

## Apnt JAVA

El nombre del archivo debe de ser igual a la clase.

**Detalle Hola Mundo con Java**

```
public class HolaMundo {  
    public static void main(String[] args) {  
        System.out.println("Hola Mundo con Java");  
    }  
}
```

**public class HolaMundo {** → Define una clase pública llamada HolaMundo  
El nombre del archivo debe llamarse igual pero con extensión .java  
respetando mayúsculas y minúsculas.

**public static void main(String[] args) {** → Define el método o función principal para poder ejecutar un programa en Java. Una función es un bloque de código que nos permite realizar una tarea en particular.

**System.out.println("Hola Mundo con Java");** → La función println permite imprimir el texto "Hola Mundo con Java" en la consola o terminal de la PC.



Atajos:

Sout: Genera método de impresión.

Psvm: Genera función main

Comentarios: Se realizan como en c, con //

## VARIABLES

¿Qué es una Variable?

Una variable es un contenedor que almacena datos que pueden cambiar durante la ejecución de un programa. Cada variable tiene un nombre, un tipo de datos, y un valor.

### Tipos de Variables en Java

#### 1. Variables Primitivas:

- Enteros: byte, short, int, long
- Punto Flotante: float, double
- Carácter: char

- d. Booleano: boolean
- 2. Variables de Referencia: Almacenan referencias a objetos, cualquier clase o tipo definido por el usuario
  - e. Cadenas: String. Ej “Karla”
  - f. Arreglos: int[]
  - g. Objetos de clase: Cualquier instancia de una clase definida por el usuario.

### Declaración y Asignación de Variables

Sintaxis de Declaración: Para declarar una variable en Java, necesitas especificar el tipo de datos seguido del nombre de la variable, siendo la siguiente sintaxis: tipo nombreVariable; el valor se puede asignar al momento de la declaración o posteriormente.

Para imprimir las variables podremos usar el `println` y posteriormente el nombre de la variable, tiene cierta similitud con C.

### FUNCIÓN DE LA RAM EN JAVA

Cuando ejecutas un programa Java, el manejo de variables y la gestión de memoria se realizan principalmente en la RAM. Aquí está una explicación detallada de cómo funciona este proceso:

1. Carga del Programa en Memoria: Cuando ejecutas un programa Java:
  - Compilación: El código fuente Java (.java) se compila en bytecode (.class) por el compilador javac.
  - Ejecución: La JVM (Java Virtual Machine) carga el bytecode en la memoria RAM para su ejecución.

#### 2. Áreas de Memoria en la JVM dentro de la memoria RAM

La JVM gestiona la memoria en diferentes áreas de la RAM, cada una con su propósito específico:

##### 1. Stack:

-Es el área de memoria donde se almacenan las variables locales, así como los valores de variables primitivas definidas dentro de un método.

-El nombre de la variable almacena la dirección de memoria.

-Cada vez que se llama a un método, se crea un nuevo marco (frame) en el stack que contiene las variables locales y los datos necesarios para la ejecución del método.

-Cada vez que sobreescrivimos los valores se modifican los valores en la misma dirección de memoria

## 2. Heap:

-Es el área donde se almacenan los objetos y sus datos asociados, los strings, listas y objetos de clase.

-La memoria en el heap es gestionada automáticamente por el recolector de basura (Garbage Collector).

-El valor que se almacena la variable no es el valor en si mismo sino la referencia al objeto, aquí de forma similar a C al modificar los valores se modifica una dirección de memoria nueva. Pues el nombre almacena la dirección y en la memoria heap se crea dicho objeto string posteriormente si modificamos se crea uno nuevo en la memoria heap y ahora en la memoria stack se modifica la memoria que se estaba almacenando.

Para modificar cadenas se puede hacer como en Python solo reasignando el valor de la cadena, aquí no soltará error.

Nota: Para imprimir con una cadena adicional existen distintos métodos, uno es el concatenación que se realiza mediante el operador +. Su sintaxis sería System.out.println("El tipo de byte es" + variable); aclaración si la variable es de un tipo diferente de string se convertirá en string para su impresión.

## REGLAS Y BUENAS PRÁCTICAS EN NOMBRES DE VARIABLES EN JAVA

Los nombres de variables en Java son esenciales para la legibilidad y mantenibilidad del código. Seguir buenas prácticas y convenciones de nomenclatura ayuda a que el código sea más comprensible y profesional. A continuación, se detallan las reglas y buenas prácticas para nombrar variables en Java, con ejemplos.

### Reglas Básicas para Nombres de Variables

1. Debe comenzar con una letra, un símbolo de dólar (\$) o un guion bajo (\_):
  - a. Correcto: nombre, \_nombre, \$nombre
  - b. Incorrecto: 1nombre, -nombre
2. No puede contener espacios ni caracteres especiales:

- a. Correcto: nombreCliente
  - b. Incorrecto: nombre cliente, nombre-cliente
3. No puede ser una palabra reservada de Java:
    - a. Correcto: int numero
    - b. Incorrecto: int int
  4. Distingue entre mayúsculas y minúsculas:
    - a. nombre, Nombre, NOMBRE son variables diferentes

## Buenas Prácticas

1. Usar Camel Case para Nombres de Variables:
  - a. Utilizar camel case para nombres de variables, comenzando con minúscula y cada nueva palabra con mayúscula.
  - b. Ejemplo: nombreCompleto, numeroDeTelefono
2. Ser Descriptivo y Claro:
  - a. Los nombres de las variables deben describir claramente su propósito.
  - b. Ejemplo: edadPersona, precioProducto
3. Prefijos y Sufijos Claros (si es necesario):
  - a. Usar prefijos como is o has para variables booleanas.
  - b. Ejemplo: isActive, hasSaldo
4. Evitar Nombres de Variables de una Sola Letra:
  - a. A menos que se usen en bucles pequeños o contextos muy específicos.
  - b. Ejemplo: int i en un bucle for, pero preferir indice fuera de ese contexto.
5. No Abusar de Abreviaturas:
  - a. Usar nombres completos siempre que sea posible para mayor claridad.
  - b. Ejemplo: totalPiezas en lugar de totPzs
6. Usar Nombres Significativos Incluso en Variables Temporales:
  - a. Asegurarse de que los nombres de las variables temporales también sean descriptivos.
  - b. Ejemplo: sumaParcial en lugar de te

## TIPO VAR EN JAVA

El tipo var fue introducido en Java 10 como una forma de inferencia de tipos en variables locales. Permite a los desarrolladores declarar variables sin especificar explícitamente su tipo, haciendo que el código sea más conciso y legible.

### Sintaxis Básica de var

La sintaxis para usar var es muy sencilla. Simplemente reemplaza el tipo de la variable con var y deja que el compilador infiera el tipo basándose en el valor inicial asignado. La sintaxis es var variableNombre = valor;

### Limitaciones y Reglas de Uso de var

- 1) Solo para Variables Locales:
  - a. var solo puede ser usado en variables locales dentro de métodos, inicializadores de bucles, y bloques de inicialización.
  - b. No puede ser usado para variables de clase, variables de instancia, o parámetros de métodos.
- 2) Debe ser Inicializada al Declararse:
  - a. La variable debe ser inicializada en la misma línea donde se declara.
- 3) El Tipo Debe ser Inferrible:
  - a. El compilador debe ser capaz de inferir el tipo de la variable a partir del valor asignado. No se le puede asignar null.
- 4) El tipo de dato nunca debe de cambiar.

### Buenas Prácticas al Usar var

-Claridad del Código: Usa var solo cuando la inferencia del tipo sea obvia y clara para otros desarrolladores que lean el código. Ejemplo: var nombre = "Juan"; // Claro: el tipo es obviamente String

## ¿QUÉ ES LA CONCATENACIÓN DE CADENAS?

La concatenación de cadenas es el proceso de unir dos o más cadenas para formar una nueva cadena. En Java, existen varios métodos para concatenar cadenas, cada uno con sus propias características y usos.

### Métodos de Concatenación en Java

1. Operador +
2. Método concat()
3. Clase StringBuilder
4. Clase StringBuffer
5. Método String.join()

## 1. Operador +

El operador + es el método más sencillo y común para concatenar cadenas en Java.

Es fácil de usar y leer. Sintaxis: String resultado = cadena1 + cadena2; Ejemplo

```
String resultado= "Hola"+ " "+ "Mundo"
```

Cuando se concatena una cadena con un valor numérico, el valor numérico se convierte en una cadena y luego se une. Cuando una cadena es el primer operando, el operador + se comporta como un operador de concatenación. Si un número es el primer operando y se concatenan otros números sin paréntesis, se realizará la suma en lugar de la concatenación.

## ¿QUÉ ES UNA CONSTANTE?

Una constante en Java es una variable cuyo valor no puede ser cambiado una vez que ha sido asignado. Las constantes se utilizan para valores que se sabe que no cambiarán durante la ejecución del programa, proporcionando claridad y evitando errores. Sintaxis para Declarar Constantes: Para declarar una constante en Java, se utiliza la palabra clave final. Una vez asignado un valor a una variable final, no se puede cambiar. La sintaxis es la siguiente final tipo nombreConstante = valor;

El nombre de las constantes debe de estar en mayúsculas y generalmente si tiene varias palabras se separa por “\_”. Ademas de que siempre deben de inicializarse los valores.

### Ejemplos de Declaración de Constantes

1. Constante de Tipo Entero: final int DIAS\_EN\_SEMANA = 7;
2. Constante de Tipo Doble: final double PI = 3.14159;

Para usar las constantes se usa como variables, solo se coloca el nombre y listo.

## Beneficios de Usar Constantes

### 1. Claridad del Código:

o Las constantes hacen que el código sea más fácil de leer y entender, ya que se puede ver claramente cuáles valores no deben cambiar.

### 2. Mantenibilidad:

o Facilitan la actualización del código, ya que los valores que pueden cambiar se definen en un solo lugar.

### 3. Prevención de Errores:

o Reducen la posibilidad de errores accidentales al evitar cambios inadvertidos en valores críticos.

## Mejores Prácticas al Usar Constantes

### 1. Nombres de Constantes en Mayúsculas:

- a. Es una convención común en Java escribir los nombres de las constantes en mayúsculas, usando guiones bajos para separar palabras.
- b. Ejemplo: MAX\_VALUE, URL\_BASE, TAMANIO\_BUFFER.

### 2. Uso de final:

- a. Asegúrate de que todas las constantes estén marcadas con la palabra clave final.

### 3. Declaración en una Clase Separada:

- a. Si tienes muchas constantes, considera declararlas en una clase separada para mantener el código organizado.

NOTA: Al imprimirse un string sin inicializar el programa mandara error.

