

Contenido

1. Bloques fundamentales.....	3
1.1. Métodos frente a Funciones y Procedimientos	3
1.1.1. Conceptos básicos	3
1.1.2. Paso de información a procedimientos y funciones (paso de parámetros)	4
1.1.3. Valor devuelto por una función	5
1.1.4. Sobrecarga de funciones.....	5
1.1.5. Recursividad	6
1.1.6. Métodos	6
1.2. Ámbito y tiempo de vida de las variables	10
1.3. Clase principal y método main().....	13
2. Identificadores	16
3. Palabras reservadas.....	16
4. Variables.....	17
5. Tipos de datos	17
5.1. Tipos primitivos	17
5.1.1. byte.....	17
5.1.2. short	17
5.1.3. int	17
5.1.4. long.....	18
5.1.5. float	18
5.1.6. double	18
5.1.7. char.....	18
5.1.8. boolean.....	18
5.2. Tipos de referencia.....	19
5.2.1. String	19
5.2.2. Date	19
6. Literales	19
6.1. Literales enteros.....	20
6.2. Literales de punto flotante.....	20
6.3. Literales de caracteres	20
6.4. Literales de cadena de caracteres (String)	20
6.5. Literales booleanos	20

6.6. Literales de valores nulos (null).....	20
7. Constantes.....	21
7.1. Notación SCREAMING_SNAKE_CASE	21
8. Operadores y expresiones.....	22
8.1. Operadores binarios.....	22
8.1.1. Operadores aritméticos	22
8.1.2. Operadores relacionales	22
8.1.3. Operadores lógicos.....	22
8.1.4. Operadores especiales	23
8.2. Operadores unarios en Java.....	23
8.3. Operadores de bit	23
8.4. Tablas resumen con los significados de cada operador.....	24
8.4.1. Tabla de operadores binarios en Java.....	24
8.4.2. Tabla de operadores unarios en Java.....	24
8.4.3. Tabla de operadores de bit en Java.....	25
8.5. Precedencia de operadores	26
9. Conversiones de tipo.....	27
9.1. Relación de conversiones implícitas.....	27
9.2. Relación de conversiones explícitas.....	28
10. Comentarios	28
10.1. Comentario de una sola línea	28
10.2. Comentario de múltiples líneas	28
10.3. Comentario de documentación (Javadoc)	29
10.3.1. Opciones de Javadoc más utilizadas	30
11. Entrada/Salida estándar.....	31
11.1. Entrada desde teclado.....	31
11.1.1. Métodos usados para la entrada desde teclado	32
11.2. Salida a pantalla	33
11.2.1. Caracteres de formato para la salida a pantalla	34
12. Despedida.....	34

1. Bloques fundamentales

1.1. Métodos frente a Funciones y Procedimientos

1.1.1. Conceptos básicos

Hasta ahora hemos visto los denominados **bloques anónimos** que constan de una o más instrucciones encerradas entre llaves. Hemos visto este tipo de bloques acompañando a las sentencias if, if-else, switch, while, do-while, y for.

Existen otros tipos de bloques a los que sí se les da un nombre. En primer lugar vamos a hablar de este tipo de bloques en el ámbito de la programación estructurada y luego veremos cómo se usan en el paradigma de orientación a objetos.

En la programación estructurada existen dos tipos de **bloques con nombre**: los procedimientos y las funciones. Vamos a verlos a continuación.

- Un **procedimiento** es un conjunto de instrucciones o pasos que se ejecutan secuencialmente para llevar a cabo una tarea específica o realizar una serie de acciones. Los procedimientos son una forma de organizar y reutilizar código al encapsular un conjunto de acciones en una unidad coherente y llamable desde otras partes del programa.

EJEMPLO: Supongamos que queremos crear un procedimiento en Java que imprime un mensaje en pantalla:

```
// Definición de un procedimiento que recibe un mensaje como parámetro
public static void imprimirMensaje(String mensaje) {
    System.out.println(mensaje);
}
```

¿Cómo podríamos imprimir el mensaje “Hola, mundo” usando el procedimiento?

- Una **función** es un bloque de código reutilizable que, al igual que un procedimiento, realiza una tarea específica, pero además puede devolver un valor.

EJEMPLO: Aquí hay una función simple en Java que suma dos números y devuelve el resultado:

```
public int suma(int a, int b) {
    int resultado = a + b;
    return resultado;
}
```

¿Cómo calcularíamos el valor de la suma de 12 y -3 usando la función?

1.1.2. Paso de información a procedimientos y funciones (paso de parámetros)

La información (variables u objetos) que se le proporcionan a un procedimiento o una función para que realicen su tarea se llaman **parámetros**.

En Java, los parámetros de procedimientos y funciones se pasan por valor para tipos primitivos (tipos simples) y por referencia para tipos de referencia (objetos). Aquí explicaré estas dos opciones y proporcionaré ejemplos de cada una de ellas:

1. Paso de parámetros por valor:

- En este enfoque, se pasa una copia del valor de la variable al procedimiento o función. Cualquier modificación realizada dentro del procedimiento no afecta a la variable original fuera de él.

EJEMPLO: ¿Qué información se visualizará en la consola?

```
public static void incrementar(int n) {  
    n++; // Modifica la copia local de 'n'  
    System.out.println("Número dentro del método: " + n);  
}  
int numero = 5;  
incrementar(numero);  
System.out.println("Número fuera del método: " + numero);
```

2. Paso de parámetros por referencia:

- En este enfoque, se pasa una referencia al objeto o estructura de datos. Cualquier modificación realizada dentro del procedimiento afecta directamente al objeto original al que se hace referencia.

EJEMPLO: ¿Qué información se visualizará en la consola?

```
public static void modificarTexto(StringBuilder str) {  
    str.append(", mundo"); // Modifica el objeto original 'str'  
    System.out.println("Texto dentro del método: " + str);  
}  
StringBuilder texto = new StringBuilder("Hola");  
modificarTexto(texto);  
System.out.println("Texto fuera del método: " + texto);
```

1.1.3. Valor devuelto por una función

Una función puede devolver cualquier tipo de dato, incluidos objetos instanciados a partir de las clases que defina el programador. Para ello se usa la sentencia **return**.

Tras haber visto el paso de parámetros por valor quizás te preguntas qué pasa con un objeto que sea creado dentro de una función después de que ésta acabe su ejecución. Pues bien, dicho objeto seguirá existiendo tras la finalización de la función siempre que haya alguna variable que lo siga referenciando. De no ser así la memoria que ocupa el objeto será liberada en la próxima ejecución del **recolector de basura**.

1.1.4. Sobrecarga de funciones

En Java pueden coexistir dos o más funciones con el mismo nombre puesto que el compilador es capaz de diferenciarlas en función del número, tipo y orden de sus parámetros. Es a esta propiedad del lenguaje a lo que se conoce como **sobrecarga**.

