Material descargable

egg

Teoría **JAVA III**

¡Hola nuevamente! 👋

En esta oportunidad, nos adentraremos en un emocionante conjunto de temas de programación que ampliarán nuestro conocimiento y habilidades en Java.

Exploraremos temas de suma importancia como: los **arrays**, los **bucles** (*"for", "while", "do while"* y *"for each"*), la **clase "Arrays"** y la **clase de envoltura "Wrappers".**

Estos temas **nos permitirán ampliar nuestras capacidades de manipulación de datos, mejorar la eficiencia de nuestros programas y adquirir habilidades prácticas** para trabajar con estructuras fundamentales en el desarrollo de aplicaciones.

¡Prepárate para llevar tu dominio de Java al siguiente nivel! Vamos 🚀

Arrays

Un **array es una estructura de datos que nos permite almacenar una colección de elementos, ya sean valores o variables**. Cada elemento en el array se identifica mediante un índice o clave.

A diferencia de las estructuras de datos primitivas**, los arrays en Java son objetos** y se instancian a partir de una clase predefinida en el lenguaje.

A continuación, veamos cómo se declaran los arrays:

```java
public static void main(String[] args) {
    int numero = 6;
    int[] array1 = {1,numero,2,3}; //array1 es un objeto de la clase int[]
    int[] array2 = new int[4]; //array2 es un objeto de la clase int[]
}
```

En la declaración de un array, debemos especificar el tipo de dato de los elementos que contendrá, seguido de unos corchetes "[]" para indicar que se trata de un array. Luego, asignamos un nombre a la variable y, opcionalmente, le asignamos un valor.

En el ejemplo mostrado, utilizamos dos tipos de asignaciones:

- Al utilizar **llaves**, estamos inicializando el array con los valores o variables que deseamos almacenar, separados por comas.
- Al utilizar el **operador "new"**, debemos especificar el nombre de la clase (en este caso *"int[]"*) y el tamaño del array, ya que los arrays tienen una longitud fija. Tanto *"array1"* como *"array2"* tienen un tamaño de 4 elementos. En el caso de *"array1"*, el tamaño se determina automáticamente según la cantidad de elementos especificados entre las llaves.

Por otra parte, es posible acceder y modificar los elementos de un array utilizando su índice. En Java, los índices de los arrays comienzan desde 0 y van hasta el tamaño del array menos 1. Por ejemplo:

```
public static void main(String[] args) {
    String[] arr = new String[3];
    arr[0] = "hola"; //Modifica el primer elemento del arreglo
    System.out.println(arr[0]); //Accede al primer elemento del arreglo
}
```

Los arrays tienen una propiedad llamada **"length"** que devuelve su tamaño. Sin embargo, *esta propiedad es de solo lectura* y no puede ser modificada, ya que es un atributo final. Por ejemplo:

```
public static void main(String[] args) {
    int[] array3 = new int[6];
    System.out.println(array3.length); //Imprime: 6
    array3.length = 3; //Marca un error: The final field array.length cannot
be assigned
}
```

💡*Recuerda que todos los elementos en un array de Java deben ser del mismo tipo. Por ejemplo, puedes tener un array de enteros o un array de strings, pero no puedes mezclar diferentes tipos en el mismo array:*

```
public static void main(String[] args) {
    int[] intArr = {1,2,3}; //Correcto
    String[] str Arr = {"hola","como","estas?"}; //Correcto
    int[] mixEr ={"hola",2,3}; //Incorrecto
}
```

Bucles

En programación, *un **bucle** es una estructura de control que permite repetir un bloque de código varias veces*.

💡*Los bucles son esenciales, ya que nos permiten realizar tareas repetitivas de manera eficiente.*

En esta ocasión, profundizaremos en los siguientes bucles:

- **For i**
- **While**
- **Do While**
- **For each**

For i

El bucle "for" se utiliza cuando conocemos la cantidad de veces que se debe repetir un bloque de código. Tiene tres partes:

- **Inicialización** → Aquí es *donde se inicializa la variable de control* del bucle. *En la mayoría de los casos, esta variable se denomina "i"*.
  El uso de "i" es simplemente una convención; se utiliza tradicionalmente para indicar *"índice"*, pero en realidad puedes usar cualquier nombre de variable válido en Java.

  ```
  for (int i = 0; ... ) {
      //Código del bucle
  }
  ```

  En este ejemplo, *"int i = 0;"* inicializa la variable *"i"* con el valor 0.

