UNIDAD DE TRABAJO 2

# ELEMENTOS DE UN PROGRAMA INFORMÁTICO



## 1. BLOQUES FUNDAMENTALES I

### 1.1. MÉTODOS FRENTE A FUNCIONES Y PROCEDIMIENTOS I

#### 1.1.1. CONCEPTOS BÁSICOS I

- 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.

# 1. BLOQUES FUNDAMENTALES II

## 1.1. MÉTODOS FRENTE A FUNCIONES Y PROCEDIMIENTOS II

### 1.1.1. CONCEPTOS BÁSICOS II

- 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:

```java
// 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?



# 1. BLOQUES FUNDAMENTALES III

## 1.1. MÉTODOS FRENTE A FUNCIONES Y PROCEDIMIENTOS III

### 1.1.1. CONCEPTOS BÁSICOS III

- 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:

```java
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. BLOQUES FUNDAMENTALES IV

## 1.1. MÉTODOS FRENTE A FUNCIONES Y PROCEDIMIENTOS IV
## 1.1.2. PASO DE PARÁMETROS I

- 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, existen dos opciones de paso de parámetros:
  - **por valor** para *tipos primitivos* (tipos simples).
  - **por referencia** para *tipos de referencia* (objetos).

- A continuación, veremos estas dos opciones así como ejemplos de cada una de ellas:



# 1. BLOQUES FUNDAMENTALES V

## 1.1. MÉTODOS FRENTE A FUNCIONES Y PROCEDIMIENTOS V
## 1.1.2. PASO DE PARÁMETROS II

- **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);
```

# 1. BLOQUES FUNDAMENTALES VI

## 1.1. MÉTODOS FRENTE A FUNCIONES Y PROCEDIMIENTOS VI
## 1.1.2. PASO DE PARÁMETROS III

- **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. BLOQUES FUNDAMENTALES VII

## 1.1. MÉTODOS FRENTE A FUNCIONES Y PROCEDIMIENTOS VII
## 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 quizá 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. BLOQUES FUNDAMENTALES VIII

## 1.1. MÉTODOS FRENTE A FUNCIONES Y PROCEDIMIENTOS VIII
## 1.1.4. SOBRECARGA DE FUNCIONES I

✧ *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**.



# 1. BLOQUES FUNDAMENTALES IX

## 1.1. MÉTODOS FRENTE A FUNCIONES Y PROCEDIMIENTOS IX
## 1.1.4. SOBRECARGA DE FUNCIONES II

✧ **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);
```

***SE PROPONE LA REALIZACIÓN DE LA TAREA 1***

# 1. BLOQUES FUNDAMENTALES X

## 1.1. MÉTODOS FRENTE A FUNCIONES Y PROCEDIMIENTOS X
## 1.1.5. RECURSIVIDAD I

- 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.