EJEMPLO: Por cierto, ¿Ves cómo el operador suma (+) también está sobrecargado?

```
public static int suma(int a, int b) {  
    return a + b;  
}  
  
public static String suma(String a, String b) {  
    return a + b;  
}  
  
int resultadoEntero = suma(1, 2);  
String resultadoCadena = suma("Hola, ", "tronco");  
  
System.out.println("Resultado de la suma de enteros: " + resultadoEntero);  
System.out.println("Resultado de la suma de cadenas: " + resultadoCadena);
```

1.1.5. Recursividad

Java soporta la recursión. La **recursión** es el proceso mediante el que algo se define en términos de él mismo. Dicho en términos de Java, *la recursión es la capacidad que permite a un procedimiento o una función el hecho de llamarse a sí mismo*. Un procedimiento o una función que se llaman a sí mismo se denomina **recursivo**.

Como curiosidad, cabe mencionar que algunos tipos de algoritmos relacionados con la inteligencia artificial se implementan usando soluciones recursivas.

EJEMPLO:

```
public static int calcularFactorial(int n) {
    if (n == 0) {
        return 1; // Caso base: el factorial de 0 es 1
    } else {
        // Llamada recursiva: n! = n * (n-1)!
        return n * calcularFactorial(n - 1);
    }
}

int numero = 5; // Puedes cambiar el número aquí
int factorial = calcularFactorial(numero);
System.out.println("El factorial de " + numero + " es " + factorial);
```

A la hora de “jugar” con el ejemplo es interesante que insertes sentencias **println()** en el código de la función para ver qué valores parciales se van devolviendo en cada momento.

Por otro lado, a la hora de programar un procedimiento o función recursiva debes tener en cuenta que siempre debe aparecer una sentencia **if** en algún sitio para obligar a dicho procedimiento/función recursiva a volver sin que la llamada recursiva sea ejecutada (es a lo que en programación se conoce como **caso base**). **¿Cuál es el caso base en el ejemplo?**

1.1.6. Métodos

En Java, aunque se puede definir y usar procedimientos y funciones, éstos no reciben ese nombre. En ambos casos se denominan métodos, de forma que un *método puede ser tanto un procedimiento como una función en el paradigma de la programación orientada a objetos*.

La **sintaxis** de un método en Java es la siguiente:

```
[modificador de acceso] [modificador de no acceso] tipo-de-retorno nombreDelMetodo([parámetros]) {
    // Cuerpo del método
    // Puede contener instrucciones y declaraciones
    return valor; // Opcional, depende del tipo-de-retorno
}
```

Los corchetes [] indican que el valor que contienen puede no especificarse, es decir, que en función del método concreto que estemos programando, dicho valor puede ser omitido.

El significado se la sintaxis es el siguiente:

- **modificador de acceso:** Puede ser **public**, **private**, **protected**, o no especificarse (por defecto, un método es accesible sólo dentro del mismo paquete).
 - **public:** El método es accesible desde cualquier parte del programa.
 - **private:** El método es accesible solo dentro de la misma clase.
 - **protected:** El método es accesible dentro de la misma clase y en subclases (heredadas).
 - (Sin modificador): El método es accesible solo dentro del mismo paquete (paquete por defecto). Este acceso se conoce como **package-private**.
- **modificador de NO acceso:** Puede ser **static**, **final**, **abstract**, u otros, dependiendo de la funcionalidad del método.
 - **static:**
 - El modificador **static** se utiliza para declarar **miembros (métodos o variables)** de una clase como estáticos, lo que significa que pertenecen a la clase en lugar de a una instancia específica de la clase.
 - Los **métodos estáticos** se pueden llamar a través del nombre de la clase sin necesidad de crear una instancia de la clase.
 - Las **variables estáticas** son compartidas por todas las instancias de la clase y almacenan datos que deben ser compartidos entre todas ellas.
 - **final:**
 - El modificador **final** se utiliza para indicar que un miembro de una clase no puede ser modificado una vez que se ha inicializado.
 - **Para variables**, significa que se trata de una constante, cuyo valor no puede cambiarse.
 - **Para métodos**, indica que el método no puede ser sobrescrito en clases hijas.
 - **abstract:**
 - El modificador **abstract** se utiliza en clases y métodos. En una clase, indica que la clase es una clase abstracta, lo que significa que no se pueden crear instancias directas de esa clase.
 - En un método, indica que el método no tiene implementación en la clase actual y debe ser implementado en una clase derivada (subclase). Las clases que contienen al menos un método abstracto deben ser declaradas como abstractas.

- **synchronized:** (se usa en entornos de concurrencia)
 - El modificador **synchronized** se utiliza en métodos para controlar la concurrencia en entornos de múltiples hilos (threads).
 - Indica que solo un hilo puede ejecutar ese método a la vez, lo que evita problemas de concurrencia como condiciones de carrera.
- **volatile:** (se usa en entornos de concurrencia)
 - El modificador **volatile** se utiliza en variables para indicar que una variable puede ser modificada por múltiples hilos y que las lecturas y escrituras a esta variable deben ser atómicas.
 - Ayuda a evitar problemas de concurrencia al garantizar que las actualizaciones de la variable sean visibles para todos los hilos.
- **tipo-de-retorno:** Indica el tipo de valor que devuelve el método. Puede ser cualquier tipo de dato, incluyendo tipos primitivos tipos de referencia o clases personalizadas. El valor **void** indica que se trata de un procedimiento, puesto que el valor devuelto es “vacío”.
- **nombreDelMetodo:** El nombre que identifica al método.
- **[parámetros]:** Los parámetros son valores que se pasan al método para su procesamiento. Pueden ser opcionales.

EJEMPLO: Método público.

```
public class EjemploMetodoPublic {
    public void mostrarMensaje() {
        System.out.println("Este es un método público.");
    }
}
```

¿Por qué el método main() se declara como public y static?

EJEMPLO: Método privado.

```
public class EjemploMetodoPrivate {
    private void realizarTareaPrivada() {
        System.out.println("Este es un método privado.");
    }

    public void realizarTareaPublica() {
        realizarTareaPrivada(); // Se puede llamar al método privado desde uno público en la misma clase.
    }
}
```

EJEMPLO: Método protegido.

```
public class EjemploMetodoProtected {
    protected void saludar() {
        System.out.println("Hola, soy un método protegido.");
    }
}

class Subclase extends EjemploMetodoProtected {
    public void mostrarSaludo() {
        saludar(); // Se puede llamar al método protegido en una subclase.
    }
}
```

EJEMPLO: Método público con parámetros.

```
public class MetodoConParametros {
    public int sumar(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        MetodoConParametros calculadora = new MetodoConParametros();
        int resultado = calculadora.sumar(5, 3);
        System.out.println("La suma es: " + resultado);
    }
}
```

Entendiendo bien el *modificador de NO acceso static*:

