# Programación estructurada

# –

# Java

gamma tech school

# ÍNDICE

# 1. ¿Qué es Java?

Java es un lenguaje de programación **orientado a objetos, multiplataforma** y **de propósito general**, diseñado para ser robusto, seguro y eficiente. Se creó en 1995 y se ha convertido en uno de los lenguajes más utilizados en el desarrollo de aplicaciones de escritorio, web, móviles y sistemas empresariales.

- **Multiplataforma:** “Escribe una vez, ejecuta en cualquier lugar” gracias a la **Máquina Virtual de Java (JVM).**
- **Orientado a objetos (OOP):** facilita la reutilización y mantenimiento del código.
- **Tipado estático:** detecta errores antes de ejecutar el programa.


# 1. ¿Qué es Java?

Java es un lenguaje de programación compilado. Eso quiere decir que lo que programamos es de *alto nivel* (cercano a nuestro idioma) pero la máquina no lo entiende. Para que lo entienda necesitamos compilar el código.

Además distingue entre mayúsculas y minúsculas. Debes tener cuidado porque <<Buenos días>> será diferente a <<buenos días>>, hay que tener cuidado o no funcionará nada.


# 1.2. Las clases

Vamos a hacer un sencillo programa que únicamente mostrará un mensaje de texto.

Cada **clase** en java necesita un fichero y debe cumplir con una serie de consideraciones:

- El nombre del fichero y la clase deben ser iguales.
- El fichero que se va a crear tendrá la extensión `.java`

Dentro del fichero, para poder crear la clase hay que escribir `public class PrimeraClase {}`. Dejando un resultado parecido a este:

```
1   public class PrimeraClase {
2       // bloque de código
3   }
```

¿Qué significa cada parte del código?

- **`public`**: palabra reservada que no puede ser utilizada para nombrar métodos, variables, clases. Significa que lo que vas a crear va a poder ser accesible <<desde fuera>>. En los ficheros `.java` solo puede haber una clase pública.
- **`class`**: palabra reservada. Indica que se va a declarar una clase. En Java todo el código necesita estar dentro de una clase.
- **`PrimeraClase:`** nombre de la clase. Se le puede dar cualquier nombre pero siempre es mejor utilizar nombres que sean descriptivos. La clase se escribe en **PascalCase** siendo la primera letra de cada palabra en mayúscula, incluida la primera.
- **`{}:`** las llaves van a indicar dónde empieza y termina un bloque de código. Todo el código de la clase irá entre la llave de apertura y cierre.

# 1.3. El método main

El método `main` es un método especial. Más adelante aprenderemos más cosas sobre los métodos. Este, de momento, lo necesitamos para hacer cualquier cosa.

Es el método que Java va a buscar para empezar la ejecución de las clases. Para que Java lo reconozca tiene que estar escrito correctamente:

```
public static void main(String[] args) {
    // bloque de código
}
```

¿Qué significa cada parte del código?

- **`public`**: palabra reservada con el mismo significado pero aplicado a un método. Esto indica que puede ser llamado desde otras clases.
- **`static`**: palabra reservada. Obligatoria para el método `main`. Indica que no se necesita un objeto para poder utilizar este método.
- **`void`**: palabra reservada. Se escribe antes del nombre del método, posición en la que se indica el tipo de retorno. El método `main` siempre será `void`.
- **`main`**: nombre del método. A un método se le puede poner cualquier nombre, pero el método `main` debe llamarse `main`. El nombre de un método se escribe en **camelCase**, cada palabra se escribe con la primera letra en mayúsculas, salvo la primera.
- **`()`**: los paréntesis envuelven, cuando estamos declarando un método, los diferentes argumentos que va a recibir.
- **`String`**: nombre de una clase (fíjate que empieza en mayúscula) que representa los textos en Java. Los argumentos del método `main` son textos.
- **`args`**: nombre que recibe el argumento. En el método `main` es una secuencia de cadenas de texto.
- **`{}`**: de nuevo las llaves. Como en la clase, indica dónde empieza y termina el bloque de código.

# 1.4. Mostrar mensajes

Ya tenemos una clase y el método `main`, la estructura básica para ejecutar un código en Java. Podemos hacer un programa y que muestre una salida con un mensaje.

```
System.out.println(x:"Mensaje de texto");
```

¿Qué significa cada parte del código?

- **`System.out.println`**: ahora mismo es pronto para explicarlo, pero necesario para poder ver la salida por la consola.
- **`()`**: dentro de los paréntesis ponemos los argumentos que le vamos a pasar al método.
- **`"Mensaje de texto"`**: valor real de lo que queremos ver por pantalla.
- **`;`**: se utiliza al final de cada sentencia de Java. Si no lo ponemos el compilador no lo entenderá y nos dará errores.

```java
public class PrimeraClase { // inicio del bloque de clase

    // aunque las llaves indican principio y fin de cada bloque
    // se utilizan cuatro espacios para facilitar la lectura

    Run main | Debug main | Run | Debug
    public static void main(String[] args) { // inicio del bloque main
        System.out.println(x:"Mensaje de texto");
    } // cierre del bloque main
} // cierre del bloque de clase

// las llaves de cierre se colocan al mismo nivel
```

# 1.5. Compilación (I)

Utilizando un método de desarrollo Java se puede ejecutar el código pulsando unos botones. Si quieres ser un buen programador deberías tener ciertas nociones sobre cómo hacer las cosas por ti mismo y entender cómo funcionan.

Para ejecutar una clase de java (`NombreClase.java`) desde la línea de comandos hay que seguir dos pasos:

1. **Compilar el código**
   El compilador `javac` es una herramienta que procesa los ficheros que hemos escrito y, si son correctos, genera ficheros legibles por el ordenador.
   Los ficheros hechos por nosotros son `*.java`
   `$ javac NombreClase.java`
2. **Ejecutar el programa**
   Si todo ha ido bien se generará un fichero `.class`. Este fichero es el que debemos ejecutar.
   `$ java NombreClase`

## Código

```java
public class PrimeraClase {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        System.out.println(x:"Mensaje de texto");
    }
}
```

## Línea de comandos

```
% javac PrimeraClase.java
% java PrimeraClase
```

Si se ha hecho correctamente cuando usemos el compilador no tendremos ningún mensaje de error.
Después de ejecutar el programa con la instrucción `java` vamos a ver por pantalla “Mensaje de texto”.

# 1.5. Compilación (II)

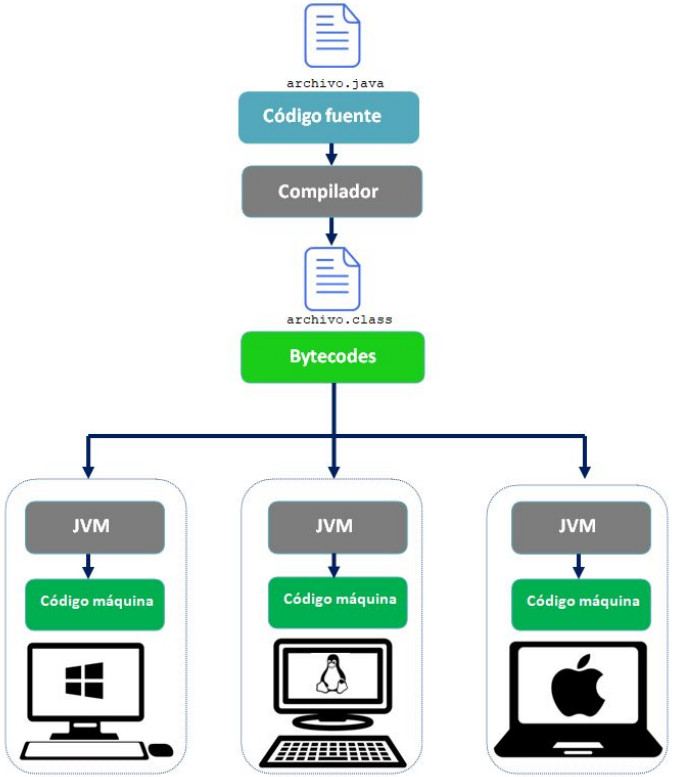
- Java es un lenguaje de programación compilado.
- Antes de su ejecución, el código fuente se convierte en **bytecode** mediante el compilador.
- Este bytecode es interpretado por la **Máquina Virtual de Java (JVM)**, lo que permite la ejecución del programa en diferentes plataformas.


JVM

(Java Virtual Machine)

# 1.5. Compilación (III)

1. **Escribir el Código Fuente:**
   - Crear un archivo con extensión `.java` que contiene el código fuente de la aplicación.
2. **Compilar el Código:**
   - Utilizar el compilador de Java (`javac`) para traducir el código fuente en bytecode, generando un archivo con extensión `.class`.
3. **Ejecutar el Programa:**
   - La JVM interpreta y ejecuta el bytecode contenido en el archivo `.class`.