# 1. BLOQUES FUNDAMENTALES XI

## 1.1. MÉTODOS FRENTE A FUNCIONES Y PROCEDIMIENTOS XI
## 1.1.5. RECURSIVIDAD II

- **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);
```

# 1. BLOQUES FUNDAMENTALES XII

## 1.1. MÉTODOS FRENTE A FUNCIONES Y PROCEDIMIENTOS XII
## 1.1.5. RECURSIVIDAD III

- 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?

***SE PROPONE LA REALIZACIÓN DE LA TAREA 2***



# 1. BLOQUES FUNDAMENTALES XIII

## 1.1. MÉTODOS FRENTE A FUNCIONES Y PROCEDIMIENTOS XIII
## 1.1.6. MÉTODOS I

- 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.

# 1. BLOQUES FUNDAMENTALES XIV

## 1.1. MÉTODOS FRENTE A FUNCIONES Y PROCEDIMIENTOS XIV
## 1.1.6. MÉTODOS II

✧ El significado de 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**.



# 1. BLOQUES FUNDAMENTALES XV

## 1.1. MÉTODOS FRENTE A FUNCIONES Y PROCEDIMIENTOS XV
## 1.1.6. MÉTODOS III

- **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.

# 1. BLOQUES FUNDAMENTALES XVI

## 1.1. MÉTODOS FRENTE A FUNCIONES Y PROCEDIMIENTOS XVI
## 1.1.6. MÉTODOS IV

- **modificador de NO acceso**: Puede ser **static, final, abstract,** u otros, dependiendo de la funcionalidad del método. (continuación …)

  - **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.



# 1. BLOQUES FUNDAMENTALES XVII

## 1.1. MÉTODOS FRENTE A FUNCIONES Y PROCEDIMIENTOS XVII
## 1.1.6. MÉTODOS V

- **modificador de NO acceso**: Puede ser **static, final, abstract,** u otros, dependiendo de la funcionalidad del método. (continuación …)

  - **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.

# 1. BLOQUES FUNDAMENTALES XVIII

## 1.1. MÉTODOS FRENTE A FUNCIONES Y PROCEDIMIENTOS XVIII
## 1.1.6. MÉTODOS VI

- **modificador de NO acceso**: Puede ser **static, final, abstract,** u otros, dependiendo de la funcionalidad del método. (continuación …)

  - **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.



# 1. BLOQUES FUNDAMENTALES XIX

## 1.1. MÉTODOS FRENTE A FUNCIONES Y PROCEDIMIENTOS XIX
## 1.1.6. MÉTODOS VII

- **modificador de NO acceso**: Puede ser **static, final, abstract,** u otros, dependiendo de la funcionalidad del método. (continuación …)

  - **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.

# 1. BLOQUES FUNDAMENTALES XX

## 1.1. MÉTODOS FRENTE A FUNCIONES Y PROCEDIMIENTOS XX
## 1.1.6. MÉTODOS VIII

- **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.



# 1. BLOQUES FUNDAMENTALES XXI

## 1.1. MÉTODOS FRENTE A FUNCIONES Y PROCEDIMIENTOS XXI
## 1.1.6. MÉTODOS IX

✧ **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?

# 1. BLOQUES FUNDAMENTALES XXII

## 1.1. MÉTODOS FRENTE A FUNCIONES Y PROCEDIMIENTOS XXII
## 1.1.6. MÉTODOS X

✧ **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.
    }
}
```



# 1. BLOQUES FUNDAMENTALES XXIII

## 1.1. MÉTODOS FRENTE A FUNCIONES Y PROCEDIMIENTOS XXIII
## 1.1.6. MÉTODOS XI

✧ **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);
    }
}
```

# 1. BLOQUES FUNDAMENTALES XXIV

## 1.1. MÉTODOS FRENTE A FUNCIONES Y PROCEDIMIENTOS XXIV
## 1.1.6. MÉTODOS XII

- 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.



# 1. BLOQUES FUNDAMENTALES XXV

## 1.1. MÉTODOS FRENTE A FUNCIONES Y PROCEDIMIENTOS XXV
## 1.1.6. MÉTODOS XIII

- 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");

# 1. BLOQUES FUNDAMENTALES XXVI

## 1.1. MÉTODOS FRENTE A FUNCIONES Y PROCEDIMIENTOS XXVI

### 1.1.6. MÉTODOS XIV

- 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 declara un método como final, pero eso lo trataremos cuando hablemos de herencia.



# 1. BLOQUES FUNDAMENTALES XXVII

## 1.2. ÁMBITO Y TIEMPO DE VIDA DE LAS VARIABLES I

- 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.

## 1. BLOQUES FUNDAMENTALES XXVIII

### 1.2. ÁMBITO Y TIEMPO DE VIDA DE LAS VARIABLES II

- 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.**



## 1. BLOQUES FUNDAMENTALES XXIX

### 1.2. ÁMBITO Y TIEMPO DE VIDA DE LAS VARIABLES III

- 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 uno 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 anidado, pero lo contrario no es cierto, es decir, los objetos y variables declarados dentro del ámbito anidad no son visibles desde fuera.

# 1. BLOQUES FUNDAMENTALES XXX

## 1.2. ÁMBITO Y TIEMPO DE VIDA DE LAS VARIABLES IV

✧ 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);
  }
}
```

¿Qué muestra este código en la pantalla?

