

# POO

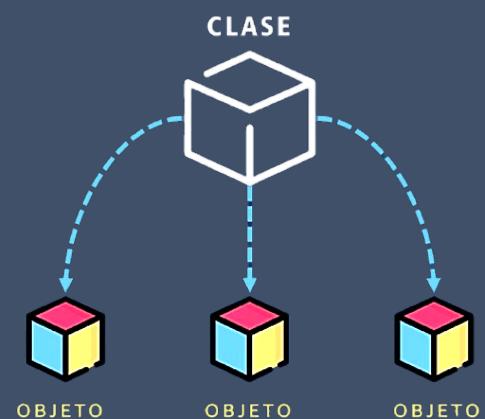
## Programación Orientada a Objetos



# Manual

Docente Ingeniero

Alexander Montoya



### Autores Estudiantes

- Kevin Julián Guerrero Penagos ID 821270
- Carlos Jhoan Calderón Falla ID 931012
- Emanuel Rincón Sierra ID 938723

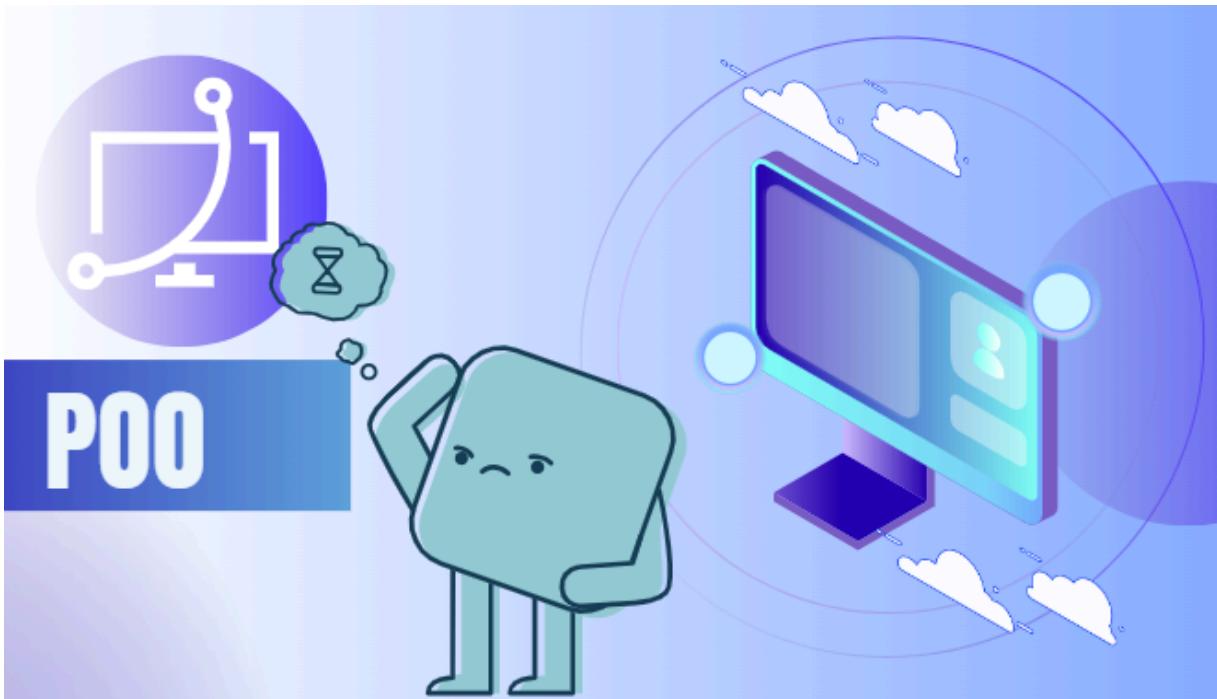
<b>Aprendiendo Programación Orientada a Objetos: teoría, lógica y videojuegos.....</b>	<b>3</b>
<b>1. Introducción a la POO.....</b>	<b>4</b>
1.1 Definición.....	4
1.2 Objetivo de la POO.....	4
1.3 Ejemplo aplicada en la Vida Real.....	5
1.4 Ejemplo aplicado a los videojuegos.....	6
1.5 Diferencia con la programación estructurada.....	7
<b>2. Instalación y Configuración del Entorno de Desarrollo.....</b>	<b>8</b>
2.1 ¿Qué es JDK?.....	8
2.2 ¿Cómo instalar el IDE?.....	8
2.3 ¿Ventajas de usar un IDE?.....	12
2.4 Creación de un Proyecto en Eclipse IDE.....	13
<b>3. Variables en la Programación Orientada a Objetos.....</b>	<b>19</b>
3.1 Concepto de variable.....	19
3.2 Tipos de variables en POO.....	20
Strings.....	20
Enteros (int).....	21
Flotantes (float).....	21
Booleanos (boolean).....	21
3.3 Variables en videojuegos: ejemplos prácticos.....	22
3.4 Variables en acción: ejemplo práctico en videojuegos.....	23
a. Incrementar vida.....	23
b. Modificar velocidad.....	23
c. Cambiar estado.....	23
d. Ejemplo práctico: enemigo.....	24
e. Mostrar información del enemigo.....	25
f. Simulación de daño al enemigo.....	25
g. Verificar si el enemigo sigue vivo.....	25
3.5 Buenas prácticas al usar variables.....	26
<b>4. Fundamentos de la Programación Orientada a Objetos.....</b>	<b>27</b>
4.1 Concepto de clase y objeto.....	27
4.2 Atributos y métodos.....	30
4.4 Encapsulamiento.....	32
4.5 Herencia.....	33
4.6 Polimorfismo.....	34
4.7 Abstracción.....	35