## ¿QUÉ ES UNA CADENA EN JAVA?

Una cadena en Java es una secuencia de caracteres. Las cadenas se utilizan para almacenar y manipular texto. En Java, las cadenas son objetos de la clase String del paquete java.lang

### Sintaxis para Crear Cadenas

Existen varias formas de crear cadenas en Java:

1. Literal de Cadena: String saludo = "Hola, Mundo!"; es la forma primitiva de crearla

2. Usando el Constructor de String: String saludo = new String("Hola, Mundo!"); la palabra new reserva espacio de memoria en heat y strings indica el tipo de objeto a crear. Es la sintaxis formal.

3. Cadenas de múltiples líneas: Se colocan triples comillas para indicar el numero de líneas a generar, siendo que podremos ir colocando enter para cada línea y escribiendo en ellas, siendo que la impresión se realizara así.

## MANEJOS DE INDICES DE CADENAS

Los índices de una cadena estan indexados de manera secuencial comenzando desde 0 hasta la longitud de la cadena menos 1, es decir el primer carácter está en el índice 0 de la cadena y el ultimo carácter en el índice n-1 donde n es el largo de la cadena, es como si fuera un arreglo.

Se puede acceder a dichos elementos del string a apartir de sus índices, para ello se utiliza el metodo charAT que nos permite obtener cada posición del índice. Para utilizarla se usa la siguiente sintaxis: cadena.charAT(indice\_a\_obtener). La obtención la podemos asignar a una variable o incluso solamente podemos colocarla en un println

## INMUTABILIDAD DE CADENAS

Una vez que se crea una cadena, los caracteres dentro de ella no pueden ser modificados. Si se desea modificar una cadena se tiene que crear un nuevo objeto de tipo string y asignarlo a nuestra variable.

## COMPARACION DE CADENAS

Para comparar dos cadenas obviamente se requiere de dos cadenas, se puede utilizar distintos formas.

Operador ==: Compara la dirección de memoria de las dos cadenas, si es el mismo da true en caso contrario da false, si las cadenas son iguales pero su dirección de memoria es diferente dará false.

Si se usa “new String” se reserva un nuevo espacio de memoria por lo que si generamos cadenas con distintos métodos y también con new string pero iguales en contenido, la igualdad del operador == nos dará false.

Método equal: Compara el contenido de las cadenas, siendo que su sintaxis es  
cadena1.equal(cadena2)

## METODOS DE CADENAS

Método lenght: Se utiliza para obtener el largo de una cadena, se utiliza con la siguiente sintaxis “cadena.length()”

Método replace: Se utiliza para reemplazar una subcadena o caracteres, su sintaxis es “cadena.replace(“subcadena a remplazar”, “subcadena nueva”)”. Si reemplazas un carácter dentro de una cadena con "", en la mayoría de los lenguajes, la cadena se acorta y los caracteres siguientes se desplazan automáticamente a la izquierda.

Método toUpperCase(): Se utiliza para convertir toda la cadena en mayúsculas. Su sintaxis es “cadena.toUpperCase()”

Método toLowerCase(): Se utiliza para generar una cadena a partir de una cadena en minúsculas, su sintaxis es “cadena.toLowerCase”

Método trim: Se utiliza para eliminar espacios al inicio y final de la cadena, su sintaxis es “cadena.trim()” o también puede ser el método strip con misma sintaxis.

## SUBCADENAS

Una subcadena es obtener una parte de una cadena original.

Existen varios métodos para trabajar con subcadenas. Métodos Principales para el Manejo de Subcadenas

1. substring(int beginIndex): Devuelve una nueva cadena que es una subcadena de esta cadena. La subcadena comienza en el índice especificado y se extiende hasta el final de la cadena.
2. substring(int beginIndex, int endIndex): Devuelve una nueva cadena que es una subcadena de esta cadena. La subcadena comienza en el índice especificado y se extiende hasta el carácter en el índice endIndex - 1.
3. indexOf(String str): Devuelve el índice de la primera aparición de la subcadena especificada en esta cadena. Si la subcadena no se encuentra, devuelve -1. Busca la primera aparición.

4. `lastIndexOf(String str)`: Devuelve el índice de la última aparición de la subcadena especificada en esta cadena. Si la subcadena no se encuentra, devuelve -1. Busca la ultima aparición, si solo hay una devuelve esa.

#### REEMPLAZAR SUBCADENAS EN JAVA

Para realizarlo se debe usar en conjunto el método `replace` usando su configuración de buscar subcadenas es decir porq su sintaxis es  
`cadena.replace("cadena_original","cadena nueva")`

#### INSERTAR SUBCADENA:

`insert(int offset, String str)`: Inserta el string en la posición especificada, siendo que su sintaxis es la mencionada.

#### ELIMINAR SUBCADENA

`delete(int start, int end)`: Elimina una subcadena de caracteres desde la posición start hasta la posición end (exclusiva)

#### MAS DE CONCATENACION

Como habíamos dicho hay varios métodos de realizar la concatenación de cadenas ademas del operador +.

El primero es el método `concat()`: Al igual que + concatena varias cadenas, el método `concat()` de la clase String concatena la cadena especificada al final de la cadena actual. Sintaxis: `String resultado = cadena1.concat(cadena2);` se puede usar concat con una cadena desde cero usando “”.concat(cadenas...), se pueden pasar mas de dos cadenas a dicho método separando por “,” las cadenas.

El segundo es la Clase `StringBuilder`: `StringBuilder` es más eficiente para concatenaciones repetitivas y en bucles, ya que es mutable y no crea nuevas instancias de cadena. Genera una sola cadena y no varias cadenas. `StringBuilder` en Java es una clase que se utiliza para almacenar secuencias de caracteres que se pueden modificar. Es una alternativa a la clase String, que es inmutable (es decir, no se puede cambiar una vez creada). `StringBuilder` permite hacer modificaciones directamente en la secuencia de caracteres sin crear un nuevo objeto string cada vez, lo cual mejora el rendimiento, especialmente cuando se realizan muchas modificaciones a una cadena. Su sintaxis es

```
var sb = new StringBuilder();
```

```
sb.append(cadena1);
sb.append(cadena2);
var resultado = sb.toString();
```

El método `to.String` es para obtener el resultado final de `stringbuilder`, generándose así la cadena final.

Clase `StringBuffer`: `StringBuffer` es similar a `StringBuilder`, pero es seguro para hilos (`thread-safe`) para varios procesos a la vez, lo que lo hace adecuado para entornos multihilo es decir que se ejecutan varios procesos se están ejecutando al mismo tiempo, su sintaxis es:

```
var sb = new StringBuffer();
sb.append(cadena1);
sb.append(cadena2);
var resultado = sb.toString();
```

Método `String.join()`: `String.join()` es útil cuando se necesita unir una colección de cadenas con un delimitador, es decir que se va añadiendo algo entre ellas. Su sintaxis es `var resultado = String.join(delimitador, cadena1, cadena2);` El delimitador puede ser "" así siendo que se verán solo las cadenas unidas directamente.

## Ventajas y Desventajas

Método	Ventajas	Desventajas
<code>+</code>	Simple y legible	Ineficiente para muchas concatenaciones
<code>concat()</code>	Método específico de <code>String</code>	Similar a <code>+</code> en términos de eficiencia
<code>StringBuilder</code>	Eficiente para muchas concatenaciones, mutable	No es seguro para hilos
<code>StringBuffer</code>	Eficiente y seguro para hilos	Ligeramente más lento que <code>StringBuilder</code> debido a la sincronización
<code>String.join()</code>	Útil para unir múltiples cadenas con un delimitador	Menos flexible para concatenaciones complejas

## CARACTERES ESPECIALES

Actúan de manera distinta cuando las mandamos a imprimir. Practicamente son los caracteres como \n. Algunos son los siguientes

\n salto de línea

\t tabulación

\' comilla simple

\\" comilla doble

Paquete: Archivo que almacena varias clases.

## LEER DATOS

La clase Scanner es parte del paquete java.util y se utiliza para leer la entrada del usuario desde varias fuentes, como el teclado, archivos, cadenas y flujos de entrada.

Para leer datos ingresados por el usuario en Java, puedes utilizar la clase Scanner. Esta clase facilita la captura de datos, ya sean numéricos o de texto.

A continuación, te mostraré cómo hacerlo línea por línea y detallaré los métodos de la clase Scanner.

Paso 1: Importar la clase Scanner

Para utilizar la clase Scanner, primero debes importarla desde la librería java.util. Esto se hace al principio de tu archivo Java:

```
import java.util.Scanner;
```

Esta línea permite que tu programa utilice la clase Scanner.

Paso 2: Crear un objeto Scanner

Luego, debes crear un objeto de la clase Scanner. Este objeto se utiliza para leer la entrada del usuario desde el teclado (System.in):

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

Aquí, scanner es el nombre del objeto que hemos creado. Puedes elegir cualquier nombre válido para tu objeto.

### Paso 3: Leer datos del usuario

Ahora, puedes usar los métodos de la clase Scanner para leer diferentes tipos de datos ingresados por el usuario. Aquí tienes algunos de los métodos más comunes:

nextLine(): Lee una línea completa de texto.

next(): Lee la siguiente palabra (hasta que encuentra un espacio).

nextBoolean()      Lee un valor booleano introducido por el usuario.

nextByte()      Lee un valor byte introducido por el usuario.

nextDouble() Lee un valor double introducido por el usuario.

nextFloat()      Lee un valor float introducido por el usuario.

nextInt()      Lee un valor int introducido por el usuario.

nextLine()      Lee una línea completa como un String.

nextLong()      Lee un valor long introducido por el usuario.

nextShort()      Lee un valor short introducido por el usuario.

Aquí tienes un ejemplo de cómo leer un nombre y una edad del usuario:

```
System.out.println("Ingrese su nombre: ");
```

```
String nombre = scanner.nextLine();
```

```
System.out.println("Ingrese su edad: ");
```

```
int edad = scanner.nextInt();
```

### Buenas Prácticas

Limpiar el buffer: Después de leer un número con nextInt(), nextDouble(), etc., es recomendable limpiar el buffer antes de leer una línea siguiente con nextLine(), pues dichos next no consumen el salto de línea pero el nextline si. Cuando se llama a nextInt, nextDouble, etc., se lee el valor numérico, pero no se

consume el carácter de nueva línea (Enter) que el usuario presiona después de ingresar el valor. Esto deja el carácter de nueva línea en el buffer, y cuando se llama a nextLine, éste lee el carácter de nueva línea restante en lugar de la siguiente línea de entrada. Esto se puede arreglarse agregando una llamada a nextLine() después de leer el número o algún dato.

Otra forma es mediante la conversión de datos utilizando las clases wrapper (clases envolventes) que proporcionan una forma de utilizar tipos de datos primitivos como objetos. Estas clases se encuentran en el paquete java.lang y cada tipo primitivo tiene una clase wrapper correspondiente. A continuación, se muestra cómo utilizar estas clases wrapper para convertir datos ingresados desde la consola.