- **Condición** → Esta es la *condición que se verifica antes de cada iteración del bucle*. Si la condición es verdadera (“true”), se ejecuta el bloque de código dentro del bucle. Si es falsa (“false”), el bucle se detiene.

```
for (... ; i < 5; ... ) {
    //Código del bucle
}
```

  En este ejemplo, *"i < 5"* es la condición. Por lo tanto, mientras el valor de *"i"* sea menor que 5, el bloque de código del bucle continuará ejecutándose.

- **Actualización** → En esta parte, *se actualiza la variable de control*. En la mayoría de los casos, simplemente se *incrementa* o *decrementa* la variable.

```
for (... ; ... ; i++) {
    //Código del bucle
}
```

  Aquí, *"i++"* incrementa el valor de *"i"* en 1 en cada iteración del bucle.

A continuación, te mostramos el *bucle "for"* completo a modo de referencia:

```
public static void main(String[] args) {
    for (int i = 0; i < 5; i++) {
        System.out.println("El valor de i es: " + i);
    }
}
```

Este bucle imprimirá los números del 0 al 4 en la consola. Después de cada iteración, "i" se incrementa en uno (i++), y mientras "i" sea menor que 5 (i < 5), el bucle continuará. Cuando "i" llegue a 5, la condición "i < 5" será falsa y el bucle se detendrá.

For i & arrays

El **bucle "for" con "i"** es muy útil para acceder a los elementos de un array a través de sus *índices*.

Observemos estos dos ejemplos de cómo mostrar los elementos de un arreglo sin y con un bucle:

- *Sin* un bucle “for”:

```
public static void main(String[] args) {
    String[] paises = {"Uruguay","Argentina","Brasil","Venezuela"};
    System.out.println(paises[0]);
    System.out.println(paises[1]);
    System.out.println(paises[2]);
    System.out.println(paises[3]);
}
```

- *Con* un bucle “for”:

```
public static void main(String[] args) {
    String[] paises = {"Uruguay","Argentina","Brasil","Venezuela"};
    for (int i = 0; i < paises.length ; i++) {
        System.out.println(paises[i]);
    }
}
```

While

El bucle "while" se utiliza cuando se desea repetir un bloque de código siempre que una condición sea verdadera.

Veamos el siguiente ejemplo:

```
public static void main(String[] args) {
    int numeroAleatorio = 0;
    while (numeroAleatorio < 8) {
        numeroAleatorio = (int) (Math.random()*(10-0+1)+0);
    }
    System.out.println("Seguro es 8 o mayor a 8: "+numeroAleatorio);
}
```

Aquí, el bloque de código se ejecuta mientras *"numeroAleatorio"* sea menor que 8, y en cada iteración del bucle, el valor de *"numeroAleatorio"* cambia.

Do While

El bucle "do-while" es similar al "while", pero la condición se evalúa después de la ejecución del bloque de código, lo que garantiza que el bloque de código se ejecute al menos una vez.

Observemos el siguiente código:

```
public static void main(String[] args) {
    Scanner pepe = new Scanner(System.in);
    int num;
    do {
        System.out.print("Por favor, ingrese un número mayor a 0: ");
        num = pepe.nextInt();
    } while (num <= 0);
    System.out.println("Ingresaste: " + num);
}
```

En este ejemplo, el programa solicita al usuario que ingrese un número, y si el número es menor o igual a 0, el programa seguirá solicitando al usuario que ingrese otro número. Esto continuará hasta que el usuario ingrese un número mayor que 0.

Si intentamos lograr lo mismo con un *bucle "while"*, tendríamos que repetir el código para leer el número del usuario:

```
public static void main(String[] args) {
    Scanner pepe = new Scanner(System.in);
    System.out.print("Por favor, ingrese un número mayor a 0: ");
    int num = pepe.nextInt();
    while (num <= 0) {
        System.out.print("Por favor, ingrese un número mayor a 0: ");
        num = pepe.nextInt();
    }
    System.out.println("Ingresaste: " + num);
}
```

Esto se debe a que el *bucle "while"* evalúa la condición antes de entrar al bloque de código, por lo que necesitamos solicitar al usuario que ingrese un número antes del bucle para inicializar "num". Luego, dentro del bucle, necesitamos volver a solicitar al usuario que ingrese otro número si el número anterior no era válido.