- Se usa cuando queremos definir un miembro de una clase (variable o método) que esté disponible para su uso independientemente de cualquier objeto de la clase. De esta forma dicho miembro puede ser usado antes de que se cree cualquier objeto de la clase.
- Todas las instancias de la clase comparten las variables estáticas que esta tenga definidas.
- Un método estático tiene varias restricciones:
 - Solamente pueden llamar directamente a otros métodos estáticos de su clase.
 - Solamente pueden acceder a variables estáticas de su clase.
 - No pueden usar las referencias **this** o **super**. (Las veremos más adelante)
- Para acceder o usar un miembro estático de una clase solamente debemos poner el nombre de la clase a la que pertenezca, seguido de un punto y a continuación el nombre del miembro que queremos utilizar. Un ejemplo de uso de un método estático es la llamada **System.out.print("Hola, mundo")**;

Ahora vamos a hablar de la palabra reservada **final** que se usa para definir elementos constantes o bien que solamente se pueden inicializar una vez. Por ejemplo:

- Declarando un parámetro como final se evita que pueda modificarse dentro del método en que se usa.
- Declarando una variable local como final se evita que se le pueda asignar un valor más de una vez.
- También podemos declarar un método como final, pero eso lo trataremos cuando hablemos de herencia.

1.2. Ámbito y tiempo de vida de las variables

Java permite la declaración de variables dentro de cualquier bloque (tanto anónimo como con nombre). Recuerda que un bloque está delimitado entre llaves {}.

De esta forma, cada bloque define lo que en programación se conoce como **ámbito** y cuando creamos un nuevo bloque también estaremos definiendo un nuevo **ámbito**. Un **ámbito** determina:

- Qué objetos son visibles a otras partes de nuestro programa.
- El **tiempo de vida** de dichos objetos.

Comúnmente los programadores piensan en dos categorías generales de **ámbito**: **global** y **local**, pero esto no encaja bien en Java donde los **ámbitos** más habituales son aquellos delimitados por una **clase** o un **método**, aunque esta distinción también sea algo artificial. Más adelante trataremos los **ámbitos** definidos por una clase, pero ahora nos vamos a centrar en las características del **ámbito definido por un método**:

- El **ámbito** empieza con la llave de apertura del **método {**.
- Si el **método** tiene **parámetros**, estos también formarán parte de su **ámbito**.
- El **ámbito** termina con la llave de cierre del **método }**.
- Este **bloque de código** se llama **cuerpo del método**.

En cuanto a los **ámbitos**, tenemos las siguientes reglas generales:

- Una variable definida en un **ámbito** no es visible (accesible) desde el código que esté definido fuera de dicho **ámbito**. Dicho de otra forma: cuando definimos una variable en un **ámbito** la estamos protegiendo frente a accesos/modificaciones no autorizados desde fuera. Este es un de los principios de la encapsulación.
- Una variable definida dentro de un bloque/ámbito se denomina **variable local**.
- Los **ámbitos** pueden anidarse. De hecho, cada vez que creas un **bloque de código** (if, for, etc.) estás creando un nuevo **ámbito** anidado con aquel en el que ya estás. Esto significa que aquellos **objetos** y **variables** que existan en el **ámbito** en el que ya estás serán visibles en el nuevo **ámbito** anidad, pero lo contrario no es cierto, es decir, los **objetos** y **variables** declarados dentro del **ámbito** anidad no son visibles desde fuera.

Veamos un ejemplo para entender cómo funciona la anidación de los ámbitos:

```
// Demonstrate block scope.
class Scope {
    public static void main(String[] args) {
        int x; // known to all code within main

        x = 10;
        if(x == 10) { // start new scope
            int y = 20; // known only to this block

            // x and y both known here.
            System.out.println("x and y: " + x + " " + y);
            x = y * 2;
        }
        // y = 100; // Error! y not known here

        // x is still known here.
        System.out.println("x is " + x);
    }
}
```

Fuente: Java – The Complete Reference Twelfth Edition – Herbert Schildt

¿Qué muestra este código en la pantalla?

Más consideraciones en cuanto a las variables en cuanto a su tiempo de vida:

- Una variable puede ser definida/declarada en cualquier parte de un bloque, pero solamente será válida/conocida después de haber sido definida/declarada. Por tanto, si definimos una variable al comienzo de un método, estará disponible para todo el código de dicho método, pero si la declaramos al final del bloque esto sería inútil ya que ningún código la tendría disponible.
- Una variable se crea dentro de su ámbito y se destruye cuando dicho ámbito termina.
- Si una variable es inicializada en un ámbito/bloque, dicha inicialización se llevará a cabo cada vez que se entre a dicho ámbito/bloque.
- Una variable declarada dentro de un método no conserva su valor entre distintas llamadas a dicho método.

En resumen, el tiempo de vida de una variable está determinado por su ámbito.

Veamos un caso particular que puede haberos pasado ya:

¿Qué muestra este programa en pantalla?

```
// Demonstrate lifetime of a variable.
class LifeTime {
    public static void main(String[] args) {
        int x;

        for(x = 0; x < 3; x++) {
            int y = -1; // y is initialized each time block is entered
            System.out.println("y is: " + y); // this always prints -1
            y = 100;
            System.out.println("y is now: " + y);
        }
    }
}
```

Fuente: Java – The Complete Reference Twelfth Edition – Herbert Schildt

Este es el resultado:

```
y is: -1
y is now: 100
y is: -1
y is now: 100
y is: -1
y is now: 100
```

Fuente: Java – The Complete Reference Twelfth Edition – Herbert Schildt

Por último, aunque los bloques pueden anidarse, no se puede definir una variable con el mismo nombre de otra variable que ya exista en un bloque externo. Por ejemplo, este código es ilegal:

```
// This program will not compile
class ScopeErr {
    public static void main(String[] args) {
        int bar = 1;
        {
            int bar = 2; // Compile-time error - bar already defined!
        }
    }
}
```

Fuente: Java – The Complete Reference Twelfth Edition – Herbert Schildt

1.3. Clase principal y método main()

En Java, cada aplicación debe contener una clase principal y, dentro de esta clase, un método denominado **main()**. *Este método es el punto de entrada del programa, es decir, por donde la máquina virtual de Java (JVM) comienza la ejecución.*

Por tanto, la estructura básica de un programa en Java es la siguiente:

```
public class NombreClasePrincipal {  
    public static void main(String[] args) {  
        // Código del programa  
    }  
}
```

- **public:** Es un modificador de acceso que permite que la clase sea accesible desde cualquier otra clase.
- **class:** Es la palabra clave que se utiliza para definir una clase.
- **NombreClasePrincipal:** Es el nombre que le damos a nuestra clase principal. Por convención, **los nombres de clase comienzan con mayúsculas**.
- **main:** Es el nombre del método que la JVM llama para iniciar el programa.
- **String[] args:** Es el parámetro del método **main()**. Representa un array de **String**, y recibe los argumentos que se pasan desde la línea de comandos al iniciar el programa. Calma, en el siguiente punto veremos qué son los arrays y cómo podemos manejarlos.