# 1.5. Compilación (IV)

**Compilar:**

- En la terminal, navega al directorio que contiene el archivo `.java` y ejecuta:
  - `$ javac Archivo.java`

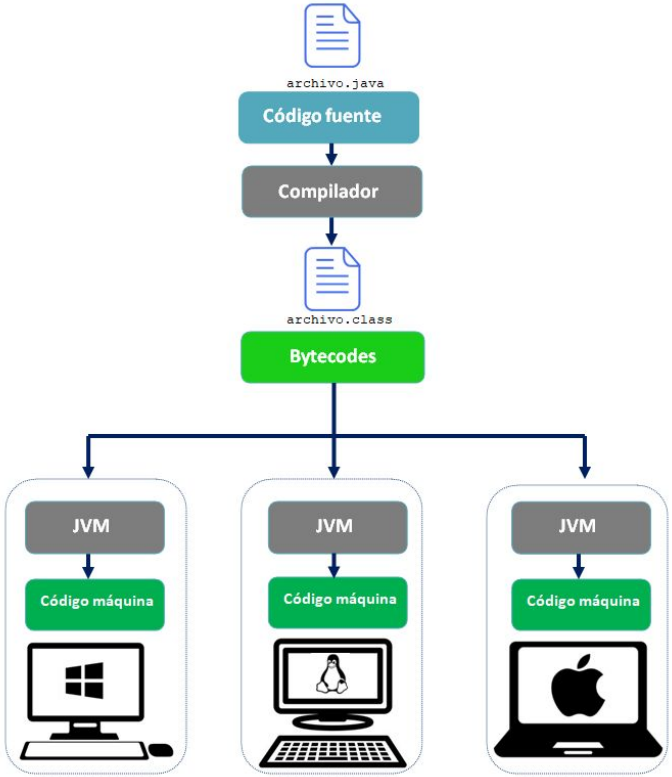
**Ejecutar:**

- Una vez compilado sin errores, ejecuta el programa con:
  - `$ java NombreClase`

**Nota:**

- Si hay errores durante la compilación, se debe corregir el código y volver a compilar.

* El símbolo `$` indica que es una instrucción para la terminal.



# 2. Argumentos, variables, métodos, operadores

Argumentos, variables, métodos y operadores son cuatro conceptos **fundamentales** en programación.

¿Pero qué son?

- **Argumentos:** información que recibe el método.
- **Variables:** en ellas almacenamos los datos.
- **Métodos:** bloques de código a los que se les pone nombre, que desempeñan alguna tarea, pudiendo ser llamados en cualquier parte del código.
- **Operadores:** nos permiten hacer operaciones sobre las variables.

Antes que nada veamos cómo hay que escribir correctamente el código:

Java™

# 2.1. Argumentos

```
public static void main(String[] args) {
    // bloque de código
}
```

El método `main` recibe como argumentos (`String[] args`) que es una secuencia de textos.
Hoy en día apenas se usan `arrays` (o secuencias) porque tenemos otras estructuras más flexibles, las colecciones. Pero nos viene bien para ir aprendiendo.
Cuando se declara el método se indican los parámetros con su tipo/clase y el nombre. En este caso el parámetro es `String` con el nombre `args`. Al ser una secuencia ponemos acceder a cada elemento, según su posición.

- Podemos ver la longitud, cantidad de elementos que hay, con el atributo `length`.
- La posición de cada elemento es según su índice. Es importante tener en cuenta que los índices empiezan en `0`, es decir, el primer elemento de la secuencia su índice es `0`.
- Los índices siempre van a continuación del nombre de la secuencia entre corchetes [ ]

```java
public class PrimeraClase {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        String primerElemento = args[0];  // esto es una variable
        String segundoElemento = args[1]; // almacena el valor de ese elemento
        String tercerElemento = args[2];

        // ¿cuántos elementos tiene args?
        int cantidadElementos = args.length;

        System.out.println("El primer elemento es: " + primerElemento);
        System.out.println("El segundo elemento es: " + segundoElemento);
        System.out.println("El tercer elemento es: " + tercerElemento);
        System.out.println("En total tengo " + cantidadElementos + " elementos");
    }
}
```

# 2.2.Variables

## Variables

Las variables se pueden entender como cajas en las que podemos guardar cosas, que son diferentes tipos de datos.

Hay diferentes maneras de declarar una variable:

- Declarándola primero y asignándole valor después:
  ```
  tipo nombreVariable;
  nombreVariable = valor;
  ```

- Declarándola con valor en una misma sentencia:
  ```
  tipo nombreVariable = valor;
  ```

Las variables siempre se escriben en **camelCase**, igual que los métodos. Cada palabra empieza en mayúscula, salvo la primera.

Para poder ir más a fondo con las variables tenemos que ver los diferentes tipos primitivos que hay en Java.

```java
public class PrimeraClase {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        String primerElemento = args[0];  // esto es una variable
        String segundoElemento = args[1]; // almacena el valor de ese elemento
        String tercerElemento = args[2];

        // ¿cuántos elementos tiene args?
        int cantidadElementos = args.length;

        System.out.println("El primer elemento es: " + primerElemento);
        System.out.println("El segundo elemento es: " + segundoElemento);
        System.out.println("El tercer elemento es: " + tercerElemento);
        System.out.println("En total tengo " + cantidadElementos + " elementos");
    }
}
```

## Alcance

Las llaves **{}** indican dónde empieza y acaba el bloque de código de una clase, método, bucle, condicional…

Una variable únicamente existe dentro del bloque de código comprendido entre unas claves. Conforme vayamos avanzando iréis viendo mejor esta lógica.

# 2.2.1. Variables: tipos primitivos

| Tipo de dato | Tamaño | Descripción | Ejemplo |
|---|---|---|---|
| `byte` | 1 byte | **números** enteros desde -128 a 127 | `byte numMini = 100;` |
| `short` | 2 bytes | **números** enteros desde -32768 a 32767 | `short numPeque = 25000;` |
| `int` | 4 bytes | **números** enteros desde -2147483648 a 2147483647 | `int numNormal = 1000000;` |
| `long` | 8 bytes | **números** enteros desde -9223372036854775808 a 9223372036854775807 | `long numGrande = 12345678900L;` |
| `float` | 4 bytes | **números** con decimales con precisión de 6 a 7 dígitos decimales, desde 3.4e-038 hasta 3.4e+038 | `float numComa = 12.34f;`<br>`float comaCientifica = 67e8f;` |
| `double` | 8 bytes | **números** con decimales con precisión de 15 dígitos decimales, desde 1.7e-308 a 1.7e+308 | `double numDoble = 54.321;`<br>`double dobleCientifica = 12E4d;` |
| `boolean` | 1 bit | valores `true` o `false` | `boolean programarMola = true;` |
| `char` | 2 bytes | almacena una única letra, caracter o valor ASCII | `char letra = 'a';`<br>`char simbolo = '0';` |

`String` empieza por mayúscula porque no es un tipo primitivo, es una clase. Descubriremos en profundidad qué son las clases. Por ahora usad `String` como un tipo de dato complejo.

# 2.2.2. Variables: declaración

## Sintaxis

En Java se tiene que especificar qué tipo de dato va a tener cada variable. Una vez declarada se puede modificar su valor pero **no se puede modificar su tipo**.

```
tipo nombreVariable = valor;
```

- Se indica el **tipo** de dato de la variable.
- Se le da **nombre** utilizando *camelCase*.
- Se utiliza el operador `=` para asignarle un valor.
- Se le asigna el valor del tipo adjudicado.
- Se utiliza `;` al **final** de cada sentencia.

## Palabras reservadas

- No pueden utilizarse las palabras reservadas como nombres de **variables, métodos, clases o identificadores.** Hay distintos grupos de palabras reservadas que se utilizan para:
  - Control de flujo
  - Definición de tipos y estructuras
  - Modificar acceso y ámbito
  - Tipos de datos primitivos
  - Palabras relacionadas con la memoria y referencias
  - Manejo de concurrencia (threads)
  - Definición de paquetes y uso de clases
  - Otras, como pruebas, serialización…

| | | |
|---|---|---|
| | `final` | Define una variable como inmutable |

# 2.2.2. Variables: String vs char

## Particularidades

`String` es un tipo de dato **no primitivo**, es una clase. Es un objeto que permite tener un número variable de caracteres (`char`) en una misma variable, como si fuera un array de caracteres (`char[]`).

Al ser un objeto tiene métodos, que puedes consultar, algunos más habituales, en el apartado 5.3.

Para poder comparar dos cadenas de texto se tendrá que utilizar el método **`.equals()`**

El operador **`+`** permite concatenar cadenas de texto, unir dos cadenas de texto en una nueva.

La diferencia entre **`char`** y **`String`** es que el primero es un tipo primitivo que únicamente puede almacenar **un** carácter que va entre comillas simples **`‘’`** mientras que el segundo es una **`class`** y va entre comillas dobles **`“”`** tenga uno o varios caracteres.

```java
// Declaración de variables
String nombre;
String apellido;
String nombreApellido;

// Inicializando variables asignando valores
nombre = "Carlos";
apellido = "Saiz";
nombreApellido = nombre + apellido; // concatenación de String

boolean esIgual = nombre.equals(apellido); // false

// String vs char
char unicoCaracter = 'a'; // char lleva comillas simples
String unicoCaracterString = "a"; // String lleva comillas dobles
String variosCaracteres = "abc";
```

**Importante:** una **`String`** es **inmutable**. Cualquier método que “modifique” un `String` devuelve una **cadena nueva**.

# 2.2.2.1. Los escapes

## Particularidades

Son secuencias especiales que empiezan con una barra invertida “\”.