## Aprendiendo Programación Orientada a Objetos: teoría, lógica y videojuegos

Este manual está orientado al aprendizaje y enseñanza de la Programación Orientada a Objetos (**POO**) desde una perspectiva técnica y práctica.

Su propósito es explicar de manera clara los conceptos fundamentales de la POO, como **clases, objetos, atributos, métodos, herencia, polimorfismo y encapsulamiento**, apoyándose en ejemplos matemáticos, lógicos y, especialmente, en contextos de **videojuegos**, donde este paradigma es clave para la creación de personajes, escenarios y mecánicas.

De esta forma queremos que pueda comprender la teoría y al mismo tiempo, visualizar su aplicación en situaciones reales, logrando un aprendizaje más dinámico y contextualizado.

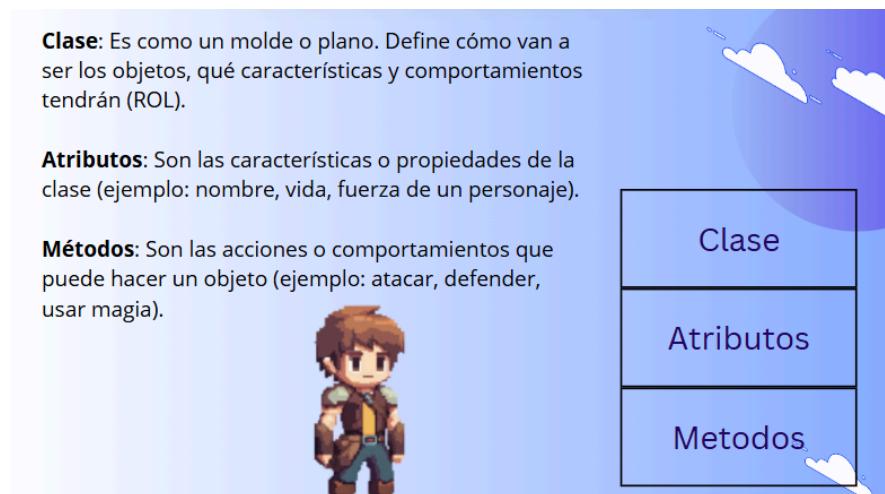


# 1. Introducción a la POO

## 1.1 Definición

La Programación Orientada a Objetos (POO) es un paradigma de desarrollo de software que organiza el código en estructuras llamadas **objetos**, los cuales son la unidad principal de construcción dentro de un programa, cada objeto combina en sí mismo tanto los **atributos** (que representan características o información del objeto) como los **métodos** (que definen las acciones o comportamientos que dicho objeto puede realizar).

De esta manera, la **POO** permite modelar el software de una forma muy similar a como los seres humanos perciben la realidad, ya que cada objeto refleja elementos tangibles o conceptuales, tales como una persona, un vehículo, un personaje de videojuego o incluso un sistema completo.



## 1.2 Objetivo de la POO.

El principal objetivo de la POO es proporcionar una forma estructurada y modular de programar, permitiendo la reutilización del código, la reducción de la complejidad y la facilidad de mantenimiento de los programas, nos facilita la creación de sistemas escalables y más cercanos a la realidad.

### 1.3 Ejemplo aplicada en la Vida Real.

Un **carro** puede representarse como una **clase**, la cual es como un molde o plantilla que describe las características y comportamientos comunes de todos los carros.

#### a. Atributos :

Son las características que describen al carro. Por ejemplo:

- **color** → puede ser rojo, azul, negro, etc.
- **marca** → Toyota, Ford, Chevrolet, etc.
- **modelo** → 2020, 2023, etc.

Estos atributos permiten diferenciar un carro de otro.

#### b. Métodos:

Son las funciones que puede realizar un carro (**NOTA** Los métodos siempre se encuentran cuando está escrito con “()”). Por ejemplo:

- **arrancar()** → enciende el motor.
- **frenar()** → detiene el carro.
- **acelerar()** → aumenta la velocidad.

Estos métodos son iguales para todos los carros, pero la forma en que se ejecutan puede variar según los atributos de cada carro.

#### c. Objetos (nace de la clase):

Cuando se crea un carro específico a partir de la clase, se obtiene un **objeto**. Por ejemplo:

- Carro1: color = rojo, marca = Toyota, modelo = 2020

- Carro2: color = azul, marca = Ford, modelo = 2023

## 1.4 Ejemplo aplicado a los videojuegos.

En el mundo de los **videojuegos**, en **(POO)** es esencial para organizar y estructurar el código, este paradigma permite representar de forma clara a los elementos del juego, facilitando su mantenimiento y evolución.