EJEMPLO: Veamos un ejemplo que hace uso de una enumeración y de un método que recibe un parámetro.

```
public class EjemploEnumeracion {

    // Enumeración simple de días de la semana.
    enum Dia {
        LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO
    }

    // Método estático para imprimir el día.
    public static void imprimirDia(Dia dia) {
        System.out.println("El día actual es: " + dia);
    }

    public static void main(String[] args) {
        Dia diaActual = Dia.MIERCOLES;
        imprimirDia(diaActual);
    }
}
```

Como puedes ver en el ejemplo anterior el método `main()` recibe argumentos de la línea de comandos a través del array de Strings `args`. Cada elemento en este array representa un argumento diferente, y se accede a ellos por su índice.

EJEMPLO: Accedemos a los argumentos de la línea de comandos.

```
public class ArgumentosMain {
    public static void main(String[] args) {
        System.out.println("Número de argumentos pasados al programa: " + args.length);

        // Iterar y mostrar los argumentos.
        for (int i = 0; i < args.length; i++) {
            System.out.println("Argumento " + i + ": " + args[i]);
        }
    }
}
```

Para ejecutar el programa desde la consola y pasarle argumentos podríamos hacerlo así:

```
java ArgumentosMain PrimerArgumento SegundoArgumento
```

De esta forma le indicamos a `java` que queremos ejecutar el programa definido a partir de la clase `ArgumentosMain` y además le proporcionamos dos parámetros con el valor `PrimerArgumento` y `SegundoArgumento`.

Así pues el programa devolvería la siguiente salida por pantalla:

```
Número de argumentos pasados al programa: 2
Argumento 0: PrimerArgumento
Argumento 1: SegundoArgumento
```

El programa indica primero que se pasaron dos argumentos (el tamaño del array `args`), y luego procede a imprimir cada argumento con su respectivo índice dentro del array `args`.

IMPORTANTE: Los parámetros que recibe un programa por línea de comandos siempre son de tipo String y, por lo tanto, si le proporcionamos al programa valores numéricos habrá que convertirlos de tipo String al tipo numérico que nos interese (byte, short, int, long, float o double). Lo mismo se aplicaría para otros tipos de datos.

A continuación, te enumero otras consideraciones importantes a tener en cuenta a la hora de crear programas en Java:

- **Paquetes:** En proyectos más grandes, las clases suelen organizarse en paquetes para mejorar la gestión y estructura del código.
- **Importaciones:** Al principio de un archivo de Java, se pueden incluir sentencias `import` para utilizar clases o interfaces de otros paquetes.
- **Comentarios:** Es una buena práctica incluir comentarios que describan el código. Los comentarios de una sola línea comienzan con `//`, mientras que los comentarios de varias líneas se encierran entre `/*` y `*/`. También puedes usar comentarios Javadoc.
- **Convenciones de Nombres:** Sigue las convenciones de nombres de Java para clases, métodos, variables y constantes para mejorar la legibilidad del código.
- **Tratamiento de Excepciones:** El código que puede causar errores en tiempo de ejecución debe manejarse apropiadamente usando bloques `try-catch`. Esto también lo veremos más adelante en este mismo tema.

Estos puntos ayudan a crear un código más estructurado, legible y mantenible, lo cual es esencial en la formación de buenos principios de programación.

2. Identificadores

Los identificadores son nombres utilizados para identificar variables, métodos, clases y otros elementos en un programa de Java. Deben seguir algunas reglas específicas:

- Deben comenzar con una letra (a-z o A-Z), el carácter de subrayado (_) o el carácter \$.
- Pueden contener letras, dígitos (0-9), el carácter de subrayado (_) y el carácter \$.
- No pueden ser una palabra reservada (palabra clave).
- Son sensibles a mayúsculas y minúsculas. ("Case Sensitive, en inglés)

Ejemplos de identificadores válidos:

```
int edad;
double alturaPersona;
String nombreCompleto;
```

En Java, se suele utilizar la notación llamada "CamelCase" para los identificadores. La notación CamelCase es un estilo de escritura en el que las palabras dentro del identificador se escriben juntas sin espacios. Hay dos variantes comunes de CamelCase:

1. **UpperCamelCase o PascalCase:** La primera letra de cada palabra se escribe en mayúscula.

Ejemplos: `NombreCompleto`, `EdadPersona`, `ClaseJava`.

2. **lowerCamelCase:** La primera letra de la primera palabra se escribe en minúscula, pero la primera letra de las palabras siguientes se escribe en mayúscula.

Ejemplos: `nombreUsuario`, `edadPersona`, `calificacionFinal`.

La notación CamelCase es ampliamente utilizada en Java y es un estándar de estilo aceptado por la comunidad de desarrolladores. Se utiliza para nombrar variables, métodos, clases y otros identificadores en el código fuente. Al seguir esta convención de nomenclatura, se mejora la legibilidad del código y facilita la comprensión de su estructura.

3. Palabras reservadas

Las palabras reservadas, o palabras clave, son términos que tienen un significado especial en el lenguaje Java y no pueden ser utilizadas como identificadores. Estas palabras tienen un propósito específico en la sintaxis del lenguaje y no pueden ser modificadas. Ejemplos:

```
class
if
else
for
while
int
double
```

4. Variables

Una variable es un espacio de memoria que se utiliza para almacenar datos en un programa de Java. Deben declararse antes de ser utilizadas y deben especificar el tipo de dato que pueden contener. Además, para usar una variable en Java, primero debemos declararla y luego asignarle un valor, aunque podemos hacerlo en la misma línea del programa.

La declaración de una variable incluye su tipo y su nombre:

```
int edad; // Declaración de la variable "edad"  
edad = 25; // Asignación del valor 25 a la variable "edad"
```

También se puede declarar y asignar una variable en una sola línea:

```
int cantidad = 10; // Declaración y asignación de la variable "cantidad"
```

5. Tipos de datos

En Java, los tipos de datos definen el tipo de valor que una variable puede contener. Los tipos de datos se dividen en dos categorías principales que pasamos a ver a continuación.

5.1. Tipos primitivos

En Java, los tipos primitivos son datos fundamentales y representan valores simples que no son objetos. Si se necesita trabajar con datos más complejos o realizar operaciones más avanzadas, se utilizan objetos y clases definidas en Java. A continuación, se presenta un resumen de cada uno de los **ocho tipos primitivos de Java**. Excepto los dos últimos, todos sirven para definir valores numéricos y se presentan de menor a mayor capacidad de representación numérica.

5.1.1. byte

Es un tipo de dato entero que almacena valores en un rango de -128 a 127.

Ejemplo:

```
byte edad = 25;
```

5.1.2. short

Es un tipo de dato entero que almacena valores en un rango de -32,768 a 32,767.

Ejemplo:

```
short poblacion = 20000;
```

5.1.3. int

Es un tipo de dato entero que almacena valores en un rango de -2,147,483,648 a 2,147,483,647. Es el tipo de dato más comúnmente utilizado para representar números enteros.