Se utilizan para representar caracteres que no pueden escribirse directamente en una cadena de texto o tienen un significado especial.

| Carácter de escape | Significado | Ejemplo en cadena | Salida en consola |
|---|---|---|---|
| `\n` | Salto de línea (line break) | `"Hola\nMundo"` | Hola<br>Mundo |
| `\t` | Tabulación (tab) | `"Hola\tMundo"` | Hola Mundo |
| `\"` | Comilla doble | `"Ella dijo: \"Hola\""` | Ella dijo: "Hola" |
| `\'` | Comilla simple | `"Es un \'éxito\'"` | Es un 'éxito' |
| `\\` | Barra invertida (backslash) | `"C:\\Usuarios\\Juan"` | C:\Usuarios\Juan |
| `\r` | Retorno de carro (carriage return) | `"Hola\rMundo"` | Mundo (sobrepone texto) |
| `\b` | Retroceso (backspace) | `"Hola\bMundo"` | HolMundo (borra una letra) |

# 2.2.3. Variables: buenas prácticas

Las variables son espacios en memoria que almacenan datos que pueden cambiar durante la ejecución del programa. Es importante asignarles nombres descriptivos para facilitar la comprensión del código.

```
// Esto es correcto
int numeroEntero;
double numeroDecimal;

// Esto no es buena práctica
int variable1;
double variable2;
```

# 2.2.3. Variables: buenas prácticas

- Java es un lenguaje de **tipado estático**: cada variable debe declararse con un tipo específico.
    - Si el valor asignado no corresponde a su tipo declarado, no compilará.
- El nombre se escribe en *camelCase*.
- No se recomienda que los nombres de las variables empiecen con el carácter **_** ni **$**
- Evitar los nombres de un solo carácter, salvo para variables temporales.

```java
// No lo hagáis, salvo que sean temporales
int i;
double d;

// No es recomendable
int _numero;
double $numero;

// No compilará
int numeroEntero = 3.5;
char unicoCaracter = "cosa";
```

# 2.2.4. Variables: casting

En Java los diferentes tipos primitivos tienen una jerarquía que dictamina cómo puede convertirse un dato de un tipo a otro. La **jerarquía** se basa en el tamaño y la precisión de cada tipo de dato.

Cambiar de un tipo de **menor** a **mayor** precisión no va a necesitar casteo:

```
byte -> short / char -> int -> long -> float -> double
```

Hacer el cambio de un tipo de dato menor a uno mayor será tan fácil como crear una nueva variable con el tipo mayor y asignarle como valor la variable que tiene el tipo menor:

```
byte datoMenor = 6;
float datoMayor = datoMenor;
System.out.println(datoMayor); // 6.0
```

En cambio, hacer el cambio de un tipo de dato mayor a otro menor va a necesitar casteo, cambiar el tipo de dato.

```
float datoMayor = 6.5;
int datoMenor = datoMayor; // no se puede
int datoMenor = (int) datoMayor; // se cambia float a int
System.out.println(datoMenor); // 6
```

Haciendo este proceso va a haber una pérdida de información. En este caso se pierde toda la parte decimal, quedándonos únicamente con la parte entera del número.

```
int grande = 130;
byte pequenyo = (byte) grande; // OVERFLOW, byte va de -128 a 127
System.out.println(pequenyo); // -126, no da error pero no es correcto

int numGrande = 120;
byte numPequenyo = (byte) numGrande;
System.out.println(numPequenyo); // 120, esto es lo esperado, cabe en el rango

int a = 5;
int b = 2;
double resultado = a / b; // int / int
System.out.println(resultado); // se espera 2.5, pero será 2.0

// uso de casteo
double resultadoCorrecto = (float) a / b; // float / int -> gana float
// double es mayor que float, no necesita casteo
System.out.println(resultadoCorrecto); // 3.5
```

**Overflow**: guardar un número en un tipo que no tiene rango suficiente **no dará error**. Cambia el número de forma extraña porque se desborda y vuelve a empezar dando la vuelta como si se tratara de un reloj.

Ejemplo: `short a = 130;` → lo queremos convertir a `byte b = (byte) a;`

`byte` llega hasta 127. Se desborda y sigue contando desde el inicio.
Con 128 empieza desde el extremo y lo tiene en cuenta como -128. Con 129 pasa al siguiente que es -127 y con 130 se queda en -126.

# 2.3. Métodos: estructura

```
[visibilidad] [tipo de método] [tipo de retorno] [nombreDelMetodo]([parámetros]) {
    // cuerpo del método
}
```

- **Visibilidad**: determina desde dónde se puede usar el método. Ejemplo:
  - `public`: accesible desde cualquier clase.
  - `private`: accesible únicamente desde la clase.
- **Tipo de método**: (*)
  - `static`: pertenece a la clase y no a un objeto concreto.
  - Si no se indica nada es necesario instanciar un objeto.
- **Tipo de retorno**: indica qué devuelve el método.
  - `void`: no devuelve nada.
  - `int`, `double`, …: devuelve un valor de ese tipo.
    - Necesaria la palabra: `return`

(*) En esta primera parte consolidamos las bases de la programación estructurada. En el siguiente bloque, con la programación orientada a objetos, veremos mejor las diferencias.

- **Nombre del método**: debe ser descriptivo y utilizar *camelCase*, ejemplos:
  - `calcularArea`, `mostrarMensaje`
- **Parámetros de entrada**: son las variables que se pasan al método al llamarlo.
  - ningún parámetro: `()`
  - un parámetro: `(int edad)`
  - varios parámetros: `(String nombre, int edad, …)`
- **Llaves `{}`**: indican el cuerpo del método, donde se define lo que va a hacer.

# 2.3. Métodos

## Métodos

Las variables sirven para almacenar datos, así como los métodos (funciones o procedimientos) sirven para almacenar bloques de código que realizan una tarea que podemos llamar utilizando su nombre y evitarnos repetir el mismo código una y otra vez.

Para declarar un método hay que definir cómo se puede acceder a él, si es o no estático, el tipo que tiene (en caso de devolver un resultado), nombre y parámetros (en caso de ser necesarios).

```
private static int suma(int a, int b) {
    return a + b;
}
```

En este caso el método tiene `return` lo que indica que va a devolver un resultado. Por eso tiene también el tipo de dato `int` asignado. Puede ser que haga una tarea sin devolver realmente ningún valor, entonces en el lugar de `int` se indica como `void` como vimos con el método `main`

- **`private`**: como `public` indica la **visibilidad** del método. De momento no tenemos clases así que lo vamos a dejar como privado. Por defecto **haced privado todo lo que podáis**. Únicamente si es necesario declararlo como `public`.
- **`static`**: palabra reservada. Muy pronto para poder hablar de ella. Indica el **tipo de función**.
- **`int`**: tipo de dato que tendrá el resultado del método. Java es muy estricto con los tipos. Si no va a devolver nada se indica como `void`. Se conoce como **parámetro de salida**.
- **`suma`**: nombre del método. El nombre de un método se escribe en **camelCase**, cada palabra se escribe con la primera letra en mayúsculas, salvo la primera.
- **`()`**: los paréntesis envuelven, cuando estamos declarando un método, los **parámetros de entrada** (como si se declararan variables que usará el método internamente).
- **`int x`**: tipo y clase de cada parámetro, cuando se llame al método se indicará sus valores como argumentos.
- **`return`**: palabra clave que indica que el resultado de esa sentencia será lo que devuelva el método.
- **`{}`**: de nuevo las llaves. Como en la clase, indica dónde empieza y termina el bloque de código.

# 2.3.1. Métodos: ejemplo

## Ejemplo de uso: Métodos

Para usar un método simplemente necesitamos llamarlo por su nombre y pasarle los valores a los parámetros. Sigamos con el método declarado:

```
private static int suma(int a, int b) {
    return a + b;
}
```

Una vez declarado el método podemos usarlo dentro del alcance que tiene el método `main`, más adelante veremos más tipos de uso.

```
suma(a:3, b:5);
```

A los parámetros a y b les damos los valores 3 y 5. Y el método `suma` devolverá el resultado de la suma entre ambos números. Pero si necesitamos este resultado más adelante necesitamos guardarlo en una variable.

Así como podemos guardar el resultado en una variable también podemos pasarle variables (siempre que su tipo sea el mismo que el tipo asignado a sus parámetros).

```
int primerNumero = 3;
int segundoNumero = 5;
suma(primerNumero, segundoNumero);
```

Ahora lo que le entra al método `suma( )` es el valor de cada variable y hará la operación con esos valores.

Puedes guardar el resultado en una variable, para ello tienes que indicar el tipo de dato que va a tener la variable; de este modo podrás utilizar el resultado tantas veces como sea necesario.

```
int resultado = suma(primerNumero, segundoNumero);
System.out.println(resultado); // Así lo mostrará por pantalla
```

Completar con archivo “_002_Metodos.java”

# 2.3. Métodos: sobrecarga (overloading)

## Definición

La **sobrecarga** es lo que ocurre cuando se definen varios métodos con el mismo nombre pero que aceptan parámetros diferentes.

### Firma

En Java la firma es lo que permite diferenciar unívocamente los diferentes métodos. Está compuesta por:

1. El **nombre** del método.
2. Los **parámetros de entrada** y/o su tipo.

El tipo de **retorno NO** forma parte de la firma.

## Utilidad

- Permite tener métodos que hagan la misma función con el mismo nombre para diferentes tipos de datos.
- Mejora la legibilidad y la organización del código.
- Evita tener que crear muchos nombres distintos para funciones similares.

```
public static void imprimir(long valor) {
public static void imprimir(String valor) {
public static void imprimir(double valor) {
```

Gracias a la sobrecarga se puede utilizar el método `imprimir()` para diferentes tipos de datos sin cambiar de nombre.