#### Clases Wrapper Disponibles en Java

1. Integer: Para convertir cadenas en enteros.
2. Float: Para convertir cadenas en números de punto flotante de precisión simple.
3. Double: Para convertir cadenas en números de punto flotante de doble precisión.
4. Boolean: Para convertir cadenas en valores booleanos.
5. Long: Para convertir cadenas en números largos.
6. Short: Para convertir cadenas en números cortos.
7. Byte: Para convertir cadenas en bytes.
8. Character: No tiene un método parse, pero puede ser utilizado para convertir un único carácter

Para usar estas clases se hace lo siguiente.

Se declara una variable string que se inicializara como la variable escaner con el método nextline. Posteriormente se declara una variable ya sea con tipo var o la clase que nosotros queramos y se inicializa usando la siguiente sintaxis “clase.parseTipoDatosPrimitivo(variableQueAlmacenaNextLine())”. Para verlo en ejemplo se puede ver así una conversión de tipo int, ejemplo:

```
String lectura=scanner.nextLine()  
var variable_dato=Integer.Parseint(lectura)
```

Así se puede escribir, pero otra forma mejor elaborada y con menos líneas de código o mejor comprimido es de la siguiente forma

```
var variable_dato=Integer.parseInt(scanner.nextLine())
```

Siendo así más detallado y con menor uso de variables.

Integer.parseInt():

- o Convierte una cadena en un entero.

```
o int entero = Integer.parseInt(consola.nextLine());
```

2. Float.parseFloat():

- o Convierte una cadena en un número de punto flotante de precisión simple.

```
o float flotante = Float.parseFloat(consola.nextLine());
```

3. Double.parseDouble():

- o Convierte una cadena en un número de punto flotante de doble precisión.

```
o double doble = Double.parseDouble(consola.nextLine());
```

4. Boolean.parseBoolean():

- o Convierte una cadena en un valor booleano.

```
o boolean booleano = Boolean.parseBoolean(consola.nextLine());
```

5. Long.parseLong():

- o Convierte una cadena en un número largo.

```
o long largo = Long.parseLong(consola.nextLine());
```

6. Short.parseShort():

- o Convierte una cadena en un número corto.

```
o short corto = Short.parseShort(consola.nextLine());
```

7. Byte.parseByte():

- o Convierte una cadena en un byte.

```
o byte byteValue = Byte.parseByte(consola.nextLine());
```

## 8. Character:

- o Para obtener un carácter, utilizamos el método charAt en la cadena.
- o `char caracter = consola.nextLine().charAt(0);`

**Cerrar el Scanner:** Siempre cierra el Scanner cuando ya no lo necesites para liberar los recursos, usando el método close. Para ello se debe de usar la siguiente sintaxis `variable_con_objeto_scanner.close();`

**NOTA:** Para dar formato a las clases float o double existen diversas formas, para ello se utiliza la siguiente sintaxis “...println(“Valor double %.1f”.formatted(variableDouble))” y otra forma es usando el método “...printf(“Valor double %.2f,variableDouble)”

Para añadir redondeo añadimos un %n a la impresión al lado del f.

**NOTA:** Al usar println se añade un salto de línea después del texto seleccionado a mostrar por lo que si se desea que el cursor quede en la misma línea después del mensaje, usa print() en lugar de println(), ya que print() no agrega un salto de línea automáticamente, es decir la sintaxis sería `System.out.print(cadena);`

### Comparación rápida

Método	¿Salto de línea?	¿Formato?	Uso principal
<code>print()</code>	<input checked="" type="checkbox"/> No	<input checked="" type="checkbox"/> No	Imprimir en la misma línea
<code>println()</code>	<input checked="" type="checkbox"/> Sí	<input checked="" type="checkbox"/> No	Imprimir con salto de línea
<code>printf()</code>	<input checked="" type="checkbox"/> No (pero puedes usar %n)	<input checked="" type="checkbox"/> Sí	Salida formateada

## NUMEROS ALEATORIOS

La clase random es la que nos permite generar números aleatorios y se encuentra en el paquete java.util. Podemos generar valores aleatorios de tipo int, float, double e incluso tipo voleado.

### Introducción a la Clase Random

La clase Random en Java se utiliza para generar números aleatorios. Esta clase proporciona métodos para generar diferentes tipos de números aleatorios, como

enteros, flotantes, y booleanos. Toda generación se inicializa desde 0, pero si se pone +1 nos dará apartir de 1.

Pasos para la generación de valores random:

Importar la Clase Random: Para utilizar la clase Random, primero debemos importarla del paquete java.util: import java.util.Random;

Creación de una Instancia de Random: A continuación, creamos una instancia de la clase Random: Random random = new Random();

Se crea una variable que almacene el valor aleatorio a generar y se usa el método next de la clase Random correspondiente al tipo de dato: var almacen=random.nextTipoDATO(valor\_a\_dar)

1. Generar un Número Entero Aleatorio: Para generar un número entero aleatorio dentro de un rango específico, utilizamos el método nextInt.

```
/* Generar un número aleatorio entre 0 y 9 */  
int numeroAleatorio = random.nextInt(10);      Valor máximo-1  
  
/* Agregar 1 para obtener un rango de 1 a 10 */  
int numeroEntre1y10 = random.nextInt(10) + 1;  (Valor máximo-1) +1
```

2. Generar un Número de Punto Flotante Aleatorio

Para generar un número de punto flotante aleatorio, utilizamos el método nextFloat o nextDouble.

```
/* Generar un número de punto flotante aleatorio entre 0.0 y 1.0 */  
float floatAleatorio = random.nextFloat();  
  
/* Generar un número de punto flotante aleatorio entre 0.0 y 1.0 */  
double doubleAleatorio = random.nextDouble();
```

3. Generar un Valor Booleano Aleatorio

Para generar un valor booleano aleatorio, utilizamos el método nextBoolean.

```
/* Generar un valor booleano aleatorio */  
boolean booleanAleatorio = random.nextBoolean();
```

## Consideraciones adicionales

### Semilla del Generador de Números Aleatorios:

- o La clase Random utiliza una semilla para generar números aleatorios. Si no se proporciona una semilla explícita, se utiliza la hora actual del sistema.
- o Para reproducir la misma secuencia de números aleatorios, se puede proporcionar una semilla fija: Random random = new Random(12345);.

### 2. Uso de ThreadLocalRandom:

- o En aplicaciones concurrentes, se recomienda usar ThreadLocalRandom para mejorar el rendimiento. import java.util.concurrent.ThreadLocalRandom;
- o Ejemplo: int numeroAleatorio = ThreadLocalRandom.current().nextInt(1, 11);.
- o El tema de hilos lo estudiaremos a detalle en el tema de Manejo de Hilos más adelante. Esto es solo para que sepan que se puede utilizar y mejorar la generación de números aleatorios para hacerlo aún más real

## FORMATO DE CADENAS

En esta lección, aprenderemos a dar formato a números y cadenas en Java utilizando tanto cadenas con cadenas simples, así como bloques de texto (text blocks). Veremos cómo usar métodos como String.format y System.out.printf para formatear la salida y cómo incluir saltos de línea y otros caracteres especiales en las cadenas.

Uso de String.format: String.format permite crear una nueva cadena con formato especificado, siendo que podremos trabajar como con C, de la siguiente forma String.format("Cadena %s %d %f", string, variableint, variablefloat). Se podría decir que su sintaxis básica es String.format("Formato", valores); el string format lo podremos asignar a una variable.

SYSTEM. OUT. PRINTF: Puede imprimirse una cadena dándole un formato directamente, para ello se usa el método de System.out.printf. El método

`System.out.printf` en Java funciona de manera similar a `String.format`, pero en lugar de devolver una cadena formateada, imprime directamente la cadena formateada en la consola. Su sintaxis es de forma similar a `string.format` siendo `System.out.printf("Formato", valores);` y se trabaja igual siendo que en los paréntesis va (“Cadena %s %d %f”, string, variableint, variablefloat) y si queremos usar un salto de línea usamos `%n`.

**TEXT BLOCK:** Los bloques de texto (Text Blocks) fueron introducidos en Java 13 como una característica de vista previa y estabilizados en Java 15. Esta característica permite escribir cadenas de texto multilínea de manera más legible y sencilla, sin necesidad de usar concatenaciones ni secuencias de escape como `\n` para saltos de línea. Los bloques de texto se definen con tres comillas dobles seguidas (“”), y puedes incluir texto en múltiples líneas directamente, respetando el formato de la entrada:

```
String texto = """
```

Este es un bloque de texto.

Puedes escribir múltiples líneas aquí.

Java lo manejará de forma eficiente.

```
""";
```

Si queremos añadir valores tenemos que trabajar con el método `formatted(variables contenedoras de datos)` Recordemos la sintaxis para los valores se sigue añadiendo `%tipodato` y para caracteres especiales el `\` y su correspondiente valor. Ejemplo

```
Información del Usuario:  
-----  
Nombre: %s  
Edad: %d  
Salario: %.2f  
-----  
""".formatted(nombre, edad, salario);  
.out.println(mensaje2);
```

Para asegurar que un número entero tenga siempre 4 dígitos (rellenando con ceros a la izquierda si es necesario) y que un número de punto flotante tenga siempre 2 dígitos decimales, utilizamos el siguiente formato: `%04d`: Entero decimal con al menos 4 dígitos, rellenado con ceros a la izquierda si es necesario. `%.2f`: Número de punto flotante con 2 dígitos decimales

## Características y Ventajas

Mejor legibilidad: Los bloques de texto permiten que el contenido de la cadena preserve su formato original, incluyendo saltos de línea y sangrías.

Sin necesidad de concatenación: Ya no necesitas concatenar cadenas o usar secuencias de escape como \n para los saltos de línea.

Manejo automático de espacios: Java automáticamente elimina los espacios extra al principio de las líneas, por lo que la sangría del bloque de texto no afecta el resultado final.

### 3. Caracteres Especiales y Saltos de Línea

Uso de Caracteres Especiales en Formateo Los caracteres especiales como el salto de línea (\n), el tabulador (\t), y otros, se pueden usar para mejorar la legibilidad de la salida en todos los métodos anteriormente dados.

## OPERADORES EN JAVA

En Java, los operadores son símbolos especiales que se utilizan para realizar operaciones

sobre variables y valores. Los operadores más comunes se pueden agrupar en varias categorías:

- Operadores Aritméticos: +, -, \*, /, %      Permiten realizar operaciones matemáticas
- Operadores de Asignación: =, +=, -=, \*=, /=, %=      Nos permiten asignar valores
- Operadores de Comparación: ==, !=, >, <, >=, <=      Nos permiten comparar valores
- Operadores Lógicos: &&, ||, !      Nos permiten combinar expresiones booleanas
- Operadores Unarios: +, -, ++, --, !      Nos permiten realizar operaciones solo un solo operando
- Operadores de Incremento y Decremento: ++, --
- Operador Condicional Ternario: ?:      Simplifica la sentencia if-else

- Operadores de Bits: &, |, ^, ~, <<, >>, >>>

Nota: Se puede declarar varias variables del mismo tipo en una linea, siguiendo la siguiente sintaxis tipo\_dato variable=5,variable=2,variable=3; la inicialización de los valores es opcional pero generalmente hagalo de una vez, aclaración no se puede colocar var.