Un ejemplo sencillo sería un **juego de aventuras** donde existe una **clase Personaje**. Esta clase funciona como plantilla para crear distintos tipos de personajes y está definida por:

- **Atributos (características)**

- **vida** → cantidad de energía o resistencia del personaje.
- **fuerza** → determina el poder de ataque.
- **nivel** → indica el progreso o experiencia alcanzada.

- **Métodos (acciones):**

- **atacar()** → permite golpear o dañar a un enemigo.
- **saltar()** → posibilita esquivar obstáculos o moverse en diferentes plataformas.
- **recuperarVida()** → aumenta la energía del personaje al usar pociones o descansando.

A partir de esta clase, se pueden crear **objetos (personajes específicos)**, como:

- **Guerrero:** vida alta, mucha fuerza, nivel inicial 1.
- **Magos:** vida moderada, fuerza mágica elevada, nivel inicial 1.
- **Arquero:** vida media, fuerza equilibrada, nivel inicial 1.

Cada uno tiene diferentes valores en sus atributos, pero todos comparten los mismos métodos para interactuar dentro del videojuego.

## 1.5 Diferencia con la programación estructurada

Mientras que la **programación estructurada** se enfoca principalmente en funciones y procedimientos, la Programación Orientada a Objetos (**POO**) organiza el software en objetos que integran tanto los datos (atributos) como las operaciones (métodos) que actúan sobre ellos.

Este enfoque permite que cada objeto sea una representación más fiel a los elementos del mundo real, facilitando la comprensión del código y la reutilización de componentes, mientras la **programación estructurada** divide el problema en bloques lógicos secuenciales, la **POO** promueve una visión más modular, escalable e intuitiva, alineada con la manera en que las personas perciben y representan la realidad.

## 2. Instalación y Configuración del Entorno de Desarrollo



### 2.1 ¿Qué es JDK?

El *Java Development Kit* (JDK) se define como un paquete de software diseñado para desarrolladores, el cual integra las herramientas esenciales para la creación y compilación de aplicaciones en Java, dentro de sus componentes principales se encuentran el compilador, el depurador, la *Java Virtual Machine* (JVM) para la ejecución del código, así como un conjunto de bibliotecas estándar que facilitan el desarrollo de programas robustos y eficientes.



Perales, I. (2021, 15 julio). *¿Qué es el JDK y el JRE en Java?* Israel Perales.

<https://www.israel-perales.com/que-es-el-jdk-y-el-jre-java/>

### 2.2 ¿Cómo instalar el IDE?

Para esto podemos utilizar tres páginas Eclipse, IntelliJ IDEA o NetBeans, pero para este curso utilizásemos eclipse.

- **¿Qué es un IDE?**

Un Entorno de Desarrollo Integrado, es una aplicación que reúne en un solo lugar todas las herramientas necesarias para escribir, probar y depurar código de manera eficiente, las herramientas que contiene un IDE pueden ser, editor de código, depurador, compilador o intérprete, automatización de tareas y una interfaz gráfica.

**Paso 1.** Ingresamos a la página de eclipse a través del siguiente enlace:

<https://www.eclipse.org/downloads/>



**Paso 2.** Al ingresar al sitio, se visualizará la página principal de descargas.

A screenshot of the Eclipse Foundation Downloads page. At the top, there's a navigation bar with links to Proyectos, Partidarios, Colaboraciones, Recursos, and La Fundación. Below the navigation, there's a large graphic of a computer monitor showing a terminal window with a download icon. To the right of the graphic, the text "Descargue la tecnología Eclipse adecuada para usted" is displayed. Below this, there's a yellow banner with the text "Descubra la IA en la Fundación Eclipse" and a link to "Más información". On the right side of the banner, there are "Más información" and "Suscribirse" buttons. At the bottom of the page, there are two sections: one for "Instale sus paquetes IDE de escritorio favoritos" featuring the Eclipse logo, and another for "El proyecto Eclipse Temurin® proporciona tiempos de ejecución óptimos para las aplicaciones Java más avanzadas".

**Paso 3.** A continuación, se debe desplazar hacia abajo hasta ubicar el botón de descarga.

A screenshot of the Eclipse Temurin download page. It features the Eclipse logo at the top. Below it, there's a section titled "Instale sus paquetes IDE de escritorio favoritos" with a "Más información" button and an orange "Descargar" button. Further down, there's a link "Descargar paquetes | ¿Necesitas ayuda?".