Ejemplo:

```
int cantidad = 1000;
```

5.1.4. long

Es un tipo de dato entero que almacena valores en un rango más amplio de -9,223,372,036,854,775,808 a 9,223,372,036,854,775,807.

Ejemplo:

```
long numeroGrande = 10000000000L; // Nota: se debe añadir la 'L' al final del valor para indicar que es de tipo long.
```

5.1.5. float

Es un tipo de dato decimal de precisión simple que almacena valores en un rango aproximado de 3.4e-38 a 3.4e38.

Ejemplo:

```
float precio = 10.99f; // Nota: se debe añadir la 'f' al final del valor para indicar que es de tipo float.
```

5.1.6. double

Es un tipo de dato decimal de doble precisión que almacena valores en un rango aproximado de 1.7e-308 a 1.7e308. Es el tipo de dato más comúnmente utilizado para números decimales.

Ejemplo:

```
double temperatura = 25.5;
```

5.1.7. char

Es un tipo de dato que almacena un solo carácter Unicode. Se utiliza para representar caracteres individuales, como letras o símbolos.

Ejemplo:

```
char inicial = 'J';
```

5.1.8. boolean

Es un tipo de dato que solo puede tomar dos valores: `true` o `false`. Se utiliza para representar valores de verdad o estados lógicos.

Ejemplo:

```
boolean esMayorDeEdad = true;
```

5.2. Tipos de referencia

Representan objetos y su valor es la referencia a la ubicación en memoria del objeto. En Java, los tipos de referencia son aquellos que hacen referencia a objetos en la memoria en lugar de contener directamente los datos. Estos tipos de datos almacenan direcciones de memoria donde se encuentran los objetos, y se utilizan para manipular y trabajar con estructuras de datos más complejas. Los tipos de referencia son una parte esencial de la programación orientada a objetos en Java y son fundamentales para trabajar con objetos y clases.

¡CUIDADO! En Java, los tipos de referencia empiezan con una letra mayúscula para así poder diferenciarlos de los tipos primitivos que se escriben siempre en minúsculas.

A continuación, vamos a ver dos ejemplos.

5.2.1. String

El tipo String es una clase predefinida en Java que representa una secuencia de caracteres. Las cadenas son inmutables, lo que significa que no se pueden cambiar una vez creadas. Puedes manipular cadenas mediante métodos, pero cada manipulación crea una nueva cadena en memoria.

Ejemplo:

```
String nombre = "Juan"; // Creación de un objeto String en memoria
```

5.2.2. Date

El tipo Date es una clase que se utiliza para representar fechas y horas en Java. Aunque a partir de Java 8, se recomienda usar la nueva API Date/Time, java.time, el siguiente ejemplo sigue siendo válido.

Ejemplo:

```
import java.util.Date;

// Creación de un objeto Date con la fecha y hora actuales
Date fechaActual = new Date();
System.out.println("Fecha actual: " + fechaActual);
```

Cuando veamos qué son las clases y los objetos trabajaremos más a fondo con los tipos de referencia de forma que por ahora es suficiente con saber que existen y cuál es su función.

6. Literales

Son valores constantes que se utilizan directamente en el código. Por ejemplo, un literal entero es simplemente un número entero sin ninguna operación adicional.

A continuación te voy a mostrar un ejemplo de cada uno de los tipos de literales disponibles en Java.

6.1. Literales enteros

Los literales enteros representan valores numéricos enteros sin punto decimal. Pueden expresarse en diferentes bases, como decimal, hexadecimal u octal.

Ejemplos:

```
int decimalLiteral = 10;      // Literal decimal
int hexadecimalLiteral = 0xA; // Literal hexadecimal (el prefijo '0x' indica que es hexadecimal)
int octalLiteral = 012;       // Literal octal (el prefijo '0' indica que es octal)
```

6.2. Literales de punto flotante

Los literales de punto flotante representan valores numéricos con punto decimal, como números de punto flotante y números dobles.

Ejemplos:

```
float floatLiteral = 3.14f;    // Literal de punto flotante (se usa la letra 'f' para indicar que es float)
double doubleLiteral = 2.718;   // Literal doble (por defecto, los literales con punto decimal son double)
```

6.3. Literales de caracteres

Los literales de caracteres representan un solo carácter en Java y se declaran entre comillas simples.

Ejemplo:

```
char charLiteral = 'A'; // Literal de carácter (representa el carácter 'A')
```

6.4. Literales de cadena de caracteres (String)

Los literales de cadena de caracteres representan una secuencia de caracteres y se declaran entre comillas dobles.

Ejemplo:

```
String stringLiteral = "Hola, mundo!"; // Literal de cadena de caracteres
```

6.5. Literales booleanos

Los literales booleanos representan los dos valores de verdad: verdadero (true) o falso (false).

Ejemplos:

```
boolean trueLiteral = true; // Literal booleano con valor verdadero
boolean falseLiteral = false; // Literal booleano con valor falso
```

6.6. Literales de valores nulos (null)

El literal `null` representa la ausencia de valor y se utiliza para inicializar objetos que aún no se han asignado.

Ejemplo:

```
String cadenaNula = null; // Literal nulo para inicializar una variable de tipo String
```

7. Constantes

Una constante es una variable cuyo valor no cambia durante la ejecución del programa. En Java, se declaran utilizando la palabra clave "**final**". Por convención, los nombres de las constantes se escriben en mayúsculas, como veremos después.

Ejemplos:

```
// Constante para la velocidad de la luz en metros por segundo.  
public final double velocidadDeLaLuz = 299792458.0;  
  
// Constante para el número de días en una semana.  
public final int diasEnUnaSemana = 7;
```

7.1. Notación SCREAMING_SNAKE_CASE

En el ámbito de la programación en Java, la notación **SCREAMING_SNAKE_CASE** se refiere a una convención de nomenclatura utilizada para declarar constantes. En esta convención, *los identificadores de las constantes están escritos en letras mayúsculas y separados por guiones bajos ("_")*. Por ejemplo, una constante que representa el valor máximo de intentos permitidos en un programa podría ser declarada de la siguiente manera:

Ejemplo:

```
public static final int MAX_ATTEMPTS = 5;
```

En este caso, "MAX_ATTEMPTS" es un identificador en SCREAMING_SNAKE_CASE que indica que se trata de una constante y que su valor no debe cambiar durante la ejecución del programa.

Esta convención es una forma de hacer que las constantes se destaquen fácilmente en el código y se distingan de las variables, que generalmente siguen la convención de escritura en camelCase (por ejemplo, "miVariable"). El SCREAMING_SNAKE_CASE se considera más legible para constantes, ya que la combinación de letras mayúsculas y guiones bajos hace que los nombres sean más claros y fácilmente distinguibles.

8. Operadores y expresiones

Los **operadores** son símbolos especiales que se utilizan para realizar operaciones en variables y valores. Las **expresiones** son combinaciones de variables, valores y operadores que producen un nuevo valor.