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



# 1. BLOQUES FUNDAMENTALES XXXI

## 1.2. ÁMBITO Y TIEMPO DE VIDA DE LAS VARIABLES V

✧ 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.

# 1. BLOQUES FUNDAMENTALES XXXII

## 1.2. ÁMBITO Y TIEMPO DE VIDA DE LAS VARIABLES VI

- Más consideraciones en cuanto a las variables en cuanto a su tiempo de vida: (continuación)
  - 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.



# 1. BLOQUES FUNDAMENTALES XXXIII

## 1.2. ÁMBITO Y TIEMPO DE VIDA DE LAS VARIABLES VII

- 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

## 1. BLOQUES FUNDAMENTALES XXXIV

### 1.2. ÁMBITO Y TIEMPO DE VIDA DE LAS VARIABLES VIII

- 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;
    {                  // creates a new scope
      int bar = 2;  // Compile-time error - bar already defined!
    }
  }
}
```

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



## 1. BLOQUES FUNDAMENTALES XXXV

### 1.3. CLASE PRINCIPAL Y MÉTODO main() I

- 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
    }
}
```

# 1. BLOQUES FUNDAMENTALES XXXVI

## 1.3. CLASE PRINCIPAL Y MÉTODO main() II

- **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.



# 1. BLOQUES FUNDAMENTALES XXXVII

## 1.3. CLASE PRINCIPAL Y MÉTODO main() III

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

Como puedes ver en el ejemplo 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.

```
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);
    }
}
```

# 1. BLOQUES FUNDAMENTALES XXXVIII

## 1.3. CLASE PRINCIPAL Y MÉTODO main() IV

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

```java
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]);
        }
    }
}
```



# 1. BLOQUES FUNDAMENTALES XXXIX

## 1.3. CLASE PRINCIPAL Y MÉTODO main() V

✧ Para ejecutar el programa desde la consola y pasarle argumentos lo haríamos 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**.

## 1. BLOQUES FUNDAMENTALES XXXX

### 1.3. CLASE PRINCIPAL Y MÉTODO main() VI

- **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.



## 1. BLOQUES FUNDAMENTALES XXXXI

### 1.3. CLASE PRINCIPAL Y MÉTODO main() VII

- 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.

# 1. BLOQUES FUNDAMENTALES XXXXII

## 1.3. CLASE PRINCIPAL Y MÉTODO main() VIII

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

- **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.

***SE PROPONE LA REALIZACIÓN DE LA TAREA 3***

✧ 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 I

✧ 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;
```

## 2. IDENTIFICADORES II

### 2.1. NOTACIÓN "CamelCase" I

- 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:

- **UpperCamelCase o PascalCase:** La primera letra de cada palabra se escribe en mayúscula.
  - **Ejemplos:** `NombreCompleto`, `EdadPersona`, `ClaseJava`.

- **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`.



## 2. IDENTIFICADORES III

### 2.1. NOTACIÓN "CamelCase" II

- 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.

- **RECOMENDACIÓN:** Usar la notación UpperCamelCase para los nombres de las clases y usar la notación lowerCamelCase para el resto de elementos del lenguaje Java.

## 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
```

***SE PROPONE LA REALIZACIÓN DE LA TAREA 4***



## 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** (darle valor) en una sola línea:

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

# 5. TIPOS DE DATOS I

- 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 DE DATOS PRIMITIVOS I

- 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. TIPOS DE DATOS II

## 5.1. TIPOS DE DATOS PRIMITIVOS II

- 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.

  - **byte:** Es un tipo de *dato entero que almacena valores en un rango de -128 a 127*. **Ejemplo:** `byte edad = 25;`
  - **short:** Es un tipo de *dato entero que almacena valores en un rango de -32,768 a 32,767*. **Ejemplo:** `short poblacion = 20000;`
  - **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. TIPOS DE DATOS III

## 5.1. TIPOS DE DATOS PRIMITIVOS III

- **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.
```

- **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.
```

- **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. TIPOS DE DATOS IV

## 5.1. TIPOS DE DATOS PRIMITIVOS IV

- **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';
```

- **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;
```

✧ Como hemos comentado antes, estos dos últimos tipos de datos primitivos no se usan para almacenar valores numéricos. ☺

# 5. TIPOS DE DATOS V

## 5.2. TIPOS DE REFERENCIA I

- *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.