**Paso 4.** Una vez localizado, se hace clic en dicho botón para iniciar el proceso.

**Paso 5.** Despu s de presionar el bot n, aparecer  un mensaje con opciones de descarga.

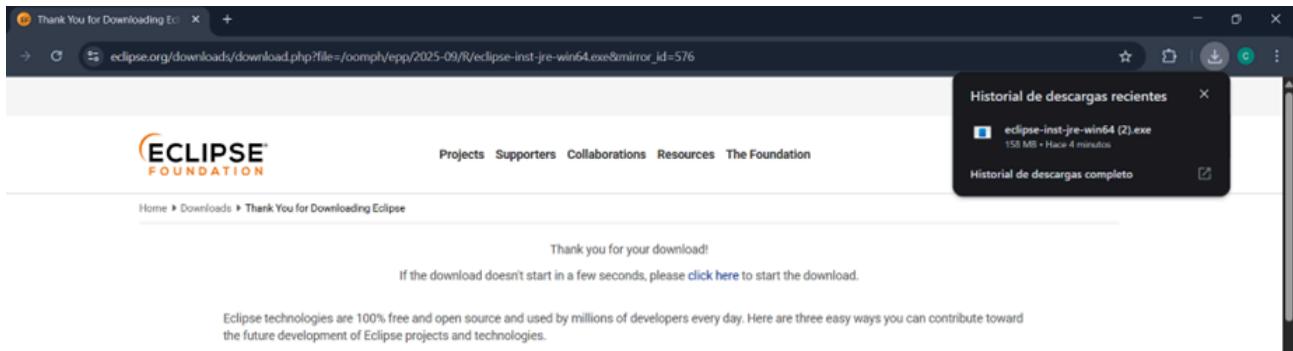


**Paso 6.** En este punto, se selecciona la opci n **Descargar x86\_64**, lo que redirige automáticamente a otra p gina.

**Paso 7.** En la nueva p gina, se mostrar  nuevamente el bot n de descarga. Al hacer clic en este, comenzar  la descarga del archivo de instalaci n.

A screenshot of the Eclipse Foundation download page for the x86\_64 version. The page header includes the Eclipse Foundation logo and navigation links for Proyectos, Partidarios, Colaboraciones, Recursos, and La Fundaci n. The main content area shows the URL "Hogar &gt; Descargas &gt; Descargas de Eclipse - Seleccionar un espejo". A note states: "Todas las descargas se proporcionan seg n los t rminos y condiciones del Acuerdo de usuario del software de Eclipse Foundation, a menos que se especifique lo contrario." Below this is a large orange "Descargar" button with a download icon. Further down, it says "Descargar desde: Brasil - C3SL - Universidad Federal de Paran  (https)" and "Archivo: eclipse-inst-jre-win64.exe SHA-512". There is also a link "&gt;&gt; Seleccionar otro espejo".

**Paso 8.** Una vez descargado, se debe acceder al archivo desde el navegador (normalmente en la parte superior derecha) o desde la carpeta de descargas del sistema.



**Paso 9.** Se da doble clic sobre el archivo descargado para iniciar el proceso de instalación.

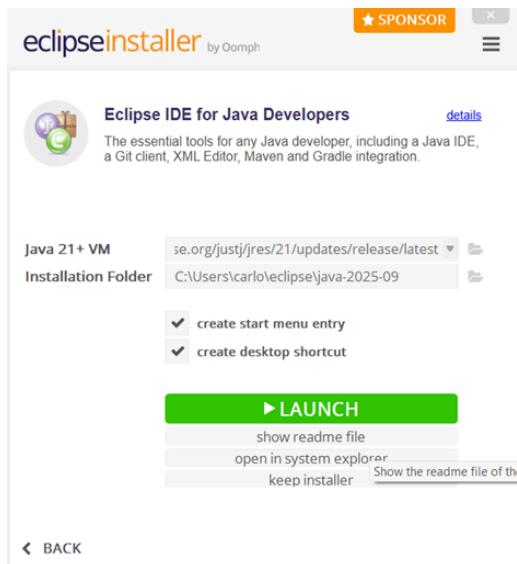
**Paso 10.** Aparecerá una ventana con diferentes opciones de instalación.



**Paso 11.** Se debe dar doble clic sobre la opción **Eclipse IDE for Java Developers**.

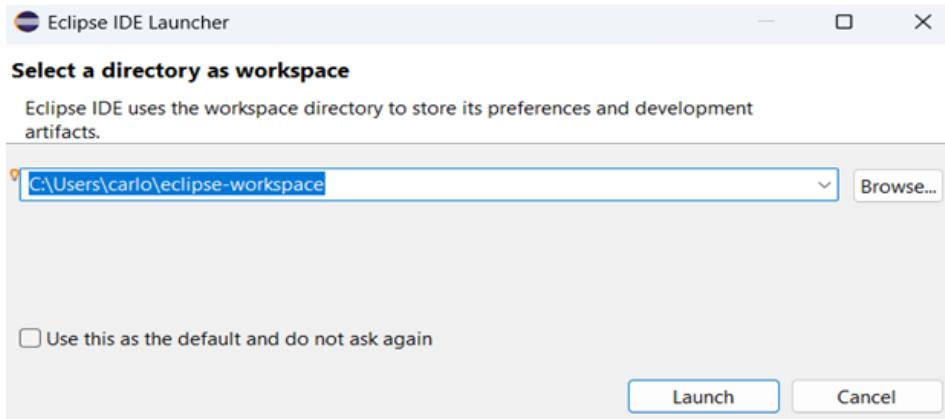
**Paso 12.** A continuación, se desplegará una nueva ventana en la que se hace clic en **Instalar** para iniciar la instalación del IDE.