8.1. Operadores binarios

8.1.1. Operadores aritméticos

```
int a = 10;
int b = 3;

int suma = a + b;           // Resultado: 13
int resta = a - b;          // Resultado: 7
int multiplicacion = a * b; // Resultado: 30
int division = a / b;       // Resultado: 3
int modulo = a % b;         // Resultado: 1
```

El **operador módulo** devuelve el resto de la división de dos números enteros.

8.1.2. Operadores relacionales

```
int x = 5;
int y = 8;

boolean igual = (x == y);    // Resultado: false
boolean noIgual = (x != y);  // Resultado: true
boolean mayorQue = (x > y); // Resultado: false
boolean menorQue = (x < y); // Resultado: true
boolean mayorOIgual = (x >= y); // Resultado: false
boolean menorOIgual = (x <= y); // Resultado: true
```

El operador **!=** también se conoce como **distinto**.

8.1.3. Operadores lógicos

```
boolean p = true;
boolean q = false;

boolean andLogico = (p && q); // Resultado: false
boolean orLogico = (p || q);   // Resultado: true
boolean notLogico = !p;        // Resultado: false
```

Traducidos al español, se llamarían “y lógico”, “o lógico” y “no lógico”.

8.1.4. Operadores especiales

```
// Operador instanceof
Animal animal = new Perro();
boolean esPerro = animal instanceof Perro; // Resultado: true, porque animal es una instancia de Perro.

// Operador ternario (Operador condicional)
int edad = 17;
String mensaje = (edad >= 18) ? "Eres mayor de edad" : "Eres menor de edad";
// Resultado: "Eres menor de edad", porque la edad es menor que 18.

// Operadores de asignación
int num = 10;
num += 5; // Resultado: num es ahora 15.
num -= 3; // Resultado: num es ahora 12.
num *= 2; // Resultado: num es ahora 24.
num /= 4; // Resultado: num es ahora 6.
num %= 2; // Resultado: num es ahora 0.

// Incremento y Decremento
int x = 5;
int y = 3;
int incrementoX = ++x; // Resultado: incrementoX es 6, x es 6.
int decrementoY = y--; // Resultado: decrementoY es 3, y es 2.
```

8.2. Operadores unarios en Java

```
int i = 5;
int j = -3;

int incremento = ++i;    // Resultado: incremento es 6, i es 6.
int decremento = j--;    // Resultado: decremento es -3, j es -4.
```

8.3. Operadores de bit

```
int num1 = 5; // Representado en binario como 00000101
int num2 = 3; // Representado en binario como 00000011

int bitAND = num1 & num2; // Resultado: 00000001 (1 en decimal)
int bitOR = num1 | num2; // Resultado: 00000111 (7 en decimal)
int bitXOR = num1 ^ num2; // Resultado: 00000110 (6 en decimal)
int complemento = ~num1; // Resultado: 1111010 (-6 en decimal, debido a la representación en complemento a dos)
int desplIzquierda = num1 << 2; // Resultado: 00010100 (20 en decimal)
int desplDerecha = num1 >> 1; // Resultado: 00000010 (2 en decimal)
int desplDerechaSinSigno = num1 >>> 1; // Resultado: 00000010 (2 en decimal)
```

8.4. Tablas resumen con los significados de cada operador

8.4.1. Tabla de operadores binarios en Java

Clasificación	Operador	Significado
Aritméticos	+	Suma
	-	Resta
	*	Multiplicación
	/	División
	%	Módulo (Resto de la división)
Relacionales	==	Igual a
	!=	No igual a
	>	Mayor que
	<	Menor que
	>=	Mayor o igual que
	<=	Menor o igual que
Lógicos	&&	AND (Y)
		OR (O)
	!	NOT (NO)
Especiales	?:	Ternario (Operador condicional)
	=	Asignación
	+=	Asignación con suma
	-=	Asignación con resta
	*=	Asignación con multiplicación
	/=	Asignación con división
	%=	Asignación con módulo (Resto)

8.4.2. Tabla de operadores unarios en Java

Operador	Significado
++	Incremento (por ejemplo, x++)
--	Decremento (por ejemplo, x--)
+	Operador positivo (unary plus)
-	Operador negativo (unary minus)
!	NOT lógico

8.4.3. Tabla de operadores de bit en Java

Operador	Significado
&	AND a nivel de bit
	OR a nivel de bit
^	XOR (OR exclusivo) a nivel de bit
~	Complemento a nivel de bit
<<	Desplazamiento a la izquierda
>>	Desplazamiento a la derecha
>>>	Desplazamiento a la derecha sin signo

8.5. Precedencia de operadores

La **precedencia** de operadores en Java determina el orden en el que se evalúan las expresiones que contienen varios operadores. Los operadores con mayor precedencia se evalúan primero, mientras que los de menor precedencia se evalúan después. En caso de tener operadores con la misma precedencia, se evalúan de izquierda a derecha.

A continuación, se presenta una tabla con los principales operadores en Java, ordenados de mayor a menor precedencia:

Precedencia	Operador	Descripción
Mayor	`()`	Paréntesis (controla la evaluación de expresiones)
	`++` `--`	Incremento y decremento
	`+` `--`	Operador unario positivo y negativo
	`!`	Operador lógico NOT (negación)
	`~`	Operador bitwise NOT (complemento a uno)
	`(tipo)`	Casting (conversión explícita de tipos)
	`**` `/` `%`	Operadores aritméticos (multiplicación, división, módulo)
	`+` `-`	Operadores aritméticos (suma, resta)
	`<<` `>>` `>>>`	Operadores bitwise de desplazamiento
	`<` `<=` `>` `>=`	Operadores de comparación
	`instanceof`	Operador de comprobación de tipo
	`==` `!=`	Operadores de igualdad
	`&`	Operador bitwise AND
	`^^`	Operador bitwise XOR (OR exclusivo)
	` `	Operador bitwise OR
	`&&`	Operador lógico AND
	` `	Operador lógico OR
	`?` `:`	Operador ternario (conditional)
Menor	`=`, `+=`, `-=` ... (y otros)	Operadores de asignación

9. Conversiones de tipo

Las conversiones de tipo, también conocidas como "**casting**", se utilizan para convertir un valor de un tipo de dato a otro tipo de dato compatible.

Hay dos tipos de conversiones:

- **Conversión implícita:** El casting implícito ocurre cuando se realiza una conversión automática de un tipo de dato de menor tamaño a uno de mayor tamaño, sin que se pierda información.

Ejemplo de casting implícito:

```
int numeroEntero = 42; // Un número entero
double numeroDouble = numeroEntero; // Casting implícito de int a double
```

- **Conversión explícita:** El casting explícito implica una pérdida de información, puesto que se realiza una conversión de un tipo de dato de mayor tamaño a otro de menor tamaño.