Completar con archivo "_002_MetodosSobrecarga.java"

# 2. Resumen: argumentos, variables, métodos

**Resumen: clase, variables, métodos**

```java
// Primero la clase
public class EjemploResumen {

    // Método main
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        // Bloque de código del método main
        int numA = 3;
        int numB = 2;

        System.out.println("Resultado de suma utilizando los números "
        + numA + "y " + numB + " = " + suma(numA, numB));
    }

    // Por convención todo lo público se declara al principio.
    // Luego se declaran las privadas
    private static int suma(int a, int b) {
        return a + b;
    }
}
```

# 2.4. Operadores

Los operadores son símbolos especiales que nos permiten poder hacer diferentes operaciones sobre una, dos o más variables. Hay diferentes tipos de operadores:

- **Asignación**: para asignar valores.
- **Aritméticos**: para operaciones matemáticas.
- **Comparación**: para hacer comprobaciones entre valores.
- **Lógicos**: para combinar condiciones.
- **Ternario**: decidir entre dos valores con una condición.

| **Tipo** | **Símbolo** |
|---|---|
| Asignación | `=, +=, -=, *=, /=, %=` |
| Aritméticos | `+, -, *, /, %` |
| Comparación | `==, !=, <, >, <=, >=` |
| Lógicos | `&&, ||, !` |
| Ternario | `? :` |

# 2.4. Operadores: asignación (I)

## Asignaciones

Las operaciones que puedan realizarse se pueden hacer por asignación. Por ejemplo, quiero sumar el valor de `b` a la variable `a`:

```
int a = 5;
int b = 10;
```

Una vez declaradas las variables tengo dos opciones para sumarle a `a` el valor de `b` :

```
a = a + b; // a es igual a sí misma + b
a += b; // súmale el valor de b
```

| | | |
|---|---|---|
| = | Asigna el valor del operando derecho al izquierdo. | `a = b` |
| += | Suma el valor del operando izquierdo al derecho y asigna el resultado al operando izquierdo. | `a += b`<br>`a = a + b` |
| -= | Resta el valor del operando izquierdo al derecho y asigna el resultado al operando izquierdo. | `a -=b`<br>`a = a - b` |
| *= | Multiplica el valor del operando izquierdo al derecho y asigna el resultado al operando izquierdo. | `a *= b`<br>`a = a * b` |
| /= | Divide el valor del operando izquierdo al derecho y asigna el resultado al operando izquierdo. | `a /= b`<br>`a = a / b` |
| %= | Calcula el residuo de la división del operando izquierdo al derecho y asigna el resultado al operando izquierdo. | `a %= b`<br>`a = a % b` |

# 2.4. Operadores (II)

## Aritméticos

Los operadores aritméticos permiten hacer operaciones matemáticas básicas como suma, resta, multiplicación, división y módulo.

El resultado será del tipo numérico de mayor capacidad, entre los tipos numéricos que intervengan en la operación.

```
int a = 5;
double b = 3.1416;
double suma = a + b;
System.out.println(suma); //8.1416
```

- La división entre dos enteros (**`int`**) devuelve un resultado entero, trunca todos los decimales.
- El módulo (**`%`**) devuelve el residuo de la división. Puede ser útil para determinar si un número es par o impar, entre otros casos de uso.

Hay un orden jerárquico en las operaciones matemáticas: paréntesis -> potencias -> multiplicación, división, módulo -> suma, resta

| | | |
|---|---|---|
| = | Asigna el valor del operando derecho al izquierdo. | `int numero = 5` |
| + | Suma el valor del operando izquierdo al derecho. | `int suma = 5 + 3` |
| - | Resta el valor del operando izquierdo al derecho. | `int resta = 5 - 3` |
| * | Multiplica el valor del operando izquierdo al derecho. | `int multi = 5 * 3` |
| / | Divide el valor del operando izquierdo al derecho. | `double div = 5 / 2` |
| % | Calcula el residuo de la división del operando izquierdo al derecho. | `int modulo = 2 % 2` |
| + | También utilizado para la concatenación de cadenas de texto | `"Hola" + "Mundo"`<br>`// Hola Mundo` |
| ++ | Incremento, añade 1 unidad | `i++ , ++i` |
| -- | Decremento, quita 1 unidad | `i-- , --i` |

# 2.4. Operadores: incremento, decremento (III)

## Incremento y Decremento

Dentro de las diferentes operaciones aritméticas (suma, resta, multiplicación, división) están los **operadores de Incremento y Decremento**:

- **`variable++`**
- **`variable--`**

También se pueden aplicar a la inversa, primero el incremento/decremento antes de mostrar el valor de la variable.

- **`++variable`**
- **`--variable`**

Incrementan o decrementan en una unidad (1) el valor de la variable que están manipulando.

```java
// Incremento
int i = 10;
System.out.println(i++); // output: 10
                         // primero muestra i
                         // segundo incremento
System.out.println(i);   // output: 11

System.out.println(++i); // output: 12
                         // primero incremento
                         // segundo muestra i
System.out.println(i);   // output: 12
```

```java
// Decremento
int d = 10;
System.out.println(d--); // output: 10
                         // primero muestra d
                         // segundo decremento
System.out.println(d);   // output: 9

System.out.println(--d); // output: 8
                         // primero decremento
                         // segundo muestra d
System.out.println(d);   // output: 8
```

# 2.4. Operadores (IV)

## Comparación

Los operadores de comparación se utilizan para comprobar dos valores y ver qué relación tienen. El resultado será un **`boolean`** (`true` o `false`).

Son fundamentales para las estructuras de control, como los condicionales `if` y los bucles `while` y `for`, donde las decisiones se basan en comparaciones.

Al comparar cadenas de texto (`String`), se debe utilizar el método `.equals()`, pues el comparador `==` compara referencias de memoria y no el texto que contienen las cadenas.

| | | |
|---|---|---|
| `==` | Igual a | `a == b` |
| `!=` | Distinto de | `a != b` |
| `<` | Menor que | `a < b` |
| `>` | Mayor que | `a > b` |
| `<=` | Menor o igual a | `a <= b` |
| `>=` | Mayor o igual que | `a >= b` |
| `.equals()` | Igual a *sólo para String | `"hola".equals("hola")` |

# 2.4. Operadores (V)

## Lógicos y Condicional ternario

Los operadores lógicos se utilizan para combinar expresiones booleanas y tomar decisiones basadas en múltiples condiciones. Devuelven un resultado booleano.

| | | |
|---|---|---|
| && | AND lógico | Devuelve true si ambas condiciones son true. |
| \|\| | OR lógico | Devuelve true si al menos una condición es true. |
| ! | NOT lógico | Devuelve false si la condición es true. |
| ^ | XOR lógico | Devuelve true si una condición es true y la otra false. |

El operador condicional, conocido como *operador ternario*, permite evaluar una expresión booleana y retornar un valor u otro basándose en si la condición es `true` o `false`. Es una forma concisa de escribir una estructura `if-else`.

```
tipo nombre = (condicion) ? valor_true : valor_false;
```

# 3. Condicionales

Los bloques condicionales son una toma de decisiones que sirven para controlar el flujo de ejecución de los diferentes bloques de código.

Estas decisiones las estamos tomando constantemente: **si** (`if`) llueve cojo el paraguas; **si** (`if`) llevo suficiente efectivo pago con él, **si no** (`else if`) con tarjeta; **si** (`if`) tengo carne congelada la descongelo, **si no** (`else if`) la cocino directamente, **si no tengo carne** (`else`) hago pescado.

- `if`
- `else if`
- `else`
- `switch/case`
- `? :`

```
// Estructura condición
if (numA > numB) {
    // Bloque código
} else if (numA < numB) {
    // Bloque de código
} else {
    // Bloque de código
}

// Estructura switch
switch (numA) {
    case 1:
        System.out.println();
        break;
    case 2:
        System.out.println();
        break;
    default:
        System.out.println();
}

// Operador ternario
tipo variable = condicion ? casoTrue : casoFalse;
```

# 3. Condicionales: expresión booleana

Los condicionales permiten controlar el flujo de código en función de si cumple o no una condición.

Las condiciones son expresiones `boolean` que necesitan, en la mayoría de los casos, **operadores de comparación** que devolverán un resultado `true` si cumple o `false` si no cumple la condición.

Veamos algunos ejemplos de expresiones `boolean` sencillas y complejas antes de seguir con las diferentes formas de controlar los flujos de ejecución a través de condicionales.

```
int a = 1; int b = 2; int c = 3; int d = 4;

if (a > 0) {
    // ¿Ejecutará el bloque?
}

if (a > b) {
    // ¿Ejecutará el bloque?
}

if (b < c && d > c) {
    // ¿Ejecutará el bloque?
}

if (b < a && (c * d > b || c + d < a)) {
    // ¿Ejecutará el bloque?
}
```

# 3. Condicionales: if/else

Si se cumple la condición ejecuta el bloque de código.

```
if (condición) {
    instrucciones;
}
```

Pero ¿y si no se cumple la condición? Los bloques condicionales **`if`** no suelen ir solos, sino acompañados de una alternativa: **`else`**.

```
if (condición) {
    instruccionesBloqueIf;
} else {
    // Se activa si if no es true
    instruccionesBloqueElse;
}
```

```java
// Temperatura
int temp = 10;

// Muestra por pantalla la temperatura
System.out.print(s:"La temperatura es de "); // no mueve cursor de línea

if (temp > 0) {
    System.out.println(temp + "ºC positivos.");
} else {
    System.out.println(temp + "ºC bajo cero.");
}
```

Si la temperatura es 0 el bloque `if` no nos sirve, pero el bloque `else` tampoco. Necesitamos más condiciones.