**Paso 13.** Una vez finalizada la instalación, se mostrará la opción de ejecución.



**Paso 14.** Se selecciona el botón **Launch**.

**Paso 15.** Finalmente, al repetir la acción de **Launch**, se abrirá automáticamente el entorno de desarrollo Java (SUGERENCIA puedes vincularlos con alguna carpeta para organizar los proyectos).



## 2.3 ¿Ventajas de usar un IDE?

**Mejora la productividad:** Ya que, al venir integrado con un autocompletado de código, el cual te sugiere funciones, variables y estructuras mientras escribes.

**Menos errores y más claridad:** Ya que, al contar con un detector de errores en tiempo real, te avisará mientras escribes si hay fallos o inconsistencias.

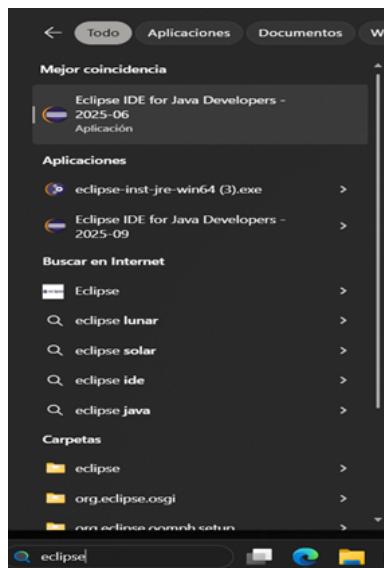
**Todo en un solo lugar:** Al incluir un editor, compilador, depurador y consola, no será necesario abrir varios programas, todo está incluido.

**Organización y estabilidad:** Al ser compatible con plugins puedes perdonar tu entorno según tus necesidades.

## 2.4 Creación de un Proyecto en Eclipse IDE

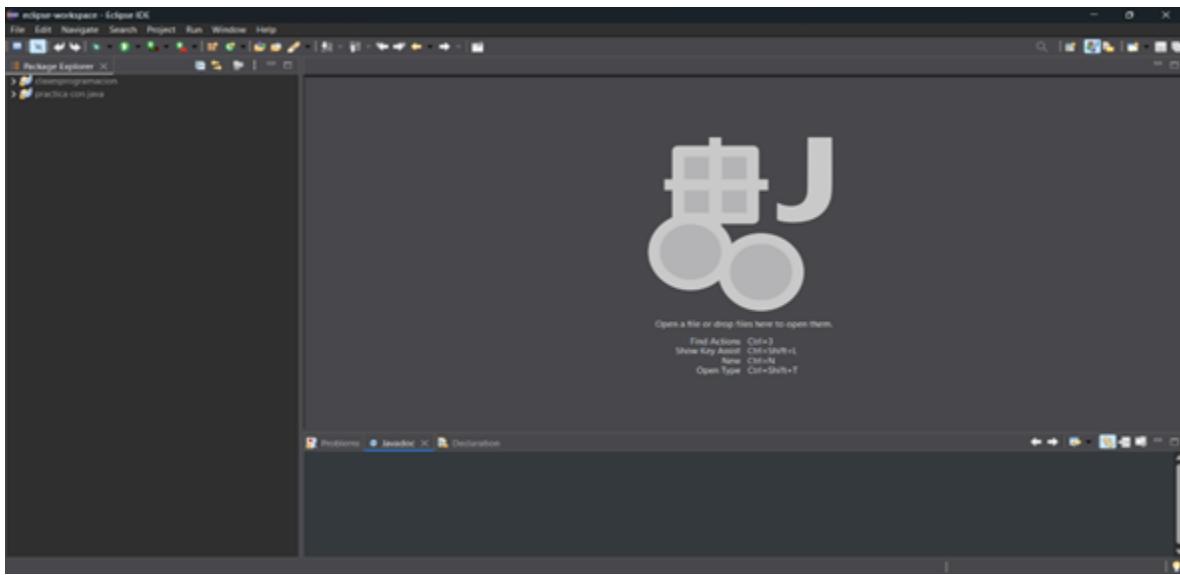
Antes de empezar a programar, el siguiente paso será crear nuestra primera carpeta (proyecto) en Java.

**Paso 1.** Abrir Eclipse: presionar **Windows + S** y escribir **Eclipse** en la barra de búsqueda.  
Dar doble clic sobre **Eclipse IDE** y automáticamente se abrirá Java.

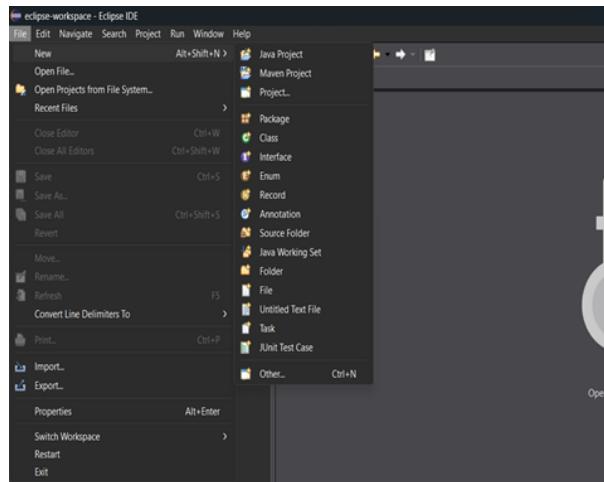


Daremos doble clic sobre Eclipse IDE y automáticamente se nos abrirá java.