# 5. TIPOS DE DATOS VI

## 5.2. TIPOS DE REFERENCIA II

- A continuación, vamos a ver dos ejemplos:
  - **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
```
  - **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);
```

***SE PROPONE LA REALIZACIÓN DE LA TAREA 5***

# 6. LITERALES I

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

## 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. LITERALES II

## 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 CARATERES

✧ 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. LITERALES III

## 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. LITERALES IV

## 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
```

***SE PROPONE LA REALIZACIÓN DE LA TAREA 6***

## 7. CONSTANTES I

- 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:**

```java
// 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. CONSTANTES II

### 7.1. NOTACIÓN SCREAMING_SNAKE_CASE I

- 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:

  - Ver **ejemplo** de en la siguiente diapositiva.

# 7. CONSTANTES III

## 7.1. NOTACIÓN SCREAMING_SNAKE_CASE II

- **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 I

- 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 I

### 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*.

***SE PROPONE LA REALIZACIÓN DE LA TAREA 7***

## 8. OPERADORES Y EXPRESIONES II

### 8.1. OPERADORES BINARIOS II
### 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. OPERADORES Y EXPRESIONES III

### 8.1. OPERADORES BINARIOS III
### 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”.

***SE PROPONE LA REALIZACIÓN DE LA TAREA 8***

# 8. OPERADORES Y EXPRESIONES IV

## 8.1. OPERADORES BINARIOS IV
## 8.1.4. OPERADORES ESPECIALES I

```java
// 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.
```

***SE PROPONE LA REALIZACIÓN DE LA TAREA 9***



# 8. OPERADORES Y EXPRESIONES V

## 8.1. OPERADORES BINARIOS V
## 8.1.4. OPERADORES ESPECIALES II

```java
// 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. OPERADORES Y EXPRESIONES VI

## 8.2. OPERADORES UNARIOS

```
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.
```

***SE PROPONE LA REALIZACIÓN DE LA TAREA 10***



# 8. OPERADORES Y EXPRESIONES VII

## 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: 11111010 (-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

**NOTA:** Para más información, en los apuntes de formato extendido hay unas **tablas resumen** con el significado de cada operador.

# 8. OPERADORES Y EXPRESIONES VIII

## 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.

***SE PROPONE LA REALIZACIÓN DE LA TAREA 11***

- 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 I

- 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
```

## 9. CONVERSIONES DE TIPO II

- Hay dos tipos de conversiones: (continuación …)

- **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. CONVERSIONES DE TIPO III

### 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;
```

***SE PROPONE LA REALIZACIÓN DE LA TAREA 12***

# 9. CONVERSIONES DE TIPO IV

## 9.1. RELACIÓN DE CONVERSIONES EXPLÍCITAS

```java
// 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;
```

***Pérdida de información:*** 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.

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

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

***SE PROPONE LA REALIZACIÓN DE LA TAREA 13***



# 10. COMENTARIOS I

✧ 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:**

```java
// 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. COMENTARIOS II

### 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. COMENTARIOS III

### 10.3. COMENTARIO DE DOCUMENTACIÓN (Javadoc) I

- 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.

- 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. COMENTARIOS IV

## 10.3. COMENTARIO DE DOCUMENTACIÓN (Javadoc) II

✧ **Ejemplo:**

```java
/**
 * 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);
    }
}
```



# 10. COMENTARIOS V

## 10.3. COMENTARIO DE DOCUMENTACIÓN (Javadoc) III
## 10.3.1. OPCIONES DE Javadoc MÁS UTILIZADAS

✧ A la derecha, te presento una tabla con algunas de las opciones de parametrización más comunes que se pueden utilizar en Javadoc, junto con sus significados.

✧ 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.