💡 ***Con el bucle "do-while" podemos evitar la duplicación de código**, ya que la condición se evalúa después de la ejecución del bloque de código.*

For each

El bucle "for-each" se utiliza para recorrer elementos en arreglos o colecciones sin tener que lidiar con índices. (Las colecciones las veremos más adelante).

```
public static void main(String[] args) {
    int[] arr = {1, 2, 3, 4, 5};
    for (int num : arr) {
        System.out.println("El valor es: " + num);
    }
}
```

En este ejemplo, la *variable "num"* toma cada valor en el *arreglo "arr"* en cada iteración del bucle, lo que simplifica el proceso de iterar a través del arreglo sin necesidad de utilizar un índice.

📌 Resumen

Hemos visto los 4 tipos de bucles que existen en Java. Repasemos en qué casos nos son útiles cada uno:

- **For i:** Se utiliza cuando conocemos o podemos calcular de alguna manera la cantidad de repeticiones que necesitamos realizar.
- **While & Do While:** Se utilizan cuando la cantidad de repeticiones que necesitamos realizar es un valor desconocido y no es fijo.
    - **While:** Se utiliza cuando queremos que el código se ejecute sólo cuando se cumple una condición.
    - **Do While:** Se utiliza cuando queremos que el código se ejecute al menos una vez antes de comprobar una condición para repetirlo nuevamente.
- **For Each:** Se utiliza cuando deseamos iterar un arreglo y no nos interesa utilizar el índice.

Es importante comprender las diferencias y los casos de uso de cada tipo de bucle para seleccionar el más adecuado según la situación.

Clase Arrays

La **clase "Arrays"** en Java es una utilidad que **proporciona varios métodos estáticos para manipular, copiar, ordenar, buscar y comparar arreglos**. Para utilizar esta clase, es necesario importarla del *paquete java.util*.

Veamos un ejemplo:

```
import java.util.Arrays;

public class ArraysEnJava {
    public static void main(String[] args) {
        int[] original = {1,2,3};
        System.out.println(original .length); //Imprime 3
        original = Arrays.copyOf(original , 10);
        System.out.println(original .length); //Imprime 10
    }
}
```

En este código, hemos importado la *clase "Arrays"* para poder utilizarla en nuestro programa. El *método "copyOf()"* nos permite crear un nuevo arreglo con los valores del arreglo original pero con un nuevo tamaño. En este caso, el nuevo arreglo creado se asigna nuevamente a la *variable "original"*.

Algunos de los métodos más comunes de la *clase "Arrays"* son:

- **Arrays.sort():** Se utiliza para ordenar un arreglo en orden ascendente. Puede ser usado con tipos primitivos y objetos que implementen la *interfaz Comparable*.

  ```
  int[] arr = {1, 5, 2, 6, 3, 7};
  Arrays.sort(arr);
  //El arreglo arr ahora está ordenado de forma ascendente
  ```

  ❗ *Más adelante hablaremos de lo que es una interfaz. Por ahora, solo tengamos en cuenta que tanto "String" como "Wrappers" utilizan esta interfaz.*

- **Arrays.binarySearch():** Realiza una búsqueda binaria de un valor específico en un arreglo ordenado. Devuelve el índice del valor buscado en el arreglo si está presente; de lo contrario, devuelve un valor negativo.

```
int[] arr = {1, 2, 3, 4, 5};
int index = Arrays.binarySearch(arr, 3);
//Devuelve 2, que es el índice de 3 en el arreglo
```

💡*La búsqueda binaria es un algoritmo de búsqueda eficiente que se utiliza para buscar elementos específicos en colecciones ordenadas. Para utilizarla, es necesario que los objetos implementen la interfaz Comparable. Esto permite comparar y ordenar los elementos de manera adecuada, asegurando un funcionamiento correcto del algoritmo.*

- **Arrays.equals():** Compara dos arreglos para determinar si son iguales, es decir, si tienen la misma longitud y los mismos elementos en la misma posición.

```
int[] arr1 = {1, 2, 3};
int[] arr2 = {1, 2, 3};
boolean isEqual = Arrays.equals(arr1, arr2);
//Devuelve true si los arreglos son iguales
```

- **Arrays.fill():** Se utiliza para llenar todos los elementos de un arreglo con un valor específico.

```
int[] arr = new int[5];
Arrays.fill(arr, 1);
//Todos los elementos del arreglo son ahora 1
```

- **Arrays.copyOf() y Arrays.copyOfRange():** Estos métodos se utilizan para copiar un arreglo o una parte de él en un nuevo arreglo.

```
int[] original = {1, 2, 3, 4, 5};
int[] copia = Arrays.copyOf(original, original.length);
//Crea una copia del arreglo original
int[] parteDeUnaCopia= Arrays.copyOfRange(original, 1, 3);
//Crea una copia de una parte del arreglo original (índices 1 a 2 - El tercer parámetro no es inclusivo)
```

- **Arrays.toString():** Convierte un arreglo en una cadena legible, lo cual es útil para la depuración.

```
int[] arr = {1, 2, 3};
System.out.println(Arrays.toString(arr));
//Imprime "[1, 2, 3]"
```

Estos métodos son solo algunos ejemplos de las funcionalidades que ofrece la clase "Arrays" en Java para trabajar con arreglos.