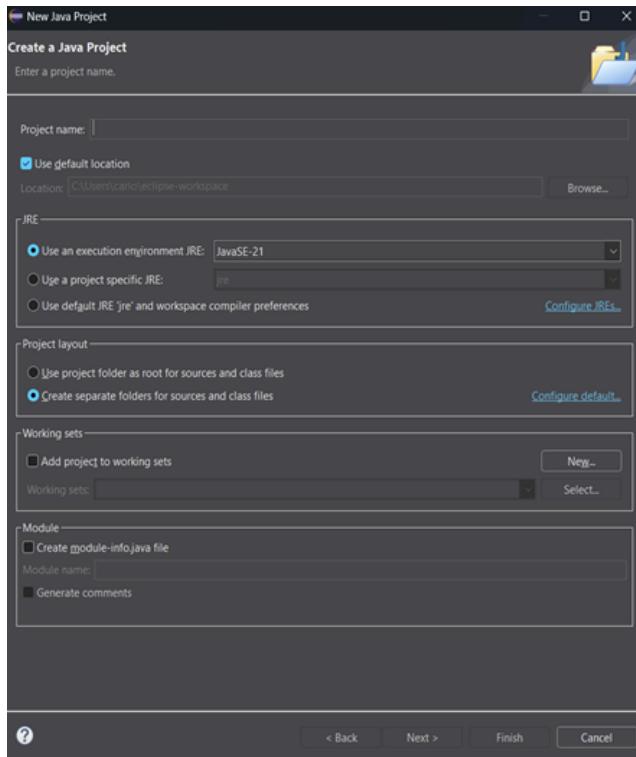
**Paso 2.** Una vez abierto, se visualizará la interfaz principal del IDE.



**Paso 3.** En la parte superior izquierda, dar clic en **File** → posicionar el cursor sobre **New** para ver más opciones.