# 3. Condicionales: anidación

Se pueden añadir condiciones dentro de otras condiciones. Esto es el concepto de **anidación**: tener una estructura dentro de otra. Generalmente, en programación, ocurre con estructuras similares. Condicionales dentro de otros condicionales, bucles dentro de bucles, colecciones dentro de otras colecciones…

Si la temperatura son 0ºC no puede salir “La temperatura es de 0ºC positivos.” ni “La temperatura es de 0ºC bajo cero.” (que es lo que sale si lo dejamos como está). Se pueden anidar condiciones y poner una nueva condición dentro del bloque `else` para hacer esa diferenciación.

```java
// Temperatura
int temp = 10;

// Muestra por pantalla la temperatura
System.out.print(s:"La temperatura es de "); // no mueve cursor de línea

if (temp > 0) {
    System.out.println(temp + "ºC positivos.");
} else {
    if (temp == 0) {
        System.out.println(temp + "ºC.");
    } else {
        System.out.println(temp + "ºC bajo cero.");
    }
}
```

Aunque esto dé el resultado esperado ir añadiendo condicionales dentro de otros dificulta la lectura y mantenimiento del código.

Siempre va a preservar la idea de poder mantener el código de maneras cómodas.

# 3. Condicionales: if/else if/else

En caso de necesitar más bloques condicionales tenemos **`else if`** para añadir otras condiciones. El ejemplo de la temperatura precisamente necesita tener tres supuestos siendo todos ellos del mismo nivel: temperatura positiva, negativa e igual a cero.

```
if (condición) {
    // Bloque que se activa si condición es true
} else if (otraCondición) {
    // Bloque que se activa si condición es false
    // y otraCondición es true
} else {
    // Bloque que se activa si todo lo anterior
    // es false
}
```

```java
// Temperatura
int temp = 10;

// Muestra por pantalla la temperatura
System.out.print(s:"La temperatura es de ");

if (temp > 0) {
    System.out.println(temp + "ºC positivos.");
} else if (temp == 0) {
        System.out.println(temp + "ºC.");
} else {
        System.out.println(temp + "ºC bajo cero.");
}
```

# 3. Condicionales: switch

Hay casos en los que podemos tener un condicional lleno de **`else if`**, como puede ser el de comprobar una figura geométrica que está en una misma variable entre diferentes posibilidades.

Dependiendo del número de lados tiene que hacer una asignación distinta:

```
if (numLados == 3) {
    figura = "triángulo";
} else if (numLados == 4) {
    figura = "cuadrilátero";
} else if (numLados == 5) {
    figura = "pentágono";
} else if (numLados == 6) {
    figura = "hexágono";
} else {
    figura = "desconocido";
}
```

Es en estos casos cuando es mejor utilizar la estructura **`switch`** que se puede escribir de diversas maneras:

```
switch (variable) {
    case valor1:
        sentencia;
        break;
    case valor2:
        sentencia;
        break;
    default:
        sentencia;
```

Esta es la estructura que siempre se ha utilizado, aunque a partir de la versión 12 de Java se añadió otra estructura más directa.

# 3. Condicionales: switch

Siguiendo con el ejemplo de la figura geométrica veamos cómo queda:

```
switch (numLados) {
    case 3:
        figura = "triángulo";
        break;
    case 4:
        figura = "cuadrilátero";
        break;
    case 5:
        figura = "pentágono";
        break;
    case 6:
        figura = "hexágono";
        break;
    default:
        figura = "desconocido";
        break;
}
```

Pero hay otra forma más directa de hacer esta misma estructura:

```
figura = switch (numLados) {
    case 3 -> "triángulo";
    case 4 -> "cuadrilátero";
    case 5 -> "pentágono";
    case 6 -> "hexágono";
    default -> "desconocido";
};
```

No solo es más directa. Permite, además, hacer una reasignación directa a una variable y evita el uso del **`break`** volviendo nuestro código más fácil de mantener y menos susceptible a errores.

# 3. Condicionales: switch

Estas son las nuevas palabras reservadas:

- **`switch`**: inicia la estructura. Va seguida de paréntesis `()` con el nombre de la variable sobre la que hacemos las evaluaciones y luego llaves `{}` dentro de las que metemos todos los casos.
- **`case`**: palabra reservada para cada caso. Va seguida del valor con el que se quiere comparar la variable en cada uno de los casos y dos puntos `:`

```
case <valor>:
```

- **`break`**: cuidado con esta palabra reservada. Alerta, suele dar problemas si nos lo olvidamos. En el caso de `switch` el `break` se utiliza para detener la ejecución en el `case` que cumple con el valor. Si entra en un `case` y no se topa con la palabra `break` pasaría al siguiente, realizando la correspondiente reasignación. En definitiva, nos sirve para detener la ejecución de un bloque de código.
- **`default`**: en caso de no cumplirse ningún `case` se le tendrá que asignar un valor igualmente a la variable. Ese valor es el que se asignará a la variable. Al ser la última instrucción no necesita `break`.

# 3. Condicionales: operador ternario

Se pueden hacer ejecuciones condicionales en una sola línea:

```
tipo variable = condicion ? caso1 : caso2
```

Puede resultar complejo de leer, pero con buena idea y uso únicamente cuando sean casos muy claros y cortos.

```
if (condición) {
    variable = caso1;
} else {
    variable = caso2;
}
```

Ambos hacen lo mismo, si es cierto devolverá el caso1 y si no lo es devolverá el caso2.

# 3.1. Constantes

Muchas veces estamos utilizando números en nuestro código. Debemos evitar utilizar esos números directamente en el código del programa ya que dificulta su mantenimiento.

Imagina que tenemos bloques condicionales repartidos por todo el código comparando valores con el 18 (la mayoría de edad) pero el Estado decide subir la edad mínima para comprar alcohol a los 21 años. Tendríamos que reemplazar ese 18 por un 21, con el riesgo de que alguno se nos escape.

Para evitar estos problemas se utilizan **constantes**, "variables" especiales que no pueden alterar su valor. Para lograr bloquear el valor necesitamos las palabras reservadas `static` y `final` en el momento de su declaración.

Además, por convención, las constantes se declaran poniendo su nombre en mayúsculas.

```
public static final int EDAD = 18;
private static final int HORA = 21;
protected static final double PI = 3.141592;
```

Pronto descubriremos qué significan las palabras `public`, `private` y `protected` con mayor profundidad. Ahora lo que nos interesa es recordar las diferentes normas de nombrado que hemos ido viendo:

camelCase: `nombreMetodo()`

camelCase: `nombreVariable`

PascalCase: `NombreClase`

SNAKE_UPPER_CASE: `NOMBRE_CONSTANTE`

# 4. Inputs y Outputs

En algunos programas vamos a necesitar la interacción con el usuario para que nos escriba a través del teclado. Para ello necesitamos utilizar la clase **`Scanner`**. Pronto aprenderemos más sobre las clases, pero lo importante es aprender a utilizarla.

Para poder utilizar cosas que no están en nuestras clases las vamos a tener que importar. Antes de declarar la clase es cuando tenemos que hacer la importación.

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

- **`System.in`**: lo vamos a utilizar para comunicaciones de entrada del sistema al programa.
- **`System.out`**: ya lo conocemos con `.print()` y lo utilizamos para comunicaciones de salida del programa al sistema.

```
// Primero el import
import java.util.Scanner;

// Segundo la clase
public class EjemploScanner {
```

Primero hay que importarlo ya que no es un objeto que estemos creando en nuestra clase. Luego declaramos la clase en la que vamos a trabajar.

Dentro del método `main`, por ahora, vamos a poder utilizarlo para interactuar con el usuario de nuestro programa.

```
// Primero el import
import java.util.Scanner;

// Segundo la clase
public class EjemploScanner {

    // Método main
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        // Instancia de objeto, ya sabrás qué es
        Scanner scanner = new Scanner(System.in);

        // Mensaje para el usuario
        System.out.println(x:"Dime tu nombre:");

        // Variable que almacena el nombre del usuario
        String nombre = scanner.nextLine();

        // Saludo del programa al usuario
        System.out.println("¡Hola, " + nombre + "! ¿Qué tal?");
```

# 4.1. Inputs: Scanner

Java es un lenguaje de programación de tipado estático y, así como cada variable tiene que estar asociada a un tipo de dato concreto, lo mismo pasa con Scanner para los diferentes *inputs* que puedan venir de los usuarios.

Si queremos que nos introduzcan un único carácter hay que tener en cuenta que para inputs de texto hay dos maneras básicas: `.next()` y `nextLine()`. El primero lo hará hasta el primer espacio y el segundo hasta el primer salto de línea. Utilizando `next()` obtendremos un input de la primera palabra, pero únicamente queremos la primera letra, que es un tipo `char`. Recuerda que `String` es una clase y tiene métodos. También una cadena de texto es una secuencia de caracteres, así que cada uno tiene su índice posicional, el primero será `0`.

`String` tiene un método para extraer el carácter que hay en una determinada posición `charAt()`. Así que si queremos guardar un único carácter:

```
char letra = scanner.next().charAt(0);
```

Con `next()` obtenemos esa `String` de la que nos quedamos únicamente su primer `char`.

| Tipo de dato | Método |
|---|---|
| `byte` | nextByte() |
| `short` | nextShort() |
| `int` | nextInt() |
| `long` | nextLong() |
| `float` | nextFloat() |
| `double` | nextDouble() |
| `boolean` | nextBoolean() |
| `String` | nextLine() / next() |

# 4.2. Outputs: err

Su uso principal es para mostrar **errores** o **advertencias** que deben ser atendidas por el usuario.