## OPERADORES ARITMETICOS

Los operadores aritméticos se utilizan para realizar operaciones matemáticas comunes como suma, resta, multiplicación, y división.

- + Suma
- - Resta
- \* Multiplicación
- / División
- % Módulo (resto de la división)

## OPERADORES UNARIOS

Los operadores unarios se aplican a un solo operando.

- |                              |   |
|------------------------------|---|
| • + Operador unario positivo | Indica que el valor es positivo           |
| • - Operador unario negativo | Indica que el valor es negativo           |
| • ++ Incremento              | Realiza un incremento en 1                |
| • -- Decremento              | Realiza un decremento en 1                |
| • ! Negación lógica          | Realiza una negación en valores booleanos |

Preincremento: Se realiza un incremento antes de asignar la variable. Primero se incrementa el valor.

Postincremento: Se realiza un incremento después de asignar la variable. Se incrementa hasta que se vuelve a usar la variable.

## OPERADORES DE ASIGNACIÓN

Los operadores de asignación se utilizan para asignar valores a las variables.

- = Asignación simple
- += Suma y asignación      La sintaxis apartir de aquí es que después del igual se indique el valor con el que se realizara la operación.
- -= Resta y asignación
- \*= Multiplicación y asignación
- /= División y asignación
- %= Módulo y asignación

## OPERADORES DE COMPARACIÓN

Los operadores de comparación se utilizan para comparar dos valores y devuelven un valor booleano (true o false).

- == Igual a
- != No igual a
- > Mayor que
- < Menor que
- >= Mayor o igual que
- <= Menor o igual que

La sintaxis de comparación es variable1 operador variable2

## Operadores lógicos

Los operadores lógicos se utilizan para combinar expresiones booleanas.

- && AND lógico. Regresa verdadero, si ambos operandos son verdaderos. Su sintaxis es “expresión1 && expresión2”. Regresa verdadero solo si ambos valores (operandos) a evaluar son verdaderos.

**|| OR lógico.** Regresa verdadero, si cualquiera de los operandos es verdadero. Su sintaxis es “expresión1 || expresión2”. Regresa verdadero si al menos uno de los valores (operandos) a evaluar son verdaderos.

**! NOT lógico.** Invierte el valor lógico, si es false regresa true, si es true regresa false. Su sintaxis es “!(expresión)”

Estos operandos se pueden combinar con los operandos de comparación para hacer expresiones mas complejas.

## Jerarquía de Operadores en Java

De mayor a menor precedencia:

1. **Postfijos** → `expr++`, `expr--`
2. **Unarios** → `++expr`, `--expr`, `+`, `-`, `!`, `~`
3. **Multiplicativos** → `*`, `/`, `%`
4. **Aditivos** → `+`, `-`
5. **Shift (Desplazamiento de bits)** → `<<`, `>>`, `>>>`
6. **Relacionales** → `<`, `<=`, `>`, `>=`, `instanceof`
7. **Igualdad** → `==`, `!=`
8. **AND bit a bit** → `&`
9. **XOR bit a bit** → `^`
10. **OR bit a bit** → `|`
11. **AND lógico** → `&&`
12. **OR lógico** → `||`
13. **Ternario** → `? :`
14. **Asignación** → `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=`, `>>=`, `>>>=`

## SENTENCIAS DE DECISIÓN

Las sentencias de decisión en Java permiten a los programas tomar diferentes acciones basadas en condiciones específicas. Las sentencias de decisión más comunes en Java son if, if-else, if-else if-else y switch.

### Sentencia if

La sentencia if se utiliza para ejecutar un bloque de código solo si una condición específica es verdadera.

Sintaxis:

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

### Sentencia if-else

La sentencia if-else se utiliza para ejecutar un bloque de código si la condición es verdadera y otro bloque de código si la condición es falsa.

Sintaxis:

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

### Sentencia if-else if-else

La sentencia if-else if-else se utiliza para evaluar múltiples condiciones en secuencia.

Sintaxis:

```
if (condicion1) { // Bloque de código a ejecutar si la condición1 es verdadera
} else if (condicion2) { // Bloque de código a ejecutar si la condición2 es verdadera
} else { // Bloque de código a ejecutar si ninguna condición anterior es verdadera
}
```

## Sentencia switch

La sentencia switch se utiliza para seleccionar una de entre muchas opciones de ejecución. Es especialmente útil cuando se desea comparar una variable con múltiples valores.

Sintaxis:

```
switch (expression_valor_a_evaluar){  
    case valor1:  
        // Bloque de código a ejecutar si expresion == valor1  
        break;  
  
    case valor2:  
        // Bloque de código a ejecutar si expresion == valor2  
        break;  
  
    default:  
        // Bloque de código a ejecutar si ninguno // de los casos anteriores es verdadero  
}
```

## Mejoras en las Versiones Recientes de Java

En Java 12 y versiones posteriores, se introdujeron las expresiones switch, que permiten utilizar una sintaxis más compacta y limpia. También soportan switch con múltiples etiquetas de caso y bloques de texto. La sintaxis es

```
switch (valor_a_evaluar){  
    case valor -> // Bloque de código a ejecutar si expresion == valor;  
    case valor -> // Bloque de código a ejecutar si expresion == valor;  
    case valor -> // Bloque de código a ejecutar si expresion == valor;  
    case valor -> // Bloque de código a ejecutar si expresion == valor;  
    case valor -> // Bloque de código a ejecutar si expresion == valor;  
    default -> // Bloque de código a ejecutar si expresion == valor;  
}
```

```
System.out.println(nombreDia);  
}  
}
```

NOTA: Si queremos que una acción en el switch dependa de varios valores, es decir que una acción para un determinado numero de valores, podemos en el case separarlos por coma. Es decir la sintaxis seria case valor1,valor2,valor3 -> //código; Ejemplo

```
var estacion = switch(mes){  
    case 1, 2, 12 -> "Invierno";  
    case 3, 4, 5 -> "Primavera";
```

Nota: Si en la versión mejorada el bloque de código del case es de mas de una línea abriremos llaves y dentro de ellas colocaremos el bloque de código, pero el punto y coma que se colocaba al final ya no se colocara.

### Detalles Importantes

#### 1. Uso del bloque {}:

o Aunque no es obligatorio usar {} para un solo enunciado después de if, else if, else, o case, es una buena práctica hacerlo para evitar errores y mejorar la legibilidad.

#### 2. break en switch:

o La sentencia break se utiliza para salir del bloque switch después de ejecutar un caso. Si se omite, el flujo de ejecución continuará con el siguiente caso, lo cual puede no ser el comportamiento deseado.

#### 3. Asociatividad y Precedencia:

o Al usar múltiples operadores en una condición, es importante recordar las reglas de asociatividad y precedencia de operadores para asegurar que las condiciones se evalúen correctamente.

## OPERADOR TERNARIO

El operador ternario en Java es una forma concisa de realizar una operación if-else en una sola línea. Este operador es especialmente útil para asignar valores a variables basándose en una condición.

La sintaxis del operador ternario es la siguiente:

```
condicion ? expresion1 : expresion2;
```

- condicion: Es una expresión booleana que se evalúa.
- expresion1: Es el valor que se devuelve si la condición es verdadera.
- expresion2: Es el valor que se devuelve si la condición es falsa.

## PALABRA YIELD

En Java, la palabra clave yield se introdujo en Java 13 como parte de la implementación de la estructura de control switch mejorado con expresiones. Se utiliza dentro de un switch como una forma de devolver un valor cuando se usa switch como una expresión en lugar de una declaración.

Ejemplo de uso de yield

```
public class YieldExample {  
    public static void main(String[] args) {  
        String day = "Lunes";  
        String tipoDia = switch (day) {  
            case "Sábado", "Domingo" -> "Fin de semana";  
            default -> {  
                System.out.println("Día laboral detectado");  
                yield "Día laboral"; // Devuelve el valor al switch  
            };  
        System.out.println("Hoy es un: " + tipoDia);  
    }  
}
```

## Ciclos en Java

Los ciclos (o bucles) en Java se utilizan para ejecutar repetidamente un bloque de código mientras se cumpla una condición específica. Java proporciona varios tipos de ciclos: for, while y do-while.

1. Ciclo while: Evalua la condición primero y si cumple con ello realiza el código.

```
while (condicion) {  
    // Bloque de código a ejecutar  
}
```

- condición: Evalúa si el ciclo debe continuar o no antes de cada iteración

2. Ciclo do-while: Permite ejecutar el código al menos una vez.

Sintaxis

```
do {  
    // Bloque de código a ejecutar  
} while (condicion);  
  
• condición: Evalúa si el ciclo debe continuar o no después de cada iteración. El  
bloque de código se ejecuta al menos una vez.
```

3. Ciclo for: Se utiliza para ejecutar repetidamente un bloque de código un número específico de veces, su sintaxis es:

```
for (inicializacion; condicion; actualizacion) {  
    // Bloque de código a ejecutar  
}  
  
• inicialización: Se ejecuta una vez al principio del ciclo y se utiliza para inicializar la  
variable de control.
```

- condición: Evalúa si el ciclo debe continuar o no.
- actualización: Se ejecuta al final de cada iteración del ciclo y se utiliza para actualizar la variable de control.

Generalmente su sintaxis ya formada es:

```
For(int numero=valor_inicial,i<=10,numero++) {bloque de codigo}
```

### Uso de break y continue en Java

Estas palabras clave se utilizan para controlar el flujo de los ciclos (for, while, do-while).

Break, La palabra clave break se utiliza para salir de un ciclo antes de que haya terminado normalmente. Cuando se encuentra break dentro de un ciclo, el control del programa se transfiere inmediatamente a la línea de código que sigue al ciclo.

Sintaxis ejemplo uso de break:

```
for (inicializacion; condicion; actualizacion) { // código  
if (condicion) { break; }
```

continue, la palabra clave continue se utiliza para saltar la iteración actual del ciclo y continuar con la siguiente. Cuando se encuentra continue dentro de un ciclo, el control del programa se transfiere inmediatamente al comienzo de la próxima iteración del ciclo.

## AREGLOS

Los arreglos (arrays) en Java son estructuras de datos que permiten almacenar múltiples valores del mismo tipo en una sola variable.