Ejemplo de casting explícito:

```
double precio = 19.99; // El valor del precio es un double (con decimales)
int precioEntero = (int) precio; // Conversión de double a entero, se trunca la parte decimal
```

9.1. Relación de conversiones implícitas

```
// Conversiones implícitas de enteros a flotantes
byte numeroByte = 10;
short numeroShort = 1000;
int numeroInt = 200000;
long numeroLong = 1234567890L;

float numeroFloat1 = numeroByte;
float numeroFloat2 = numeroShort;
float numeroFloat3 = numeroInt;
float numeroFloat4 = numeroLong;

// Conversiones implícitas de flotantes a dobles
float numeroFloat = 3.14f;
double numeroDouble1 = numeroFloat;

// Conversiones implícitas de enteros a dobles
double numeroDouble2 = numeroByte;
double numeroDouble3 = numeroShort;
double numeroDouble4 = numeroInt;
double numeroDouble5 = numeroLong;
```

9.2. Relación de conversiones explícitas

```
// Conversiones explícitas de flotantes a enteros
float numeroFloat = 123.456f;
double numeroDouble = 456.789;

byte numeroByte = (byte) numeroFloat;
short numeroShort = (short) numeroFloat;
int numeroInt = (int) numeroFloat;
long numeroLong = (long) numeroFloat;

// Conversiones explícitas de dobles a enteros
int numeroEntero1 = (int) numeroDouble;
long numeroEntero2 = (long) numeroDouble;
```

Por ejemplo, si convertimos un entero con valor 127 a byte, este almacenará el mismo valor, pero si el entero tiene un valor de 128, el byte almacenará un valor de -128.

```
int valorEntero1 = 127;
int valorEntero2 = 128;

// Conversión de entero a byte
byte byte1 = (byte) valorEntero1;
byte byte2 = (byte) valorEntero2;
```

10. Comentarios

Los **comentarios** son notas dentro del código que se utilizan para explicar el propósito o funcionamiento del mismo. Los comentarios no se compilan y no afectan la ejecución del programa. En Java, existen tres tipos principales de comentarios:

10.1. Comentario de una sola línea

Estos comentarios *comienzan con `//` y abarcan solo una línea de código*. Se utilizan para agregar explicaciones breves y claras sobre una línea específica.

Ejemplo:

```
// Esto es un comentario de una sola línea
int numero = 42; // También se puede colocar un comentario al final de una línea de código
```

10.2. Comentario de múltiples líneas

Estos comentarios *comienzan con `/*` y terminan con `*/`*. Pueden abarcar varias líneas y son útiles para agregar explicaciones más extensas.

Ejemplo:

```
/* Esto es un comentario de múltiples líneas.  
 Se extiende por varias líneas de código.  
 Es útil para proporcionar información detallada. */  
int x = 10;  
int y = 20;  
int resultado = x + y; // Suma los valores de 'x' e 'y'
```

10.3. Comentario de documentación (Javadoc)

Estos comentarios *comienzan con `/**` y terminan con `*/`*. Se utilizan para generar documentación automática utilizando la herramienta Javadoc. Pueden incluir información sobre clases, métodos, parámetros y retornos, y se utilizan para generar documentación detallada para el código.

Ejemplo:

```
/**  
 * Esta clase representa un ejemplo de comentarios en Java.  
 * Se utiliza para mostrar los diferentes tipos de comentarios.  
 */  
public class CommentExample {  
    /**  
     * Este método suma dos números enteros y devuelve el resultado.  
     *  
     * @param a El primer número entero.  
     * @param b El segundo número entero.  
     * @return La suma de los dos números enteros.  
     */  
    public static int sumar(int a, int b) {  
        return a + b;  
    }  
  
    public static void main(String[] args) {  
        int x = 10;  
        int y = 20;  
        int resultado = sumar(x, y); // Llamada al método sumar  
        System.out.println(resultado);  
    }  
}
```

Los comentarios de documentación Javadoc son especialmente útiles cuando se generan documentos API para un proyecto, ya que proporcionan información detallada sobre cómo usar clases y métodos. Además, Javadoc permite convertir los comentarios a un formato web (HTML) para aportar la documentación en línea de la aplicación que se esté desarrollando.

10.3.1. Opciones de Javadoc más utilizadas

A continuación, te presento una tabla con algunas de las opciones de parametrización más comunes que se pueden utilizar en Javadoc, junto con sus significados:

Opción	Significado
`@param`	Describe un parámetro de un método o constructor. Se utiliza para explicar qué representa cada parámetro y cómo se debe usar. Ejemplo: `@param nombre Nombre del usuario.`
`@return`	Describe el valor de retorno de un método. Se utiliza para explicar qué valor se devuelve y su significado. Ejemplo: `@return La suma de los elementos.`
`@throws`	Documenta las excepciones que puede lanzar un método. Se utiliza para indicar qué excepciones pueden ocurrir y bajo qué circunstancias. Ejemplo: `@throws NullPointerException Si el argumento es nulo.`
`@see`	Hace referencia a otra clase, método o campo relacionado con el elemento actual. Se utiliza para proporcionar enlaces a otras partes de la documentación. Ejemplo: `@see OtraClase#metodoRelacionado`
`@since`	Indica desde qué versión de la API está disponible un elemento. Se utiliza para mostrar cuándo se introdujo por primera vez un método o clase. Ejemplo: `@since 1.0`
`@deprecated`	Marca un elemento como obsoleto. Se utiliza para indicar que un método, clase o campo ya no se debe usar y se proporciona información sobre la alternativa recomendada. Ejemplo: `@deprecated Reemplazado por otroMetodo().`
`@version`	Indica la versión de la clase o del paquete. Se utiliza para especificar la versión actual del software. Ejemplo: `@version 2.0`
`@author`	Identifica al autor del código o la documentación. Se utiliza para mostrar quién es el responsable de un elemento. Ejemplo: `@author Juan Pérez`

Es importante tener en cuenta que esta tabla no incluye todas las opciones de parametrización disponibles en Javadoc, ya que hay muchas otras opciones y también existen etiquetas personalizadas que se pueden utilizar según las necesidades del proyecto. La tabla muestra solo las opciones más comunes y ampliamente utilizadas.

11. Entrada/Salida estándar

En el desarrollo de aplicaciones en Java, el manejo de la entrada y salida estándar es fundamental para interactuar con los usuarios y mostrar resultados. A continuación, veremos cómo realizar la entrada desde el teclado y la salida a pantalla utilizando la entrada/salida estándar en Java.

11.1. Entrada desde teclado

Para recibir datos desde el teclado, utilizaremos la clase **Scanner** de Java, que proporciona métodos para leer diferentes tipos de datos de manera sencilla. Antes de usar Scanner, debemos importar la clase en nuestro programa. A continuación, se muestra un ejemplo de cómo realizar la entrada de un número entero desde el teclado:

Ejemplo:

```
import java.util.Scanner;

Scanner scanner = new Scanner(System.in);

System.out.print("Ingresa un número entero: ");
int numero = scanner.nextInt();

System.out.println("El número ingresado es: " + numero);

scanner.close(); // Cerramos el Scanner cuando ya no lo necesitamos.
```

En este ejemplo, hemos utilizado **Scanner.nextInt()** para leer un número entero del usuario. Si necesitas leer otro tipo de dato, como un número decimal o una cadena de texto, puedes usar los métodos correspondientes de **Scanner (nextDouble(), nextLine(), etc.)**.