***SE PROPONE LA REALIZACIÓN DE LA TAREA 14***

| 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` |

# 11. ENTRADA/SALIDA ESTÁNDAR I

✧ 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*.

## 11.1. ENTRADA DESDE TECLADO I

✧ Para recibir datos desde el teclado, utilizaremos la clase **Scanner** de Java (debemos importar la clase en nuestro programa para poder usarla). Scanner proporciona métodos para leer diferentes tipos de datos de manera sencilla.

- **Ejemplo:**

```java
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.
```

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. ENTRADA/SALIDA ESTÁNDAR II

## 11.1. ENTRADA DESDE TECLADO II

### 11.1.1. MÉTODOS USADOS PARA LA ENTRADA DESDE TECLADO

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

| Método | Descripción |
|---|---|
| `next()` | Lee y devuelve la siguiente palabra (hasta el espacio en blanco) como una cadena de texto. |
| `nextInt()` | Lee y devuelve el siguiente número entero como un valor de tipo `int`. |
| `nextLong()` | Lee y devuelve el siguiente número entero largo como un valor de tipo `long`. |
| `nextFloat()` | Lee y devuelve el siguiente número de punto flotante como un valor de tipo `float`. |
| `nextDouble()` | Lee y devuelve el siguiente número de punto flotante como un valor de tipo `double`. |
| `nextBoolean()` | Lee y devuelve el siguiente valor booleano (true o false) como un valor de tipo `boolean`. |

| Método | Descripción |
|---|---|
| `nextLine()` | Lee y devuelve la siguiente línea completa (incluido el espacio en blanco) como una cadena de texto. |
| `nextByte()` | Lee y devuelve el siguiente número entero como un valor de tipo `byte`. |
| `nextShort()` | Lee y devuelve el siguiente número entero como un valor de tipo `short`. |
| `nextPattern(String pat)` | Lee y devuelve el siguiente token que coincida con el patrón proporcionado como una expresión regular. |
| `hasNext()` | Verifica si hay más elementos disponibles para leer desde la entrada. Devuelve `true` si hay más, `false` si no. |
| `hasNextX()` | (Ejemplo: `hasNextInt()`, `hasNextDouble()`, etc.) Verifica si el siguiente token es del tipo X. |

✧ **IMPORTANTE:** Cuando vayas a leer una cadena (String) tras haber leído un número, debes ejecutar el método **nextLine()** para purgar el buffer de teclado.

# 11. ENTRADA/SALIDA ESTÁNDAR III

## 11.2. SALIDA A PANTALLA I

✧ 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.

- **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.");
```



# 11. ENTRADA/SALIDA ESTÁNDAR IV

## 11.2. SALIDA A PANTALLA II

✧ 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 |
|---|---|
| `print()` | Imprime un valor en la pantalla sin agregar un salto de línea al final. |
| `println()` | Imprime un valor en la pantalla y agrega un salto de línea al final. |
| `printf(String format, Object... args)` | Permite imprimir una cadena de formato y reemplazar los marcadores de posición con los valores proporcionados. |
| `format(String format, Object... args)` | Similar a `printf()`, permite formatear una cadena y reemplazar los marcadores de posición con valores. |

# 11. ENTRADA/SALIDA ESTÁNDAR V

## 11.2. SALIDA A PANTALLA III

✧ **Ejemplo:**

```java
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?**



# 11. ENTRADA/SALIDA ESTÁNDAR VI

## 11.2. SALIDA A PANTALLA IV

### 11.2.1. CARACTERES DE FORMATO PARA LA SALIDA A PANTALLA I

✧ A la derecha, te presento una tabla con algunos de los caracteres de formato más comunes utilizados con el método **printf()**, 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. |

# 11. ENTRADA/SALIDA ESTÁNDAR VII

## 11.2. SALIDA A PANTALLA V

## 11.2.1. CARACTERES DE FORMATO PARA LA SALIDA A PANTALLA II

✧ **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?**

***SE PROPONE LA REALIZACIÓN DE LA TAREA 15***



# 12. DESPEDIDA

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

***SE PROPONE LA REALIZACIÓN DE LA TAREA 16 (REPASO)***