1. Declaración de un arreglo, con esto solo se define una variable de tipo arreglo.

```
Tipo_dato[] nombre_arreglo;
```

2. Inicialización de un arreglo:

```
Nombre_arreglo = new dato[5]; // Un arreglo de 5 elementos
```

3. Declaración e inicialización en una sola línea:

```
Tipo_dato[] nombre_arreglo = new tipo_dato[5];
```

4. Cuando se hace esto todas las celdas o posiciones se inicializan con los valores por default de cada tipo. Las posiciones inician desde 0.
5. Modificación de valores: Se realiza accediendo a cada dato por su índice con la sintaxis de nombreArreglo[Posición]=Nuevo\_Valor
6. Para leer los valores de un arreglo se realiza de la misma forma en que se modifican siendo que se coloca la sintaxis de nombre\_Arreglo[Posición]
7. Si se quiere declarar un arreglo y al mismo tiempo inicializar los valores podremos hacer la siguiente sintaxis tipo\_dato[] arreglo= {valores,valores....}, si se quiere usar var se usa var[] arreglo=new tipodato[]{valor,valor,valor....}
8. Para obtener el largo de un arreglo se hace mediante el método length. Su sintaxis sería nombre\_arreglo.length

## RECORRER ARREGLOS

Para recorrer un arreglo, podemos utilizar un bucle for, ya sea en su forma tradicional o each. En Java, existen dos formas principales de recorrer estructuras de datos como arreglos y colecciones (ArrayList, HashSet, etc.): el bucle for tradicional y el bucle for-each. Cada uno tiene sus ventajas y desventajas según la situación en la que se utilicen.

1. Bucle for tradicional: Este bucle utiliza un índice para recorrer una estructura de datos de manera controlada

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

Ventajas del for Tradicional

- ✓ Permite acceder a los índices de los elementos.
- ✓ Se puede recorrer la estructura en cualquier orden.
- ✓ Permite modificar elementos dentro del recorrido.
- ✓ Puede recorrer solo una parte de la estructura si es necesario.

2. Bucle for-each: Este bucle está diseñado específicamente para recorrer arreglos y colecciones de forma más sencilla y segura. No se necesita manejar índices manualmente. Su sintaxis básica es

```
for (TipoDeDatosArreglo variable : estructura_colección) {
```

```
// Código a ejecutar  
} donde variable es donde se almacenara el valor de cada celda en cada iteración.
```

## INTRODUCIR VALORES A UN ARREGLO DE FORMA DINAMICA ENTRE COMILLAS

Se realiza pidiendo al usuario el largo del arreglo y posteriormente en la declaración del arreglo dicho valor se asigna en lugar de colocarlo directamente, y posteriormente para ir añadiendo los valores en cada índice utilizamos un bucle for controlado por el largo del arreglo y en cada iteración ir pidiendo el valor y asignándolo a la posición correspondiente.

## GENERAR UN ARRAY DINAMICO

Un ArrayList en Java es una estructura de datos dinámica de la colección java.util, que permite almacenar elementos de manera similar a un arreglo, pero con la ventaja de que su tamaño puede cambiar dinámicamente conforme se agregan o eliminan elementos. Esto lo hace más flexible que un arreglo convencional, cuyo tamaño es fijo al momento de su declaración.

A diferencia de los arreglos tradicionales, un ArrayList maneja automáticamente la asignación de memoria. Cuando se llena la capacidad actual, se crea un nuevo arreglo con mayor tamaño y se copian los elementos del anterior, permitiendo el crecimiento dinámico.

Para usar un ArrayList, es necesario importarlo desde java.util.ArrayList. Se puede declarar con un tipo de dato específico utilizando generics (<>), asegurando que todos los elementos sean del mismo tipo.

### 1. Importar la clase

Antes de usar ArrayList, es necesario importarlo desde la librería java.util:

```
import java.util.ArrayList;
```

### 2. Declaración de un ArrayList

Se usa la sintaxis de generics (<>) para especificar el tipo de datos:  
`ArrayList<TipoDeDatosConClasesEnvolventes> nombreLista = new ArrayList<>();`

### 3. Métodos básicos de ArrayList

Agregar elementos:

```
nombreLista.add(elemento);
```

Acceder a un elemento por índice:

```
nombreLista.get(índice);
```

Modificar un elemento en un índice específico: nombreLista.set(índice, nuevoValor);

Eliminar un elemento por índice: nombreLista.remove(índice);

Obtener el tamaño del ArrayList: nombreLista.size();

## MATRICES

Una matriz es un arreglo de 2 dimensiones de n columnas y m renglones. Para hacer la declaración de estas se usa la siguiente sintaxis de manera explícita: tipo\_dato[][]  
nombre\_arreglo =new tipo\_dato[filas][columnas]

La segunda forma es con var siendo var nombre\_matriz=new tipo\_dato[fila][columna]

Para recorrerlos y/o modificarlos se usa la sintaxis de nombre\_arreglo[fila][columna].

Para hacer el recorrido total de los arreglos se debe hacer uso de un bucle anidado dentro de otro bucle es decir, de un for, siendo el primero para los renglones y el segundo para las columnas.

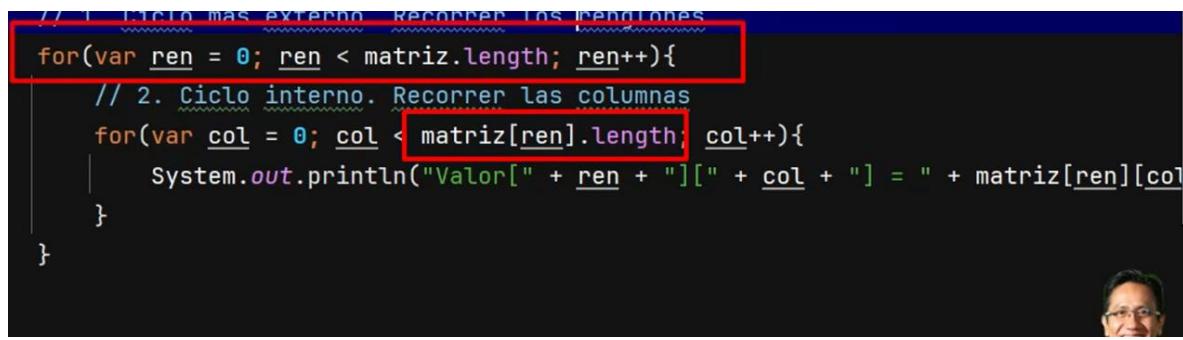
```
//IMPRESION VALORES
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        System.out.print(matriz[i][j] + " ");
        System.out.print(matriz2[i][j] + " ");
    }
}
```

Sintaxis simplificada para una matriz: Se pueden asignar los valores de una matriz desde la declaración siendo que se sigue la siguiente sintaxis:

```
Var matriz= new tipo_dato[][]{
    {valores_fila_n},
    {valores_fila_n+1},
```

```
{valores_fila_n+2}}
```

Si usamos el método length en la variable que almacena nos dará la cantidad de filas que se tienen pero si queremos obtener el numero de columnas ingresamos a una fila y posteriormente usamos el método length. La sintaxis seria matriz[columna\_cualquiera].length. Esto nos podrá ayudar a recorrer también una matriz siendo esta una forma de verlo:



```
// 1. Ciclo mas externo. Recorrer las filas
for(var ren = 0; ren < matriz.length; ren++){
    // 2. Ciclo interno. Recorrer las columnas
    for(var col = 0; col < matriz[ren].length; col++){
        System.out.println("Valor[" + ren + "][" + col + "] = " + matriz[ren][col])
    }
}
```



Para introducir datos se sigue la sintaxis de modificación de igual matrices.

## FUNCIONES

Es un bloque de código reutilizable que realiza una operación en particular. Puede tener parámetros de entrada y también de salida. Las funciones nos sirven para la modularidad y reusabilidad de código.

La sintaxis básica es de :

```
[modificadores] tipoDeRetorno nombreFuncion([tipo parametro1, tipo parametro2...])
{
    // Cuerpo de la función
    [return valor]; // (Opcional si el tipo de retorno no es void)
}
```

Explicación de cada parte:

1. [modificadores] → (Opcional) Controlan la accesibilidad y comportamiento del método.

public → Accesible desde cualquier parte del programa.

private → Accesible solo dentro de la misma clase.

**protected** → Accesible dentro del mismo paquete o por herencia.

**static** → Pertenece a la clase en lugar de a instancias.

**final** → No se puede sobrescribir en clases hijas.

2. **tipoDeRetorno** → Indica el tipo de dato que devuelve la función.

**void** → No devuelve ningún valor.

Tipos primitivos (int, double, boolean, etc.).

Objetos (String, ArrayList, etc.).

3. **nombreFuncion** → Nombre que identifica a la función (debe seguir las reglas de nombres en Java).

4. **(parámetros)** → (Opcional) Lista de valores que la función puede recibir.

Cada parámetro se define con su tipo de dato y nombre.

5. { **cuerpo de la función** } → Contiene el código que ejecuta la función.

6. **return** → (Opcional) Devuelve un valor si el tipo de retorno no es void.

La función se declara antes de la función main.

Para invocar una función en Java, se utiliza su nombre seguido de paréntesis, pasando los argumentos si es necesario. Si el método es estático (static), se puede llamar directamente dentro de la misma clase o usando el nombre de la clase seguido de un punto y el método. Si el método es de instancia, es necesario crear un objeto de la clase con new y luego llamar al método a través del objeto. Cuando el método pertenece a otra clase, se debe hacer referencia a esa clase al invocarlo.

```
public class Calculadora {
    // Método de instancia
    public int sumar(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        Calculadora calc = new Calculadora(); // Crea
        int resultado = calc.sumar(5, 3); // Llamar a
        System.out.println("Suma: " + resultado);
    }
}
```

```
public class Ejemplo {
    // Método estático
    public static void saludo() {
        System.out.println("¡Hola, bienvenido a Java!");
    }

    public static void main(String[] args) {
        saludo(); // Llamada directa dentro de la misma
        Ejemplo.saludo(); // También se puede usar el n
    }
}
```

## Modificadores en Java para funciones (métodos)

Los modificadores en Java permiten definir el nivel de acceso, comportamiento y otras restricciones en los métodos. Se colocan antes del tipo de retorno en la declaración de la función.

### 1. Modificadores de acceso

Controlan **quién** puede llamar al método.

Modificador	Descripción
<code>public</code>	Accesible desde <b>cualquier parte</b> del programa.
<code>private</code>	Solo accesible <b>dentro de la misma clase</b> .
<code>protected</code>	Accesible en la misma clase, en <b>subclases (herencia)</b> y en el <b>mismo paquete</b> .
(Sin modificador - <i>default</i> )	Solo accesible dentro del <b>mismo paquete</b> .

### 2. Modificadores de comportamiento

Definen **cómo** se comporta el método.

Modificador	Descripción
<code>static</code>	Pertenece a la <b>clase</b> y no a los objetos. Se llama sin necesidad de instanciar la clase.
<code>final</code>	No puede ser sobrescrito por subclases (herencia).
<code>abstract</code>	No tiene implementación, debe ser implementado en una subclase.
<code>synchronized</code>	Permite acceso <b>seguro</b> en entornos multihilo.