Se puede redirigir desde la terminal a un archivo.

Cuando trabajamos con un IDE, normalmente, se destaca en la consola el mensaje de error en un color distinto, que suele ser **rojo**.

En un momento más avanzado del curso iremos utilizando cada vez menos esta metodología para gestionar errores y conoceremos otras formas de lanzar mensajes de información y de error con una librería desarrollada por Apache.

| Método | Funcionamiento |
|---|---|
| `.print()` | Imprime el argumento en la consola. |
| `.println()` | Imprime el argumento en la consola y añade un salto de línea. |
| `.printf()` | Permite formatear la salida. |
| `.format()` | Similar a `printf()` pero es más versátil. |
| `.flush()` | Limpia el buffer. |
| `.close()` | Cierra el flujo de salida. Importante cuando la salida se redirige a otros archivos. |

# 4.3. Outputs: out

Se utiliza para mostrar información o resultados. Tiene un mayor rendimiento que `System.err` y por eso los outputs de errores se utilizan en momentos muy concretos que se necesite destacar esa advertencia.

Se puede redirigir su salida a archivos externos.

Comparte métodos con `System.err`

| **Método** | **Funcionamiento** |
|---|---|
| `.print()` | Imprime el argumento en la consola. |
| `.println()` | Imprime el argumento en la consola y añade un salto de línea. |
| `.printf()` | Permite formatear la salida. |
| `.format()` | Similar a `printf()` pero es más versátil. |
| `.flush()` | Limpia el buffer. |
| `.close()` | Cierra el flujo de salida. Importante cuando la salida se redirige a otros archivos. |

# 4.4. Ejemplo: In, Out

```java
// import de la clase Scanner
import java.util.Scanner;
// class creada para el ejemplo
public class AAA_EjemploInOut {
    // método main
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        // Creación del objeto en una variable
        Scanner sc = new Scanner(System.in);

        // Output: mensaje que aparecerá en consola
        System.out.println(x:"Introduce un número entero mayor que 5:");
        // Input: guardado en una variable para poderlo usar posteriormente
        int numero = sc.nextInt(); // el usuario escribe y presiona ENTER

        // Comprobación del número introducido
        if (numero <= 5) { // Si está mal lanzará un error
            System.err.println("Has introducido: " + numero + ". Número incorrecto.");
        } else {
            System.out.println("Número introducido: " + numero);
        }
    }
}
```

# 5. Bucles

Ya sabemos controlar el flujo de código en función de si se cumplen o no ciertas condiciones. Con los **bucles** vamos a poder hacer las mismas instrucciones una y otra vez mientras se cumpla una condición.

Al igual que con las condiciones hay varios tipos de bucle que nos van a facilitar el flujo del programa según qué tarea queramos que haga.

- `for each`
- `for`
- `while`
- `do while`

```java
public class EjemploBucles {
    // Método main
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        // Array de char
        char[] nombre = {'B', 'u', 'c', 'l', 'e'};

        // Bucle for-each
        for (char letra : nombre) {
            System.out.println(letra);
        }

        // Bucle for
        for (int i = 0; i < nombre.length ; i++) {
            System.out.println(nombre[i]);
        }

        // Bucle while
        int j = 0;
        while (j < nombre.length) {
            System.out.println(nombre[j]);
            j++;
        }

        // Bucle do-while
        int k = 0;
        do {
            System.out.println(nombre[k]);
            k++;
        } while (k < nombre.length);
```

# 5. Bucles: for-each

- Disponible a partir de la versión 5 de Java.

*For each* significa 'para cada'. Está pensado para aplicar las instrucciones del bloque de código por cada elemento de una secuencia, se utiliza cuando vamos a aplicar esa tarea a todos los elementos. Si no se hace a todos ellos hay otras estructuras.

```
for (<tipo> <elemento> : <lista>) {
    // bloque de código
}
```

Recuerda que al ejecutar nuestro programa podemos introducir diferentes argumentos. El siguiente bucle va a recorrer los distintos argumentos y mostrarlos por la consola:

```
public class ForEach {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        for (String s : args) {
            System.out.println(s);
        }
    }
}
```

- **`for`**: palabra reservada que inicia la estructura del bucle.
- **`<tipo>`**: al ser Java tipado hay que indicar el tipo de cada elemento de la secuencia.
- **`<elemento>`**: nombre que tendrá cada elemento. Como cualquier variable. Este nombre lo usaremos dentro del bloque del bucle para hacer referencia al elemento.
- **`:`** : se puede leer como 'dentro de' o 'en'.
- **`<lista>`**: *array*, lista, colección… aquello que contiene todos los elementos que queremos recorrer en el bucle.

# 5. Bucles: for

Este bucle es “el de toda la vida”. Le estamos dando la instrucción de que vamos a repetir un bloque de código una cantidad de veces determinada.

Se inicia en el valor que tiene inicialización, hasta que se cumpla una condición avanzando como se indique en las instrucciones de avance.

```
for (<inicialización>; <condición>; <avance>) {
    // bloque de código
}
```

Recuerda que al ejecutar nuestro programa podemos introducir diferentes argumentos. El siguiente bucle va a recorrer los distintos argumentos y mostrarlos por la consola:

```
public class For {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++)
            System.out.println(args[i]);
        }
    }
}
```

- **`for`**: palabra reservada que inicia la estructura del bucle.
- **`<inicialización>`**: instrucción que declara e inicializa la variable con la que trabajaremos en el bucle. Muy típico: `int i = 0;`
- **`<condición>`**: instrucción con la que se controla si hay que seguir iterando (repitiendo el bucle) o se debe finalizar.
- **`<avance>`**: instrucción que hace avanzar el bucle, modificando por ejemplo el valor de la variable de inicialización utilizada de contador. Si no se hace bien se genera un bucle infinito que se ejecutará hasta que reviente la memoria. Muy típico: `i++;`

# 5. Bucles: while

Este bucle es condicional. Mientras se cumpla la condición se seguirá ejecutando. Hay veces que no tenemos una lista o no sabemos la cantidad concreta de veces que necesitamos se ejecute el bucle, es aquí cuando es de gran utilidad.

```
<inicialización>;
while (<condición>) {
    // bloque de código
    <avance>;
}
```

Recuerda que al ejecutar nuestro programa podemos introducir diferentes argumentos. El siguiente bucle va a recorrer los distintos argumentos y mostrarlos por la consola:

```java
public class While {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        int i = 0;
        while (i < args.length) {
            System.out.println(args[i]);
            i++;
        }
    }
}
```

- **`while`**: palabra reservada que inicia la estructura del bucle.
- **`<inicialización>`**: si lo necesitamos, se hace fuera del bucle. Es la instrucción que declara e inicializa la variable con la que controlaremos el bucle.
- **`<condición>`**: condición para entrar en el bucle, si es cierta entrará.
- **`<avance>`**: instrucción que hace avanzar el bucle, modificando por ejemplo el valor de la variable de inicialización utilizada de contador. Si no se hace bien se genera un bucle infinito que se ejecutará hasta que reviente la memoria.

# 5. Bucles: do while

Es el bucle menos frecuente y funciona de una manera similar a `while`. Primero ejecuta el bloque de código y luego comprueba la condición. Siempre usaremos este bucle cuando queramos ejecutar sí o sí el primer bloque de código, como puede ser una petición a un usuario que si nos la da mal seguiremos insistiendo, pero la primera vez se le va a pedir pase lo que pase.

```
<inicialización>;
do {
    // bloque de código
    <avance>;
} while (<condición>); // No olvidar ;
```

Recuerda que al ejecutar nuestro programa podemos introducir diferentes argumentos. El siguiente bucle va a recorrer los distintos argumentos y mostrarlos por la consola:

```java
public class DoWhile {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        int i = 0;
        do {
            System.out.println(args[i]);
            i++;
        } while (i < args.length);
    }
}
```

- **`do`**: palabra clave que indica que tiene que hacer el siguiente bloque de código.
- **`while`**: palabra reservada que inicia la estructura del bucle.
- **`<inicialización>`**: si lo necesitamos, se hace fuera del bucle.
- **`<condición>`**: condición para seguir dentro del bucle, si es falsa terminará.
- **`<avance>`**: instrucción que hace avanzar el bucle. Si no se hace bien se genera un bucle infinito que se ejecutará hasta que reviente la memoria.

# 6. Útiles: Math.random(), array, métodos String

- `Math.random()`

  Es una herramienta fundamental de Java para generar números “aleatorios”.

- `Array`

  Estructura de datos que permiten almacenar múltiples valores del mismo tipo en una misma variable.

- Métodos de `String`

  La clase `String` tiene una variedad de métodos para manipular y analizar cadenas de texto.

```java
public class NombreAleatorio {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        // Definición de arrays con nombres y apellidos
        String[] nombres = {"Ana", "Luis", "María", "Carlos", "Elena"};
        String[] apellidos = {"García", "Pérez", "López", "Martínez", "Sánchez"};

        // Generación de índices aleatorios para seleccionar nombre y apellido
        int indiceNombre = (int) (Math.random() * nombres.length);
        int indiceApellido = (int) (Math.random() * apellidos.length);

        // Creación del nombre completo
        String nombreCompleto = nombres[indiceNombre] + " " + apellidos[indiceApellido];

        // Impresión del nombre completo en mayúsculas
        System.out.println("Nombre generado: " + nombreCompleto.toUpperCase());
    }
}
```

# 6.1. Útiles: Math.random()

**`Math.random()`** es un método `static` de la clase `Math` que devuelve un número aleatorio decimal del tipo `double` entre `0.0` y `1.0`.

