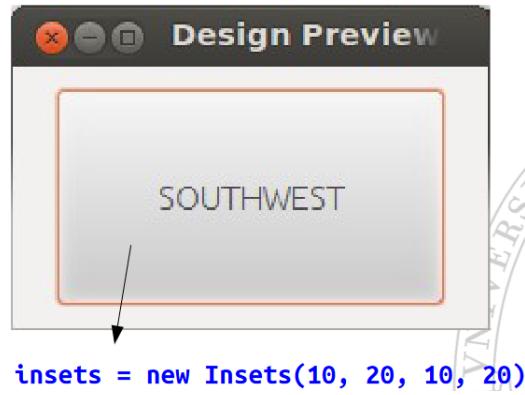
Atributos públicos:

- int anchor



Atributos públicos:

- Insets insets



Atributos públicos:

- **int padx, int pady**

Especifican cuánto espacio añadir a la anchura/altura mínima de los componentes.

Eventos en aplicaciones gráficas

Un **evento** es un suceso que ocurre como consecuencia de la interacción del usuario con la interfaz gráfica.

- Pulsación de un botón.
- Cambio del contenido en un cuadro de texto.
- Deslizamiento de una barra.
- Activación de un JCheckBox.
- Movimiento de la ventana.

Pulsación de un botón

La clase JButton tiene un método:

- **void addActionListener(ActionListener l)**

Que especifica el objeto (manejador de evento) que se encargará de tratar el evento de pulsación del botón.

Este objeto ha de interpretar la interfaz **ActionListener** (paquete **java.awt.event**)

```
JButton b = new JButton("Aceptar");
GridBagConstraints gbc = new GridBagConstraints(...);
getContentPane().add(b, gbc);
```

Cuando el usuario pulse el botón, se llamará al método **actionPerformed** de todos los manejadores de eventos que se hayan registrado.

```
public class Manejador implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        ...
    }
}
```

Los métodos de **ActionEvent** son:

- **public Object getSource()**
- **public int getModifiers()**

Ejemplo.

```
public class BotonVentana extends JFrame
{
    public BotonVentana ()
    {
        super ("Botón");
        setSize (200, 100);
        setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
        Container cp = getContentPane ();
        cp.setLayout (new FlowLayout ());
        JButton boton = new JButton ("¡Púlsame!");
        boton.addActionListener (new EventoBotonPulsado ());
        cp.add (boton);
    }
}
```



```

public class EventoBotonPulsado implements ActionListener
{
    public void actionPerformed (ActionEvent e)
    {
        System.out.println ("¡Gracias!");
    }
}

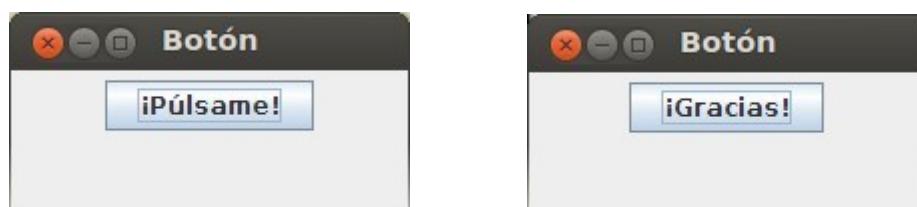
```



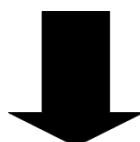
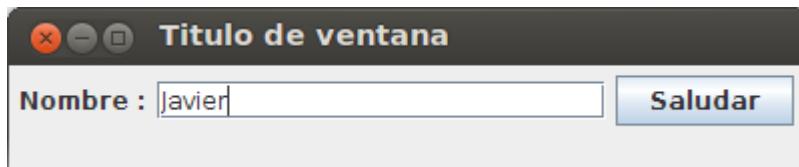
```

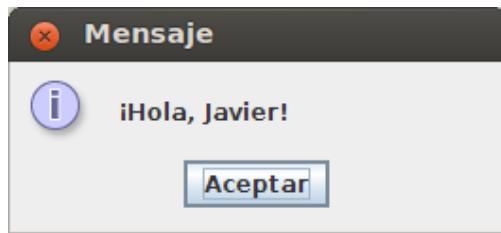
public class EventoBotonPulsado implements ActionListener {
    public void actionPerformed (ActionEvent e)
    {
        JButton boton = (JButton) e.getSource ();
        boton.setText ("¡Gracias!");
    }
}

```



Acceso a componentes de la interfaz





```
public class EventoSaludo implements ActionListener
{
    public void actionPerformed (ActionEvent e)
    {
        JOptionPane.showMessageDialog(null, "¡Hola, " + cuadroTexto.getText() + "!");
    }
}
```

A large black rectangular box containing a white question mark and a white left-pointing arrow.

¿Cómo podemos acceder al objeto **JTextField**?

```
public class EventoSaludo implements ActionListener
{
    private JTextField cuadroTexto;

    public EventoSaludo (JTextField cuadroTexto){
        this.cuadroTexto = cuadroTexto;
    }

    public void actionPerformed (ActionEvent e)
    {
        JOptionPane.showMessageDialog (null, "¡Hola, " +
            cuadroTexto.getText () + "!");
    }
}
```

```

public class MiVentana extends JFrame{

    public MiVentana () {
        super ("Titulo de ventana");
        setSize (400, 300);
        setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
        setLayout (new FlowLayout ());
        Container cp = getContentPane ();
        cp.add (new JLabel ("Nombre :"));
        JTextField texto = new JTextField (20);
        cp.add (texto);
        JButton botonSaludo = new JButton ("Saludar");
        cp.add (botonSaludo);
        botonSaludo.addActionListener (new EventoSaludo (texto));
    }
}

```

Eventos en un JTextField

CaretListener: Cambios en la posición del cursor.

- **void caretUpdate(CaretEvent e)**

DocumentListener: Cambios en el texto.

- **void changedUpdate(DocumentEvent e)**
- **void insertUpdate(DocumentEvent e)**
- **void removeUpdate(DocumentEvent e)**

```

JTextField text = ...;
text.addCaretListener ( ...);
text.getDocument ().addDocumentListener ( ...);

```

Eventos en una ventana

WindowListener

- **void windowActivated(WindowEvent e)**
- **void windowClosed(WindowEvent e)**

- void windowClosing(WindowEvent e)
- void windowDeactivated(WindowEvent e)
- void windowDeiconified(WindowEvent e)
- void windowIconified(WindowEvent e)
- void windowOpened(WindowEvent e)

Referencias

Cosmina I., (2021). Java 17 for Absolute Beginners: Learn the Fundamentals of Java Programming, Apress.

Deitel P., Deitel H., (2019). Java How to Program, Late Objects, Pearson Education, Limited.

Sage K., (2019). Concise Guide to Object-Oriented Programming, Springer.

Schildt H., (2022). Java: A Beginner's Guide, Ninth Edition, McGraw Hill.

Sestoft P., (2016). Java Precisely, Third edition, MIT Press.