**FUNCION RECURSIVA:** Es una función que se llama a sí misma, para crearlas tiene reglas: Se llama a sí misma, debe avanzar hacia un caso base de lo contrario se cae en caso infinito ósea poner una condición para que finalice.

DATO:

En Java, todos los parámetros de las funciones se pasan por valor, incluso cuando se trata de objetos. Esto significa que cuando pasamos un tipo primitivo (como int, double, boolean, etc.), el método recibe una copia del valor original y cualquier modificación dentro del método no afecta la variable fuera de él. Sin embargo, cuando pasamos un objeto, lo que realmente se pasa es una copia de la referencia que apunta al objeto en memoria, lo que permite modificar los atributos del objeto.

original dentro del método, aunque no es posible cambiar la referencia para que apunte a un nuevo objeto. Esto puede dar la impresión de que los objetos se pasan por referencia, pero en realidad lo que se está pasando es la referencia por valor. Por ejemplo, si se pasa un objeto Persona con un atributo nombre, dentro del método se puede modificar persona.nombre y el cambio se reflejará fuera del método, ya que la referencia sigue apuntando al mismo objeto en memoria. Sin embargo, si dentro del método se intenta asignar persona a una nueva instancia de Persona, el objeto original no se verá afectado, ya que la referencia copiada dentro del método solo cambia localmente. En contraste, cuando se pasan valores primitivos, cualquier modificación dentro del método solo afecta la copia local y no el valor original fuera del método. En conclusión, Java no permite el paso por referencia real, sino que siempre opera bajo el paso por valor, incluso con objetos, donde se pasa por valor una copia de la referencia, permitiendo modificar los atributos pero sin cambiar la referencia original del objeto.

## CLASES Y OBJETOS

Java es un lenguaje orientado a objetos.

En Java, una clase es una plantilla o modelo a partir del cual se crean objetos. Define atributos (propiedades) y métodos (comportamientos) que los objetos de esa clase pueden tener. Un objeto es una instancia de una clase, es decir, una entidad que posee los atributos y comportamientos definidos en la clase.

Cuando se crea una clase en Java, se definen los atributos como variables y los métodos como funciones dentro de la clase. Para crear un objeto, se usa la palabra clave new, que reserva memoria y llama al constructor de la clase.

**CLASE:** Una clase se compone de atributos y métodos. Los atributos son las características de nuestros objetos.

Los métodos son las acciones que pueden realizar nuestros objetos. En si estas acciones son funciones pero cuando se asocian con una clase se les llama métodos. Una vez que se ha definido una clase, se pueden crear objetos a esto se les llama instanciar una clase.

## 2. Declaración de Clases en Java

En Java, una clase se declara utilizando la palabra clave `class` seguida del nombre de la clase.

#### Sintaxis de una Clase

```
public class NombreClase {  
    // Atributos  
    TipoDatos nombreAtributo;  
    // Constructor  
    public NombreClase(TipoDatos parametro) {  
        this.nombreAtributo = parametro;  
    }  
    // Métodos  
    public void metodo() {  
        // Código del método  
    }  
}
```

- ◆ Los atributos representan el estado del objeto.
- ◆ Los métodos representan el comportamiento del objeto.
- ◆ El constructor es un método especial que se ejecuta al crear un objeto para inicializar sus atributos.

### 3. Creación y Uso de Objetos

Para crear un objeto en Java, se utiliza la palabra clave `new` junto con el constructor de la clase.

#### Sintaxis para Crear un Objeto

```
NombreClase objeto = new NombreClase(valorParametro);
```

- ◆ `NombreClase` es el tipo de dato del objeto.

- ◆ new NombreClase() invoca al constructor de la clase y devuelve una instancia de la clase.

## Acceso a Métodos y Atributos

Para acceder a los atributos o métodos de un objeto, se usa el operador. (punto).

objeto.metodo(); o ; objeto.atributo

NOTA: Si creamos métodos dentro de las clases y queremos usar sus atributos, en los parámetros de la función no se colocan estos, siendo que se pueden mandar o utilizar solo colocando su nombre, pero debemos tener cuidado con que si la función tiene otros parámetros, no colocar el mismo nombre.

## 4. Múltiples Clases en un Archivo Java

En un archivo Java, se pueden definir múltiples clases, pero solo una de ellas puede ser pública (public). La clase pública debe tener el mismo nombre que el archivo.

Ejemplo de Múltiples Clases en un Mismo Archivo (Sin Código Específico)

```
public class ClasePrincipal {  
    public static void main(String[] args) {  
        // Código principal  
    }  
}  
  
class ClaseSecundaria {  
    // Definición de otra clase  
}
```

- ◆ ClasePrincipal es pública y coincide con el nombre del archivo .java.
- ◆ ClaseSecundaria no es pública y solo puede ser usada dentro del mismo paquete.

Si se necesitan múltiples clases públicas, cada una debe estar en su propio archivo con el mismo nombre que la clase.

## ◆ 5. Modificadores de Acceso en Clases y Atributos

Java permite restringir la visibilidad de clases, atributos y métodos mediante **modificadores de acceso**:

Modificador	Clase	Atributo	Método	Descripción
<code>public</code>	✓	✓	✓	Accesible desde cualquier parte del programa
<code>private</code>	✗	✓	✓	Solo accesible dentro de la misma clase
<code>protected</code>	✗	✓	✓	Accesible dentro del mismo paquete y subclases
<i>(sin modificador)</i>	✓	✓	✓	Accesible solo dentro del mismo paquete

Para acceder a atributos `private`, se deben usar **métodos getter y setter**.

## 6. Métodos Especiales en Java

### Métodos get y set (Encapsulamiento)

Los métodos getter y setter permiten acceder y modificar atributos privados de manera controlada.

```
public class NombreClase {  
    private TipoDatos atributo;  
  
    public TipoDatos getAtributo() {  
        return atributo;  
    }  
  
    public void setAtributo(TipoDatos nuevoValor) {  
        this.atributo = nuevoValor;  
    }  
}
```

- ◆ `getAtributo()` devuelve el valor del atributo.
- ◆ `setAtributo(valor)` permite modificar el valor del atributo.

Para acceder y modificar un atributo privado con `get` y `set`, se debe instanciar un objeto de la clase y llamar a estos métodos:

```
public class Principal {  
    public static void main(String[] args) {  
        NombreClase objeto = new NombreClase();  
        // Asignar valor con setter  
        objeto.setNombreAtributo(valor);  
        // Obtener valor con getter  
        System.out.println(objeto.getNombreAtributo());  
    }  
}
```

## 7. Constructores en Java

Un constructor es un método especial que se ejecuta automáticamente cuando se crea un objeto. Se usa para inicializar atributos. Se agrega el constructor en caso de no crearlo en java.

### Características del Constructor

- Tiene el mismo nombre que la clase.
- No tiene tipo de retorno (ni void).
- Se puede sobrecargar con varios constructores.

### Sintaxis de un Constructor

```
public NombreClase(tipoDato parametroInicializacion) {  
    // Código del constructor //INICIALIZACION VALORES  
}  
◆ Se ejecuta automáticamente al crear un objeto con new NombreClase();.
```

## 8. Sobrecarga constructores

La sobrecarga de constructores en Java se refiere a la capacidad de una clase para tener múltiples constructores con el mismo nombre pero diferentes parámetros. Al igual que la sobrecarga de métodos, la sobrecarga de constructores permite crear

objetos de una clase de distintas maneras, proporcionando flexibilidad a la hora de inicializar los objetos.

En Java, todos los constructores deben tener el mismo nombre que la clase. La sobrecarga se logra variando el número y/o tipo de los parámetros. La sobrecarga de constructores es útil cuando deseas crear objetos con diferentes configuraciones o inicializaciones sin necesidad de crear múltiples clases o métodos. Esto ayuda a mejorar la flexibilidad y la legibilidad del código.

Beneficios:  Permite crear objetos de una clase con diferentes configuraciones iniciales.

- Facilita el uso de una clase en distintos contextos.
- Evita la necesidad de crear varios métodos con diferentes nombres para inicializar un objeto.

Una forma de ver esto es con el siguiente ejemplo:

```
public class NombreClase {  
    private int valor;  
    private String texto;  
    // Constructor por defecto  
    public NombreClase(){  
        // Inicializa con valores predeterminados  
        this.valor = 0;  
        this.texto = "Valor por defecto";  
    }  
    // Constructor con un parámetro  
    public NombreClase(int valor){  
        this.valor = valor;  
        this.texto = "Valor personalizado";  
    }  
}
```

```
// Constructor con dos parámetros

public NombreClase(int valor, String texto) {

    this.valor = valor;

    this.texto = texto;

}

}
```

NombreClase(): Es el constructor por defecto, que no toma ningún parámetro y asigna valores predeterminados a los atributos.

NombreClase(int valor): Es otro constructor que recibe un solo parámetro para inicializar el atributo valor.

NombreClase(int valor, String texto): Este constructor permite inicializar ambos atributos con valores personalizados.

¿Cómo Funciona la Sobrecarga de Constructores?

Cuando se crea un objeto, Java selecciona automáticamente el constructor adecuado según los parámetros proporcionados. Si no se pasan parámetros, se utiliza el constructor por defecto. Si se pasan los parámetros adecuados, se invoca el constructor correspondiente.

## 9. Operador THIS

El operador this en Java es una referencia que hace referencia al objeto actual de la clase en la que se utiliza. Se puede utilizar en varios contextos dentro de una clase para referirse a la instancia del objeto y diferenciarlos de otros elementos que puedan tener el mismo nombre. Se crea el momento de la creación de un objeto y se puede utilizar dentro del objeto (y la clase).

¿Cuándo usar el Operador this?

this es útil en los siguientes casos:

Referirse a los atributos de la clase: Ayuda a diferenciar entre los parámetros del método o constructor y los atributos de la clase cuando tienen el mismo nombre.

Llamar a otros constructores dentro de la misma clase: Permite invocar un constructor desde otro constructor utilizando la palabra clave this.

Pasar el objeto actual como parámetro: Puedes pasar el objeto actual de la clase como argumento a un método de otra clase.

## 2. Sintaxis y Uso de this

Referirse a los Atributos de la Clase

Cuando un parámetro o una variable local tiene el mismo nombre que un atributo de la clase, el uso de this se vuelve necesario para diferenciarlos.

```
public class Persona {  
    private String nombre;  
    // Constructor  
    public Persona(String nombre) {  
        // Usamos 'this.nombre' para referirnos al atributo de la clase  
        // y 'nombre' para el parámetro  
        this.nombre = nombre;  
    }  
}
```

En este ejemplo:

this.nombre hace referencia al atributo nombre de la clase.

nombre es el parámetro del constructor.