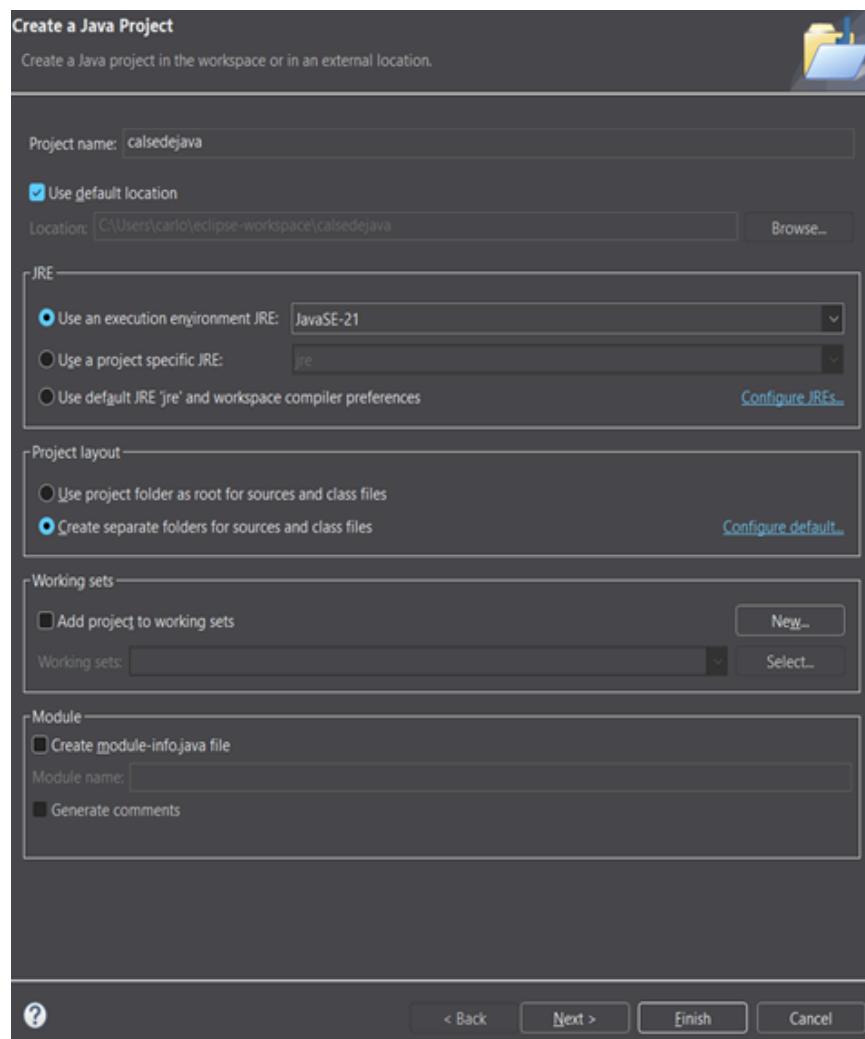


**Paso 4.** Seleccionar **Java Project**. Se abrirá una ventana emergente.

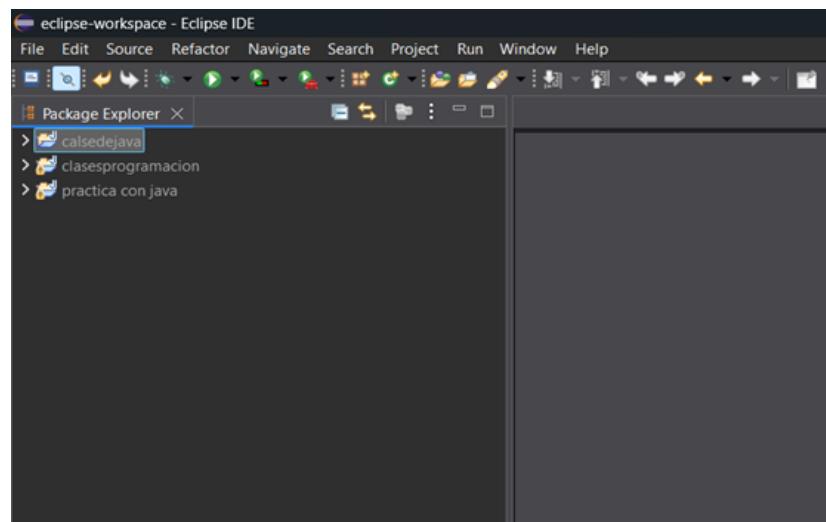


**Paso 5.** En la casilla **Project name**, asignar un nombre al proyecto (ejemplo: **clasedejava**), es importante que cuando asigne un nombre a una carpeta no dejes espacios. Después de asignado el nombre abajo hay un botón llamado **finish**, le daremos clic y automáticamente se nos creará la carpeta.

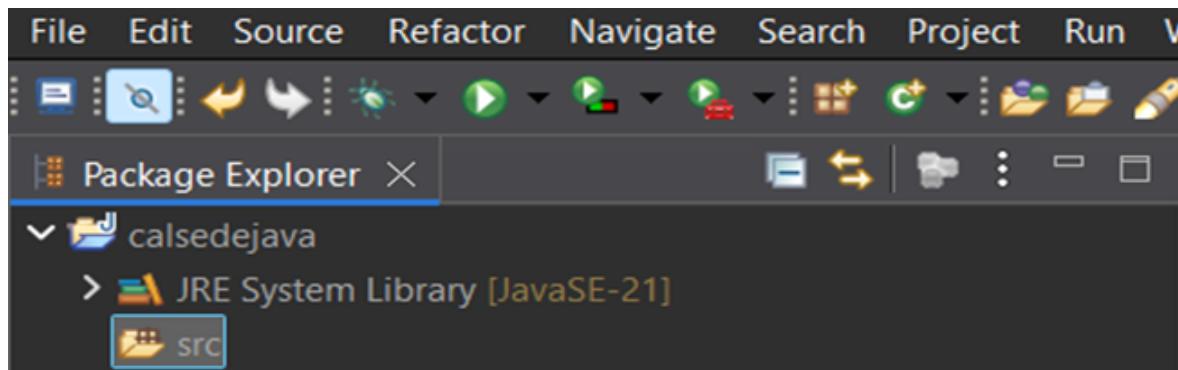
⚠ Importante: no se deben dejar espacios en los nombres.



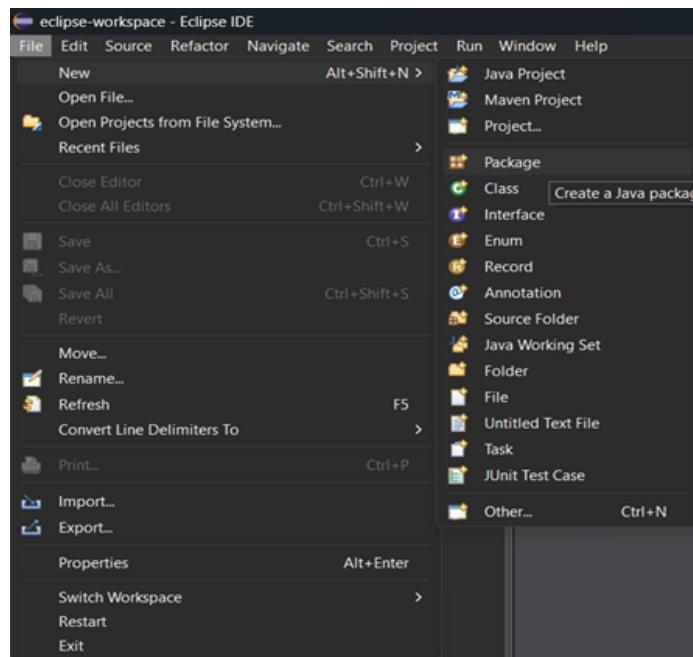
**Paso 6.** El proyecto creado aparecerá en la barra lateral izquierda.