Para poder utilizarlo hay que aplicar una sencilla fórmula utilizando el mínimo y el máximo entre el rango que necesitamos.

Por ejemplo, si queremos simular un dado que va del 1 al 6 el mínimo será 1 y el máximo será 6.

Fórmula:

`Math.random() * (max - min + 1) + min`

**Atención:**

Si únicamente queremos un número entero vamos a tener que castear el tipo de dato. Este método devolverá un `double`. Para quedarnos la parte entera simplemente convirtiendo el tipo a `int` lo tendremos. Nuestro mínimo ya es un `int`, así que lo que necesitamos castear es el método * número. Para ello se tendrá que agrupar entre paréntesis de la siguiente manera:

`(int) (Math.random() * (max - min + 1)) + min`

```
// FORMULA: Math.random() * (max - min + 1) + min

// Número aleatorio entre 1 y 6
// (max - min + 1) = 6 - 1 + 1 = 6
int dadoAleatorio = (int) (Math.random() * 6) + 1;

// Número aleatorio entre 0 y 10
// (max - min + 1) = 10 - 0 + 1 = 11
int numAleatorio = (int) (Math.random() * 11) + 0;

// Una array, por posición, empieza con indice = 0
// Un elemento aleatorio tendrá índice entre 0 y su longitud - 1
```

# 6.1. Útiles: Clase Random()

**Random** es una clase nativa de Java y genera números *pseudoaleatorios* basados en una semilla. Para poder usarlo es necesario importarlo, antes de la declaración de la clase:

```
import java.util.Random;
```

Para poder utilizar sus métodos necesitamos instanciar un objeto.

```
Random <nombre> = new Random();
```

Si se utiliza una semilla los números aleatorios que va a generar serán siempre los mismos. Es necesario instanciar el objeto con la semilla.

```
Random <nombre> = new Random(42);
```

El número 42 puede ser cualquier otro dentro del rango permitido en el tipo numérico **`long`**.

Un objeto random sin semilla va a utilizar la hora actual en milisegundos, por eso cada ejecución es diferente. En cambio, al usar semilla, dos objetos diferentes pero con la misma semilla generarán una secuencia aleatoria idéntica.

## METODOS

- **`nextInt()`**: genera un número aleatorio dentro del rango **`int`**.
- **`nextInt(int bound)`**: genera un número entre 0 y el número que se ponga como argumento -1.
```
int numAleatorio = rd.nextInt(10);
// genera aleatorio entre 0 y 9 (10-1)
```

- **`nextInt(int origen, int limit)`**: genera un número entre origen (incluido) y limit (excluido).
```
int numAleatorio = rd.nextInt(1, 11);
// genera aleatorio entre 0 y 10 (11 excluido)
```

- **`nextLong()`**: genera un número aleatorio dentro del rango **`long`**.
- **`nextDouble()`**: genera un número aleatorio entre `0.0` y `1.0`.
- **`nextDouble(double origen, double limit)`**: genera un número entre origen (incluido) y limit (excluido).
```
int numAleatorio = rd.nextDouble(1.0, 5.0);
// genera un aleatorio entre 1.0 y 4.9999999… (5.0 excluido)
```

- **`nextFloat()`**: genera un número aleatorio entre `0.0f` y `1.0f`.
- **`nextBoolean()`**: genera `true` o `false` con un 50% de probabilidad para cada uno.
- **`nextBytes(byte[])`**: rellena una `array` de `byte` con valores aleatorios. Para eso la `array` debe existir primero.
```
// crea una array de 5 elementos que se llama arrayByte
byte[] arrayByte = new byte[5];
// pon a cada elemento un número del rango byte
rd.nextBytes(arrayByte);
```

- **`setSeed(long semilla)`**: establece una nueva semilla al objeto Random.
```
rd.setSeed(258); // las secuencias aleatorias se basarán en esta semilla
```

# 6.1. Útiles: Random vs Math.random()

| Característica | Random | Math.random() |
|---|---|---|
| **Forma de uso** | Se usa creando un objeto `Random rd = new Random();` | Se usa como un método estático `Math.random();` |
| **Necesidad de instancia** | **Sí**, necesita una instancia (`new Random()`) | **No**, se accede directamente **sin** crear un objeto |
| **Rango de valores** | **Varía** según el método (`nextInt(10)`, `nextDouble(1.0, 5.0)`) | Siempre entre `0.0` y `1.0` |
| **Personalización con semilla** | **Sí**, permite definir una semilla (`new Random(1234)`) | **No**, siempre genera valores distintos en cada ejecución |
| **Tipo de retorno** | Puede devolver `int`, `long`, `double`, `float`, `boolean`, `byte[]` | Devuelve solo `double` |
| **Facilidad de uso** | Más flexible y configurable | Más simple y directo |
| **Soporte para múltiples tipos de datos** | **Sí**, permite generar diferentes tipos de valores aleatorios | **No**, solo genera **`double`** en el rango `[0.0, 1.0]` |
| **Uso recomendado** | Cuando se necesita **reproducibilidad** o más **control** sobre la aleatoriedad | Cuando solo se necesita un **número aleatorio** sin configuraciones adicionales |

# 6.2. Útiles: array

Es una colección de elementos **ordenados**, todos del **mismo tipo**, que son accesibles a través de su **índice**. Algo parecido a tener una lista de cosas enumeradas en la vida real.

Como las variables se pueden declarar sin asignar valor:

`<tipo>[] nombre;`

También se pueden inicializar indicando el tipo de dato y la cantidad de elementos:

`<tipo>[] nombre = new tipo[cantidad];`

```
int[] numeros = new int[5];
```

También se pueden declarar e inicializar al mismo tiempo con información:

`<tipo>[] nombre = {elemento1, elemento2, …}`

```
int[] numeros = {1, 3, 6, 8, 10};
```

Para acceder a cualquier elemento del `array` se necesita su índice, que se indica entre corchetes a continuación del nombre que tiene la colección:

`nombre[indice]`

```
numeros[3] // el indice 3 lo tiene el número 8 --> [1, 3, 6, 8, 10]
```

Una vez se define un `array` **no se puede modificar su tamaño**.

```java
// Declaración
int[] numeros;
String[] textos;
boolean[] trueFalse;

// Inicialización
double[] decimales = new double[3]; // podrá tener 3 elementos
                                    // todos ellos double

// Declaración + Inicialización
char[] letras = {'a', 'b', 'c'}; // se ponen los elementos
                                 // siempre entre llaves {}
                                 // separados por coma ,

// Acceso --> siempre por índice. Si quiero acceder a la b...
char letraB = letras[1];

// Recorrido con bucle for / for-each
for (int i = 0; i < letras.length; i++) {
    // imprime cada letra
    System.out.println(letras[i]);
}

for (char l : letras) {
    // imprime cada letra
    System.out.println(l);
}
```

# 6.2. Útiles: array

## Print de array

¿Qué ocurre cuando vamos a hacer un `System.out.println(array);`?

No nos va a mostrar los diferentes elementos que tenemos dentro del **array**. Nos mostrará su localización en la memoria de la siguiente manera:

```
// Tenemos un array de int
int[] numeros = {1, 2, 3, 4};

// Queremos ver los elementos del array por pantalla
System.out.println(numeros);

// Salida por pantalla (invención de ejemplo):
[I3b457b6689
```

- **`[`:** indica que es un array
- **`I`:** indica que es del tipo `int`
- **`numeros`:** es el hash, un código, de su lugar en la memoria

### ¿Cómo se puede solucionar esto?

La clase **`Arrays`** está dentro del paquete `java.util` → como Scanner.
Para poder utilizar los métodos que están asociados a **array** se tendrá que importar primero la clase. → `import java.util.Arrays;`
Una vez está importada la clase se podrán utilizar diferentes métodos que se pueden consultar en el manual:
https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html

El que nos interesa para poder ver el contenido del **array** es `toString()`.

```
System.out.println(Arrays.toString(numeros));

// Ahora la salida que veremos es:
[1, 2, 3, 4]
```

```java
import java.util.Arrays;
public class _05_EjemploArrayPresentacion {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        // Declaración + inicialización de array
        int[] numeros = {1, 3, 5, 7, 9};
        String[] nombres = {"Pepe", "Sara", "Flor"};

        // Muestra por pantalla el array
        System.out.println(numeros); // [I@659e0bfd
            // [ --> es un array
            // I --> es int
            // 659e0bfd --> hash de la memoria
        System.out.println(nombres); // [Ljava.lang.String;@2a139a55
            // [L --> es un array
            // java.lang.String --> es de la clase String
            // @2a139a55 --> hash de la memoria en hexadecimal

        // QUIERO VER EL CONTENIDO DEL ARRAY
        // Se importa java.util.Arrays y se usa la función toString()
        System.out.println(Arrays.toString(numeros)); // [1, 3, 5, 7, 9]
        System.out.println(Arrays.toString(nombres)); // [Pepe, Sara, Flor]
    }
}
```