Sin el uso de this, el compilador no sabría si nos estamos refiriendo al atributo de la clase o al parámetro del método.

## 3. Llamar a Otro Constructor de la Misma Clase

Dentro de un constructor, puedes usar this para llamar a otro constructor de la misma clase. Esto es útil cuando quieres tener una lógica de inicialización común.

```
public class Estudiante {  
    private String nombre;  
    private int edad;
```

```

// Constructor con dos parámetros

public Estudiante(String nombre, int edad) {

    this.nombre = nombre;

    this.edad = edad;

}

// Constructor con un parámetro

public Estudiante(String nombre) {

    // Llamada al constructor con dos parámetros

    this(nombre, 18); // Valor por defecto para edad

}

}

```

En este ejemplo, el constructor con un parámetro usa `this` para llamar al constructor con dos parámetros, asignando un valor predeterminado a la edad.

- ◆ 4. Pasar el Objeto Actual como Parámetro

También puedes usar `this` para pasar el objeto actual de la clase como un argumento a otro método.

```

public class CuentaBancaria {

    private double saldo;

    public CuentaBancaria(double saldo) {

        this.saldo = saldo;

    }

    // Método para realizar una transferencia

    public void transferir(CuentaBancaria cuentaDestino, double monto) {

        cuentaDestino.depositar(monto);

    }

    // Método para depositar dinero

```

```
public void depositar(double monto){  
    this.saldo += monto;  
}  
}
```

Aquí, el método transferir recibe un objeto CuentaBancaria y usa this para referirse al objeto actual y llamarlo dentro del método depositar.

#### 5. Uso de this en Métodos de la Misma Clase

Aunque no siempre es necesario, puedes usar this en un método para referirte al objeto actual, especialmente si quieras hacer que el código sea más explícito.

```
public class Circulo {  
    private double radio;  
  
    public Circulo(double radio) {  
        this.radio = radio;  
    }  
  
    // Método para calcular el área  
  
    public double calcularArea() {  
        return Math.PI * Math.pow(this.radio, 2);  
    }  
}
```

En este caso, this.radio es opcional, pero puede usarse para hacer explícito que se está utilizando el atributo del objeto.

#### ◆ 6. this en Métodos Estáticos

No puedes usar el operador this dentro de métodos estáticos en Java, ya que un método estático no está asociado con una instancia específica de la clase. Los métodos estáticos pertenecen a la clase, no a un objeto específico.

## PAQUETES EN JAVA

Es una colección de archivos y directorios. Nos permiten organizar las clases. Los nombres de los paquetes van en minúscula y si llevan varias palabras se unen con “\_”. Se crea una carpeta siendo que se arrastran las clases creadas a dicho archivo.

En Java, existen principalmente dos tipos de paquetes:

**Paquetes predeterminados (sin nombre explícito):** Si no se especifica un nombre de paquete, la clase pertenece al paquete predeterminado (por defecto).

**Paquetes explícitos (nombrados):** Son los paquetes que se definen con un nombre específico para organizar el código.

### ◆ 2. Sintaxis para Declarar un Paquete

Para declarar que una clase pertenece a un paquete, se utiliza la palabra clave package al inicio de la clase, antes de cualquier otro código.

#### Sintaxis

```
package nombre_del_paquete;  
public class MiClase {  
    // Código de la clase  
}
```

**package:** Palabra clave para declarar el paquete.

**nombre\_del\_paquete:** El nombre del paquete al que pertenece la clase.

## 3. Acceso a Clases y Miembros de un Paquete

Una vez que una clase está organizada dentro de un paquete, puedes acceder a ella desde otras clases siguiendo ciertas reglas de visibilidad.

**Clases del mismo paquete:** Pueden acceder a todas las clases dentro del paquete, sin necesidad de importar.

**Clases de otros paquetes:** Para acceder a clases de otros paquetes, es necesario importar las clases o paquetes usando la palabra clave import.

La sintaxis seria: import paqueteria.clase

Los métodos deben de ser declarados públicos pues sino no podremos acceder a las funciones de las clases si estamos usando dicha clase en otro archivo, otro paquete.

En lugar de importar cada clase individualmente, puedes importar todo un paquete de la siguiente manera:

```
import java.util.*;
```

## ENCAPSULAMIENTO

El encapsulamiento nos permite controlar el acceso a los atributos de nuestra clase. Para evitar acceder a los atributos directamente se usara el modificador de acceso private antes de declarar que tipo de dato son. Para poder leer y modificar los atributos se crearan los metodos conocidos como get y set. Los métodos get nos permiten leer el valor del atributo y el método set nos permite modificar el valor de un atributo.

## HERENCIA

La herencia en Java es un mecanismo que permite que una clase (subclase o clase hija) herede atributos y métodos de otra clase (superclase o clase padre). Esto facilita la reutilización del código y la organización jerárquica de clases, permitiendo que las subclases extiendan la funcionalidad de una clase base sin necesidad de reescribir código.

- ◆ 1. Características de la Herencia

Reutilización de código: Una subclase hereda atributos y métodos de la superclase, evitando duplicación.

Jerarquía de clases: Se establece una relación de tipo "es un" (ejemplo: un Perro es un Animal).

Extensibilidad: Las subclases pueden agregar nuevos métodos o sobreescribir métodos heredados para modificar su comportamiento.

Herencia simple: Java solo permite herencia de una sola clase (no herencia múltiple), pero una clase puede implementar múltiples interfaces.

## 2. Sintaxis de la Herencia

La herencia en Java se implementa con la palabra clave extends.

```
class SuperClasePadre {  
    // Atributos y métodos de la superclase  
}  
  
class SubClase extends SuperClasePadre {  
    // Atributos y métodos adicionales de la subclase  
}
```

Si importas solo la subclase hija a otro archivo, puedes acceder a los métodos protected solo a través de métodos de la subclase.

Si en un mismo código tenemos 3 clases y queremos importarlas esto no puede suceder, por lo que para ello tendremos que reunir las 3 en una sola clase, es decir las 3 clases dentro de una clase principal o también simplemente separarlas por código.

NOTA: Los métodos de la clase padre deben de ser en su mayoría protectec, public causaría problema, pues protectec permite que dichos métodos solo se enfrasquen en la clase padre e hija.

## SOBREESCRITURA

Sobreescritura en Java (@Override)

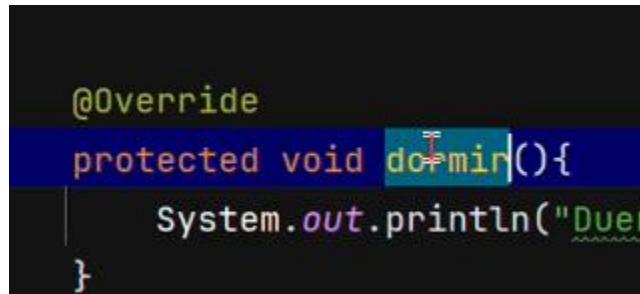
La sobreescritura (o overriding) en Java es un mecanismo que permite a una subclase redefinir un método heredado de su superclase. Esto se usa para cambiar el comportamiento de un método sin modificar la clase original.

- ◆ Reglas de la Sobreescritura

El método en la subclase debe tener el mismo nombre, tipo de retorno y parámetros que el de la superclase. Sí, en la sobreescritura (overriding) de métodos en Java, los parámetros deben ser exactamente los mismos que los del método original en la superclase. Esto incluye tanto el número de parámetros como sus tipos. Si cambias

los parámetros en el método sobrescrito, Java lo interpretará como un nuevo método, no como una sobrescritura.

Debe usar la anotación @Override para indicar que se está sobrescribiendo un método. Una línea arriba



```
@Override
protected void dormir(){
    System.out.println("Duerme");
}
```

A screenshot of a Java code editor showing a snippet of code. The code is annotated with '@Override' above a 'protected void' method named 'dormir'. Inside the method body, there is a single line of code: 'System.out.println("Duerme");'. The code is color-coded, with 'Override' in green, 'protected' in red, 'void' in blue, 'dormir' in orange, 'System.out' in purple, and the print statement in green.

El método en la superclase debe ser public o protected, ya que un método private no se hereda.

No se puede reducir la visibilidad del método sobrescrito (por ejemplo, de public a private).

Puede lanzar las mismas o menos excepciones que el método original.

Solo se modifica el comportamiento de la función.

## SOBREESCRITURA PALABRA SUPER

Sirve para acceder al método de la clase padre que se sobreescrivo en la hija usando la palabra super. En Java, la palabra clave super se utiliza dentro de una subclase para hacer referencia a la superclase. Cuando se utiliza en el contexto de la sobreescritura, se puede emplear de dos maneras principales:

Si una subclase sobrescribe un método de la superclase y quieres invocar el método original de la superclase (antes de que fuera sobrescrito), puedes usar la palabra clave super. Esto es útil cuando necesitas ejecutar la versión original del método, pero también agregar o modificar comportamiento en la subclase. Esto se realiza en el cuerpo de la definición de la clase, no en la instancia de esta.

Si se quiere llamar al método de la clase padre se usa la sintaxis de "super.metodoPadre()"

Además de invocar métodos de la superclase, la palabra clave super también se puede usar para llamar al constructor de la superclase desde la subclase. La sintaxis es "super(parametrosConstructorPadre)"

## POLIMORFISMO

Significa múltiples formas. Se define como diferentes comportamientos dependiendo del tipo de dato con el que estemos transformando. En Java, esto significa que un método o una referencia a un objeto puede comportarse de múltiples maneras dependiendo del contexto en el que se use. Java soporta dos tipos principales de polimorfismo:

Polimorfismo en tiempo de compilación (Sobrecarga de métodos – Overloading): Se refiere a la capacidad de definir múltiples métodos con el mismo nombre dentro de una clase, pero con diferentes parámetros (número o tipo). Características de la sobrecarga de métodos

- ✓ Se define dentro de la misma clase.
- ✓ Se diferencia por la cantidad y/o tipo de parámetros.
- ✓ No depende de los modificadores de acceso ni del tipo de retorno.
- ✓ Es resuelto en tiempo de compilación.

Ejemplo el siguiente: class Calculadora {

```
// Método con dos parámetros enteros
int sumar(int a, int b) {
    return a + b;
}

// Método con tres parámetros enteros
int sumar(int a, int b, int c) {
    return a + b + c;
}

// Método con dos parámetros de tipo double
double sumar(double a, double b) {
    return a + b;
}
```

Polimorfismo en tiempo de ejecución (Sobreescritura de métodos – Overriding):

Ocurre cuando una subclase redefine un método que ya está definido en su superclase con la misma firma (nombre, parámetros y tipo de retorno).

Características de la sobreescritura de métodos

- ✓ Se aplica entre una superclase y una subclase.
- ✓ El método en la subclase debe tener la misma firma que en la superclase.
- ✓ Puede usarse la anotación @Override para indicar que el método está siendo sobreescrito.
- ✓ Es resuelto en tiempo de ejecución gracias al mecanismo de ligadura dinámica.