Clases de Envoltura “Wrappers”

En Java, **cada tipo de dato primitivo tiene una clase envolvente o "wrapper" que encapsula el tipo de dato primitivo en un objeto**.

Las clases wrapper son útiles porque permiten tratar a los tipos primitivos como objetos, lo que facilita el uso de métodos y la representación de valores nulos, algo que los tipos primitivos no pueden hacer. Además, son necesarias cuando se trabaja con colecciones de objetos que no pueden almacenar tipos de datos primitivos (lo veremos más adelante).

A continuación, te mostramos las clases “wrapper” correspondientes a cada tipo de dato primitivo:

- **boolean** → Boolean
- **char** → Character
- **int** → Integer
- **double** → Double
- **byte** → Byte
- **short** → Short
- **long** → Long
- **float** → Float

Y ahora, repasemos los métodos más importantes que proporcionan estas clases:

Boolean

Métodos de instancia

- **variable.toString():** Devuelve un objeto String que representa el valor de este booleano.

Métodos de clase

- **Boolean.valueOf(String s):** Devuelve un objeto booleano dependiendo del string ingresado por parámetro. Si s es igual a “true” (ignorando mayúsculas y minúsculas) devuelve verdadero, sino devuelve falso.

Character

💡*Ten presente que los caracteres tienen una representación numérica en la tabla ASCII.*

Métodos de instancia

- **variable.toString():** Devuelve un objeto String que representa el valor de esta variable de tipo Character.
- **variable.compareTo(Character anotherCharacter):** Compara numéricamente con anotherCharacter, devuelve 0 si son iguales, -1 si "variable" es menor numéricamente, y 1 si "variable" es mayor numéricamente.

Métodos de clase

- **Character.valueOf(String s):** Devuelve un objeto booleano dependiendo del string ingresado por parámetro. Si s es igual a "true" (ignorando mayúsculas y minúsculas) devuelve verdadero, sino devuelve falso.
- **Character.getNumericValue(char ch):** Devuelve el valor int que representa el carácter Unicode especificado por parámetro.
- **Character.getType(char ch):** Devuelve un valor que indica la categoría general de ch. Puedes ver las categorías 👉**aquí**.
- **Character.getNumericValue(char ch):** Devuelve el valor int que representa el carácter Unicode especificado por parámetro.
- **Character.isLetter(char ch):** Determina si el carácter especificado es una letra.
- **Character.isWhitespace(char ch):** Determina si el carácter especificado es un espacio en blanco.

Integer

Métodos de instancia

- **variable.toString():** Devuelve un objeto String que representa el valor de este Integer.

- **variable.compareTo(Integer anotherInteger):** Compara numéricamente con anotherInteger, devuelve 0 si son iguales, -1 si "variable" es menor numéricamente, y 1 si "variable" es mayor numéricamente.

Métodos de clase

- **Integer.valueOf(String s):** Devuelve un objeto Integer dependiendo del string ingresado por parámetro.
- **Integer.MAX_VALUE:** Una constante que contiene el valor máximo que puede tener un int (2^31)-1.
- **Integer.MIN_VALUE:** Una constante que contiene el valor mínimo que puede tener un int -2^31.

Double

Métodos de instancia

- **variable.toString():** Devuelve un objeto String que representa el valor de este Double.
- **variable.compareTo(Double anotherDouble):** Compara numéricamente con anotherDouble, devuelve 0 si son iguales, -1 si "variable" es menor numéricamente, y 1 si "variable" es mayor numéricamente.

Métodos de clase

- **Double.valueOf(String s):** Devuelve un objeto Double dependiendo del string ingresado por parámetro.
- **Double.MAX_VALUE:** Una constante que contiene el mayor valor finito positivo de tipo double, (2-2^(-52))*2^1023.
- **Double.MIN_VALUE:** Constante que contiene el menor valor positivo distinto de cero de tipo double, 2^-1074.