# 6.3. Útiles: métodos String

- **`length()`**: devuelve la longitud, el número de caracteres que contiene.
- **`charAt(int index)`**: devuelve el carácter en la posición indicada en `index`, que tiene que ser un número `int`.
- **`substring(int beginIndex, int endIndex)`**: extrae una subcadena que va de `beginIndex` a `endIndex-1`, siendo ambos números enteros.
- **`toUpperCase()/toLowerCase()`**: convierte toda la cadena en mayúsculas/minúsculas.
- **`equals(Object anObject)`**: compara si dos cadenas son iguales y devuelve un `boolean`.
- **`equalsIgnoreCase(String anotherString)`**: compara si dos cadenas son iguales, ignorando mayúsculas y minúsculas, devuelve un `boolean`.
- **`contains(CharSequence s)`**: comprueba si coincide la secuencia de uno o más caracteres dentro de la `String`, devuelve `boolean`.
- **`indexOf(String str)`**: devuelve el índice en el que se encuentra la cadena especificada, devuelve un `int`.
- **`replace(CharSequence target, CharSequence replacement)`**: reemplaza el texto que haya como target por el texto que haya como replacement.
- **`trim()`**: elimina espacios a principio y final de la `String`.
- **`split(String regex)`**: divide la `String` en un `array` de cadenas cortando por donde se indique en la expresión `regex`.
- **`concat(String str)`**: concatena la `String` con la que se indique.
- **`isEmpty()`**: comprueba si está vacía, es decir, su longitud es `0`; devuelve `boolean`.
- **`startsWith(String prefix)`**: comprueba si empieza con la `String` que se indique, devuelve un `boolean`.
- **`endsWith(String suffix)`**: comprueba si termina con la `String` que se indique, devuelve `boolean`.

**Importante:** una **`String`** es **inmutable**. Cualquier método que "modifique" un `String` devuelve una **cadena nueva**.

```java
String texto = "Hola Mundo";
int longitud = texto.length(); // 10
char letra = texto.charAt(index:0); // 'H'
String subcadena = texto.substring(beginIndex:0, endIndex:4); // "Hola"
String mayus = texto.toUpperCase(); // "HOLA MUNDO"
String minus = texto.toLowerCase(); // "hola mundo"
boolean esIgual = texto.equals(anObject:"Hola Mundo"); // true
boolean esIgualIgnore = texto.equalsIgnoreCase(anotherString:"hola mundo"); // true
boolean contiene = texto.contains(s:"Mundo"); // true
int posicion = texto.indexOf(str:"Mundo"); // 5
String reemplazo = texto.replace(target:"Mundo", replacement:"Java"); // "Hola Java"

// texto con espacios
String textoEspacios = "  Hola Mundo  ";
String sinEspacios = textoEspacios.trim(); // "Hola Mundo"

String[] palabras = texto.split(regex:" "); // ["Hola", "Mundo"]

// varias cadenas
String saludo = "Hola";
String mundo = " Mundo";
String concatenacion = saludo.concat(mundo); // "Hola Mundo"

// cadena vacia
String vacia = "";
boolean estaVacia = vacia.isEmpty(); // true

boolean empiezaCon = texto.startsWith(prefix:"Hol"); // true
boolean terminaCon = texto.endsWith(suffix:"ndo"); // true
```

# 6.4. Útiles: Fechas

En los diferentes lenguajes de programación hay muchos formatos para representar las fechas.

Las fechas son fundamentales para:

- **Registrar eventos**: creación de un documento, log de registro…
- **Gestión de vencimientos**: facturación, suscripciones…
- **Cálculos de tiempo**: duración de un evento, diferencia entre dos fechas…
- **Ordenar y filtrar datos** en aplicaciones, bases de datos…

Trabajar con fechas es desafiante, pues hay que tener en cuenta diferentes problemáticas:

- Diferencia entre **zonas horarias**.
- **Formatos de fecha** → DD/MM/YYYY (Europa) || MM/DD/YYYY (EE.UU.)
- **Diferencias entre fecha y hora**: algunas aplicaciones únicamente necesitan la fecha, otras incluyen además horas, minutos, segundos.
- **Fechas relativas**: ayer, hoy, mañana.

# 6.4. java.util.Date || java.util.Calendar

En Java la manera de tratar las fechas ha ido evolucionando. Hay **tres principales formas** y, aunque dos de ellas están obsoletas, todavía se pueden encontrar en código que se haya hecho hace tiempo y hay que entender su funcionamiento aunque estén prácticamente deprecadas. Esta sección debe completarse con los archivos de código que hacen referencia a las clases `Date` y `Calendar`, así como al paquete `time`.

## `import java.util.Date;`

- Fue introducida en **Java 1.0 – 1996**
- Sus métodos están **obsoletos**.
- **No** soporta **zonas horarias** correctamente.
- Representa un instante de tiempo en **milisegundos** a contar desde **1970-01-01 00:00:00 UTC.**
- **No es inmutable** y eso puede ocasionar errores.

Es muy problemática para operar con fechas, como sumar días, meses o años a una fecha.

Actualmente debe usarse `java.time`

## `import java.util.Calendar;`

- Fue introducida en **Java 1.1 – 1997**
- Sus métodos son confusos, aunque siguen funcionando a día de hoy.
- **No** maneja **zonas horarias** correctamente (uso complicado con `TimeZone`).
- Meses poco intuitivos (0 = enero).
- **No es inmutable** y eso puede ocasionar errores.

Permite hacer operaciones con fechas como añadir días, meses, quitar años…

Actualmente debe usarse `java.time`

FechasDate.java

FechasCalendar.java

# 6.4. java.time

**`java.time`** es una API moderna y robusta introducida en **Java 8** superando las limitaciones de las clases anteriores `Date` y `Calendar`. Tiene diferentes clases que nos permiten la utilización de múltiples métodos. Las tres principales clases son `LocalDate`, `LocalTime` y `LocalDateTime`. Las fechas que proporcionan las diferentes clases son **inmutables** y además hay un buen soporte para las diferentes zonas horarias.

**`import java.time.LocalDate;`**

- Trabajar cómodamente con fecha sin hora ni zona horaria.
- Se puede hacer la instancia de muchas maneras, según la necesidad.
- Puede **comparar** fechas.
- Extraer diferentes **componentes** fácilmente.
- Hacer diferentes **operaciones**.
- Dar diferentes **formatos** a la fecha.

Para profundizar más en el manual está la lista de los diferentes métodos que se pueden utilizar.

https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html

**`import java.time.LocalTime;`**

- Trabaja con una hora, sin zona horaria. Útil cuando hay que manejar uso de horas indistintamente de la fecha (como horas de alarmas…)
- Se puede hacer la instancia de muchas maneras.
- Puede **comparar** tiempos.
- Extraer diferentes **componentes** fácilmente.
- Hacer diferentes **operaciones**.
- Dar diferentes **formatos** a la hora.

Para profundizar más en el manual está la lista de los diferentes métodos que se pueden utilizar.

https://docs.oracle.com/javase/8/docs/api/java/time/LocalTime.html

**`import java.time.LocalDateTime;`**

- Trabaja con fecha y hora sin información de zona horaria. Útil cuando la zona no es relevante, como para la programación de tareas.
- Se puede hacer la instancia de muchas maneras.
- Puede **comparar** tiempos.
- Extraer diferentes **componentes** fácilmente.
- Hacer diferentes **operaciones**.
- Dar diferentes **formatos**.

Para profundizar más en el manual está la lista de los diferentes métodos que se pueden utilizar.

https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html

# 6.4. java.time

**`java.time`** es una API moderna y robusta introducida en **Java 8** superando las limitaciones de las clases anteriores `Date` y `Calendar`. Tiene diferentes clases que nos permiten la utilización de múltiples métodos. Las tres principales clases son `LocalDate`, `LocalTime` y `LocalDateTime`. Las fechas que proporcionan las diferentes clases son **inmutables** y además hay un buen soporte para las diferentes zonas horarias.

**`import java.time.Duration;`**

- Mide periodos de tiempo basados en segundos y nanosegundos.
- Útil para cálculos de tiempo específicos, como tiempo transcurrido en una operación.
- Se pueden crear **periodos** de tiempo.
- **Calcular** el tiempo transcurrido entre dos fechas u horas de manera específica.
- Extraer diferentes **componentes** fácilmente.

Para profundizar más en el manual está la lista de los diferentes métodos que se pueden utilizar.

https://docs.oracle.com/javase/8/docs/api/java/time/Duration.html

**`import java.time.Period;`**

- Mide periodos de tiempo basados en fechas (día, mes, año).
- Útil para diferencia entre fechas y calcular edades, por ejemplo.
- Se pueden crear **periodos** de tiempo.
- **Calcular** el tiempo transcurrido entre dos fechas u horas de manera específica.
- Extraer diferentes **componentes** fácilmente.

Para profundizar más en el manual está la lista de los diferentes métodos que se pueden utilizar.

https://docs.oracle.com/javase/8/docs/api/java/time/Period.html

**`import java.time.ZonedDateTime;`**

- Fecha, hora y zona horaria.
- Se puede hacer la instancia de muchas maneras.
- Puede **comparar** tiempos.
- Extraer diferentes **componentes** fácilmente.
- Hacer diferentes **operaciones**.
- Dar diferentes **formatos**.
- Modificar **zonas horarias**.

Para profundizar más en el manual está la lista de los diferentes métodos que se pueden utilizar.

https://docs.oracle.com/javase/8/docs/api/java/time/ZonedDateTime.html