Un ejemplo de esto es:

```
class Animal {  
    void hacerSonido() {  
        System.out.println("El animal hace un sonido");  
    }  
}  
  
class Perro extends Animal {  
    @Override  
    void hacerSonido() {  
        System.out.println("El perro ladra");  
    }  
}  
  
class Gato extends Animal {  
    @Override  
    void hacerSonido() {  
        System.out.println("El gato maulla");  
    }  
}
```

```

    }

public class Main {
    public static void main(String[] args) {
        Animal miAnimal = new Perro(); // Upcasting
        miAnimal.hacerSonido();      // Salida: "El perro ladra"
        miAnimal = new Gato();
        miAnimal.hacerSonido();      // Salida: "El gato maulla"
    }
}

```

#### 4. Upcasting y Downcasting en polimorfismo

El upcasting y el downcasting son técnicas que permiten manejar objetos en un contexto polimórfico.

##### Upcasting

Cuando se asigna un objeto de una subclase a una referencia de su superclase.

- ✓ Se hace implícitamente.
- ✓ Se pueden llamar solo los métodos definidos en la superclase.

Ejemplo:

```

Animal a = new Perro(); // Upcasting implícito
a.hacerSonido();      // Salida: "El perro ladra"

```

##### Downcasting

Cuando se convierte un objeto de la superclase a un tipo de subclase.

- ✓ Se hace explícitamente con casting.

- ✓ Es potencialmente peligroso si no se verifica el tipo.

Ejemplo:

```
Animal a = new Perro();  
Perro p = (Perro) a; // Downcasting explícito  
p.hacerSonido(); // Salida: "El perro ladra"
```

Si el casting es incorrecto, puede lanzar una excepción ClassCastException.

## METODO POLIMORFICO

Un método polimórfico es un método que puede operar sobre diferentes tipos de objetos que comparten una misma superclase o implementan una interfaz. Gracias al polimorfismo en tiempo de ejecución, cuando se invoca un método en una referencia de la superclase, se ejecutará la versión del método correspondiente al tipo real del objeto. Un método polimórfico es aquel que puede tener diferentes comportamientos dependiendo del objeto que lo invoque. En otras palabras, es un método que es redefinido en clases derivadas para proporcionar una implementación específica manteniendo la misma firma (nombre y parámetros).

- ◆ Principales características:
  - ✓ Se define en una superclase o interfaz y puede ser sobrescrito en las subclases.
  - ✓ La selección del método ocurre en tiempo de ejecución (Dynamic Method Dispatch).
  - ✓ Permite que una única implementación maneje múltiples tipos de
    - a) Polimorfismo con Herencia (Sobreescritura de Métodos)

En este enfoque, una superclase define un método y las subclases lo sobrescriben (@Override). Ya sea que se usen dichos métodos de forma directa o que se puedan trabajar en conjunto para crear un método mas complejo.

B) Creación de métodos polimórficos: El método polimórfico utiliza el tipo de la clase base como parámetro. Esto permite que se le pase cualquier objeto que sea de esa clase o de sus subclases. Aunque el parámetro es de tipo genérico (por ejemplo,

Animal), el método que realmente se ejecuta es el específico del tipo real del objeto (por ejemplo, Perro o Gato).

## CLASE OBJECT

En Java, Object es la clase base de la que heredan implícitamente todas las clases. Es la raíz de la jerarquía de clases y proporciona métodos fundamentales que pueden ser sobrescritos para modificar su comportamiento en clases personalizadas.

### 2. Métodos Principales de Object

La clase Object define varios métodos esenciales que pueden ser utilizados o sobrescritos en las subclases:

a) `toString()`

Devuelve una representación en cadena del objeto. Por defecto, imprime el nombre de la clase seguido del código hash del objeto en hexadecimal.

b) `equals(Object obj)`

Compara si dos objetos son iguales. La implementación predeterminada compara referencias de memoria, pero puede sobrescribirse para comparar contenido.

c) `hashCode()`

Devuelve un número entero que representa un código hash del objeto. Si `equals()` se sobrescribe, es recomendable sobrescribir `hashCode()`.

d) `getClass()`

Devuelve un objeto de tipo `Class<?>` que representa la clase del objeto en tiempo de ejecución.

e) `clone()`

Realiza una copia superficial del objeto. Se debe implementar `Cloneable` y manejar `CloneNotSupportedException`.

## Relación con Otras Clases

### Relación con Otras Clases

La clase Object es la superclase raíz en Java, lo que significa que todas las clases en Java heredan directa o indirectamente de ella, ya sea explícita o implícitamente. Si

una clase no especifica una herencia explícita mediante `extends`, automáticamente extiende `Object`. Esto garantiza que cualquier objeto en Java tiene acceso a los métodos fundamentales definidos en `Object`, como `toString()`, `equals()`, y `hashCode()`.

Las clases envolventes de tipos primitivos (como `Integer`, `Double`, `Boolean`), las estructuras de datos de Java (`ArrayList`, `HashMap`), y las clases definidas por el usuario también descenden de `Object`, lo que les permite interactuar de manera uniforme en colecciones genéricas como `List<Object>`. Además, `Object` juega un rol crucial en la programación genérica, ya que permite almacenar cualquier tipo de objeto en estructuras de datos flexibles. En cuanto a la reflexión, el método `getClass()` de `Object` es esencial para obtener información en tiempo de ejecución sobre cualquier instancia, facilitando operaciones como la introspección de clases y métodos.

En cuanto a la reflexión, el método `getClass()` de `Object` es esencial para obtener información en tiempo de ejecución sobre cualquier instancia, facilitando operaciones como la introspección de clases y métodos. La sobrescritura de métodos de `Object` es una práctica común en el diseño de clases personalizadas y es clave para garantizar su correcto funcionamiento dentro del ecosistema de Java.

## METODO TO STRING

Método `toString()` en Java

### 1. Descripción General

El método `toString()` es un método definido en la clase `Object` que devuelve una representación en cadena del objeto. Su implementación predeterminada devuelve:

`NombreDeLaClase@CódigoHash`

Por ejemplo:

```
class Ejemplo {}  
public class Main {  
    public static void main(String[] args) {  
        Ejemplo obj = new Ejemplo();  
        System.out.println(obj.toString()); // Salida: Ejemplo@1b6d3586 (código hash)  
    }  
}
```

Como se observa, el formato predeterminado no es útil para mostrar información significativa del objeto.

## 2. Sobrescritura de `toString()`

Para proporcionar una representación más significativa del objeto, se puede sobrescribir este método.

Ejemplo: Clase Persona con `toString()` personalizado

```
class Persona {  
    String nombre;  
    int edad;  
  
    Persona(String nombre, int edad) {  
        this.nombre = nombre;  
        this.edad = edad;  
    }  
  
    @Override  
    public String toString() {  
        return "Persona[nombre=" + nombre + ", edad=" + edad + "]";  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Persona p = new Persona("Ana", 25);  
        System.out.println(p.toString()); // Salida: Persona[nombre=Ana, edad=25]  
    }  
}
```

En este caso, `toString()` proporciona información detallada del objeto en lugar de solo la referencia en memoria.

### 3. Uso Automático de `toString()`

Cuando un objeto se imprime con `System.out.println(obj)`, Java llama automáticamente a `toString()`.

Ejemplo:

```
Persona p = new Persona("Luis", 30);
```

```
System.out.println(p); // Automáticamente llama a p.toString()
```

Esto equivale a:

```
System.out.println(p.toString());
```

### 4. Uso en Clases de Java (String, Integer, etc.)

Muchas clases estándar de Java sobrescriben `toString()` para devolver información útil:

```
String texto = "Hola";
```

```
System.out.println(texto.toString()); // Salida: Hola
```

```
Integer num = 100;
```

```
System.out.println(num.toString()); // Salida: 100
```

### 5. Relación con Object y Clases Personalizadas

Todas las clases en Java heredan `toString()` de `Object`, pero generalmente se sobrescribe para mostrar información relevante.

Es útil en depuración y en estructuras de datos (`ArrayList`, `HashMap`), donde los objetos deben representarse en texto.

## CONTEXTO ESTATICO EN JAVA

El contexto estático en Java se refiere a los elementos que pertenecen a la clase en lugar de a una instancia específica. Estos elementos se declaran con la palabra clave `static` y tienen características particulares en cuanto a su acceso y uso.

### 1. Características del Contexto Estático

Pertenecen a la clase, no a los objetos: Se pueden acceder sin necesidad de crear una instancia de la clase.

Se comparten entre todas las instancias: Un único valor es compartido por todos los objetos de la misma clase.

Se inicializan cuando se carga la clase en memoria: No dependen de la creación de objetos.

No pueden acceder directamente a elementos de instancia (this): Dado que static pertenece a la clase y no a una instancia específica, no puede usar this.

## 2. Elementos Estáticos en Java

### a) Variables Estáticas

Las variables estáticas se declaran con static y pertenecen a la clase en lugar de a una instancia. Para acceder a ellas se coloca nombre\_clase.atributoEstatico

Ejemplo:

```
class Ejemplo {  
    static int contador = 0; // Variable compartida por todas las instancias
```

### b) Métodos Estáticos

Los métodos estáticos pertenecen a la clase y se pueden llamar sin necesidad de crear un objeto. Generalmente también sirven para acceder a los atributos estáticos del objeto. Estos métodos no requieren crear el objeto así que solo podemos llamarlo usando la sintaxis clase.metodoEstatico().

Ejemplo:

```
class Utilidades {  
    static int sumar(int a, int b) {  
        return a + b;  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {
```

```
    System.out.println(Utilidades.sumar(5, 3)); // Salida: 8
}
}
```

Reglas de los métodos estáticos:

No pueden usar this ni super.

No pueden acceder a variables o métodos de instancia directamente.

### c) Bloques Estáticos

Los bloques static se ejecutan una vez, cuando la clase se carga en memoria.

Ejemplo:

```
class Ejemplo {
    static int valor;
    // Bloque estático de inicialización
    static {
        valor = 100;
        System.out.println("Bloque estático ejecutado.");
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println(Ejemplo.valor); // Salida: "Bloque estático ejecutado." y "100"
    }
}
```

Explicación:

El bloque estático se ejecuta antes de que se acceda a valor.

Solo se ejecuta una vez, sin importar cuántas instancias se creen.

### **Uso del Contexto Estático en Aplicaciones Reales**

- **Constantes Globales:** Se pueden definir valores constantes con static final.
- **Métodos Utilitarios:** Como los de la clase Math (Math.sqrt(), Math.pow()).
- **Contadores Compartidos:** Para rastrear el número de instancias creadas.
- **Singletons:** Patrón donde una clase solo tiene una única instancia.
- **Se pueden usar:** Se pueden usar sin crear un objeto.