11.1.1. Métodos usados para la entrada desde teclado

A continuación, te presento una tabla con algunos de los métodos más comunes de la clase **Scanner** en Java para leer diferentes tipos de datos desde el teclado:

Método	Descripción
<code>'next()'</code>	Lee y devuelve la siguiente palabra (hasta el espacio en blanco) como una cadena de texto.
<code>'nextInt()'</code>	Lee y devuelve el siguiente número entero como un valor de tipo <code>'int'</code> .
<code>'nextLong()'</code>	Lee y devuelve el siguiente número entero largo como un valor de tipo <code>'long'</code> .
<code>'nextFloat()'</code>	Lee y devuelve el siguiente número de punto flotante como un valor de tipo <code>'float'</code> .
<code>'nextDouble()'</code>	Lee y devuelve el siguiente número de punto flotante como un valor de tipo <code>'double'</code> .
<code>'nextBoolean()'</code>	Lee y devuelve el siguiente valor booleano (true o false) como un valor de tipo <code>'boolean'</code> .
<code>'nextLine()'</code>	Lee y devuelve la siguiente línea completa (incluido el espacio en blanco) como una cadena de texto.
<code>'nextByte()'</code>	Lee y devuelve el siguiente número entero como un valor de tipo <code>'byte'</code> .
<code>'nextShort()'</code>	Lee y devuelve el siguiente número entero como un valor de tipo <code>'short'</code> .
<code>'nextPattern(String pat)'</code>	Lee y devuelve el siguiente token que coincide con el patrón proporcionado como una expresión regular.
<code>'hasNext()'</code>	Verifica si hay más elementos disponibles para leer desde la entrada. Devuelve <code>'true'</code> si hay más, <code>'false'</code> si no.
<code>'hasNextX()'</code>	(Ejemplo: <code>'hasNextInt()'</code> , <code>'hasNextDouble()'</code> , etc.) Verifica si el siguiente token es del tipo X.

IMPORTANTE: Al leer un número desde teclado, Java lee solamente los caracteres numéricos, dejando el carácter de salto de línea (INTRO) en el buffer de memoria del teclado. Esto puede generar un problema porque si justo después de haber leído un número quiero leer una cadena (String) lo que estaré leyendo será el carácter de salto de línea que quedaba en el buffer del teclado. Por tanto, tendré la sensación de que no he leído nada. Para evitar este “error”, si voy a leer un String tras haber leído un número debo ejecutar el método **nextLine()**, de forma que habré dejado vacío (purgado) el buffer de teclado.

11.2. Salida a pantalla

Una opción para mostrar resultados o mensajes en la pantalla es utilizar el método **System.out.println()** que imprime una línea de texto seguida de un salto de línea. Aquí hay un ejemplo de cómo imprimir un mensaje simple en la pantalla:

Ejemplo:

```
System.out.println("¡Hola, bienvenido al Ciclo Formativo de Grado Superior de Desarrollo de Aplicaciones Multiplataforma!");
System.out.println("Este es un ejemplo de salida a pantalla en Java.");
```

A continuación, te presento una tabla con algunos de los métodos más utilizados de la clase **System.out** en Java para volcar datos a la pantalla:

Método	Descripción
<code>'print()'</code>	Imprime un valor en la pantalla sin agregar un salto de línea al final.
<code>'println()'</code>	Imprime un valor en la pantalla y agrega un salto de línea al final.
<code>'printf(String format, Object... args)'</code>	Permite imprimir una cadena de formato y reemplazar los marcadores de posición con los valores proporcionados.
<code>'format(String format, Object... args)'</code>	Similar a <code>'printf()'</code> , permite formatear una cadena y reemplazar los marcadores de posición con valores.

Ejemplo de uso de algunos de estos métodos:

```
int edad = 30;
double altura = 1.75;
String nombre = "Juan";

// Utilizando print() y println() para imprimir en pantalla
System.out.print("Nombre: ");
System.out.println(nombre);
System.out.print("Edad: ");
System.out.println(edad);
System.out.print("Altura: ");
System.out.println(altura);

// Utilizando printf() para formatear la salida
System.out.printf("Nombre: %s, Edad: %d, Altura: %.2f", nombre, edad, altura);
```

PREGUNTA: ¿Cuál será la salida del programa del ejemplo?

En este ejemplo, hemos utilizado diferentes métodos para imprimir información en la pantalla. `print()` y `println()` se utilizan para imprimir valores sin y con salto de línea, respectivamente. `printf()` se utiliza para imprimir valores formateados, donde `%s`, `%d`, y `%.2f` son marcadores de posición que serán reemplazados por los valores de las variables proporcionadas.

Recuerda que la combinación adecuada de estos métodos te permitirá mostrar información de manera organizada y legible para los usuarios de tus aplicaciones Java.

11.2.1. Carácteres de formato para la salida a pantalla

A continuación, te presento una tabla con algunos de los caracteres de formato más comunes utilizados con el método `printf()` en Java, junto con su significado:

Carácter de Formato	Significado
`%s`	Se utiliza para formatear una cadena de texto.
`%d`	Se utiliza para formatear un número entero con signo.
`%f`	Se utiliza para formatear un número de punto flotante (decimal).
`%.nf`	Donde 'n' es el número de dígitos decimales deseado, formatea un número de punto flotante con 'n' decimales.
`%c`	Se utiliza para formatear un carácter.
`%b`	Se utiliza para formatear un valor booleano ("true" o "false").
`%x`	Se utiliza para formatear un número entero en hexadecimal (base 16).
`%o`	Se utiliza para formatear un número entero en octal (base 8).
`%e`	Se utiliza para formatear un número de punto flotante en notación científica.
`%t`	Se utiliza para formatear una fecha/hora utilizando argumentos de tiempo.

Ejemplo:

```
String nombre = "María";
int edad = 25;
double altura = 1.68;
char genero = 'F';
boolean esEstudiante = true;

System.out.printf("Nombre: %s, Edad: %d, Altura: %.2f, Género: %c, Es estudiante: %b", nombre, edad, altura, genero, esEstudiante);
```

PREGUNTA: ¿Cuál será la salida del programa del ejemplo?

Con estos ejemplos, deberías tener una base sólida para realizar la entrada desde teclado y mostrar resultados en pantalla utilizando la entrada/salida estándar en Java. Estas técnicas son esenciales para el desarrollo de aplicaciones interactivas y para obtener datos de los usuarios.

12. Despedida



¡Continúa practicando y explorando más funcionalidades del lenguaje Java para mejorar tus habilidades de programación!