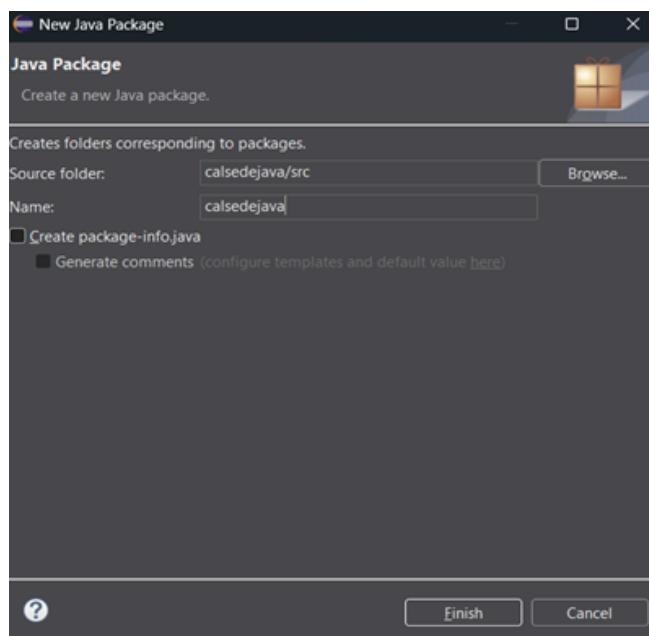
**Paso 7.** Dar doble clic sobre el proyecto creado. Allí se visualizarán dos carpetas. Seleccionar la carpeta **src**.



**Paso 8.** Volver a dar clic en **File → New → Package**.



**Paso 9.** En la ventana que aparece, dar clic en **Finish**.



Después de hacer lo anterior haremos un clásico hola mundo.

A screenshot of a Java code editor showing a simple "Hello World" program. The code is as follows:

```
1 package paquete1;
2
3 public class Hello {
4
5     public static void main(String[] args) {
6         System.out.println("Hello Mundo de Java");
7     }
8
9 }
```

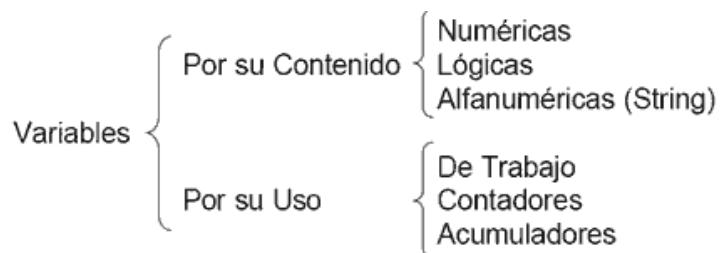
The code is color-coded: "package" and "String" are orange, "public", "class", "Hello", "main", "System", "out", "println", and "args" are green, and the braces and numbers are black.

### 3. Variables en la Programación Orientada a Objetos

#### 3.1 Concepto de variable

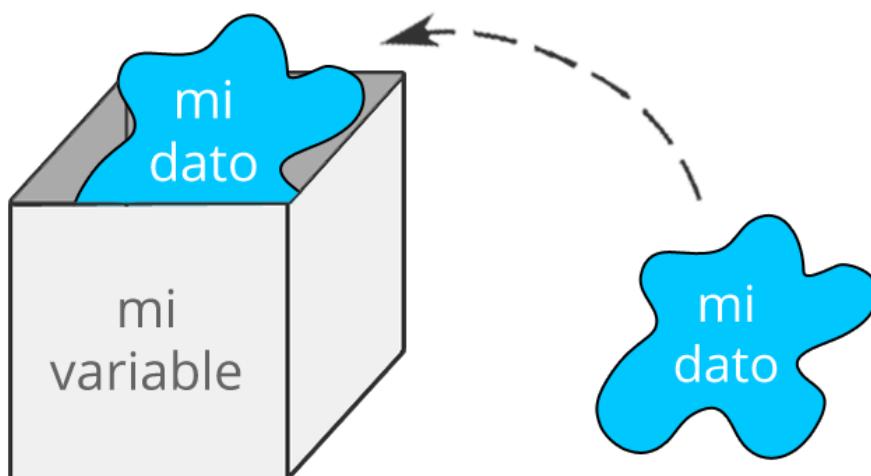
Una **variable** es un espacio de memoria identificado por un nombre único que se utiliza para **almacenar información temporal** dentro de un programa. Esta información puede ser de distintos tipos, como números, texto o valores lógicos, dependiendo del propósito.

Las variables son fundamentales en la programación, ya que permiten al programa **recordar y manipular datos** mientras se ejecuta. Por ejemplo, pueden almacenar el puntaje de un jugador, la vida de un personaje o la velocidad de un objeto en un videojuego.



Fuente: Desarrolloweb.com. Recuperado de  
<https://desarrolloweb.com/articulos/expresiones-instruccion-programacion.html>

Las variables facilitan la aplicación de **operaciones matemáticas, comparaciones lógicas y decisiones condicionales**, lo que convierte al programa en algo dinámico y adaptable según la interacción del usuario o el desarrollo de la lógica del sistema.



Fuente: TD Robótica. (s.f.). Categoría de Variables. Recuperado de  
<https://aprender.tdrobotica.co/courses/mblock-n1-introduccion/lecci%C3%B3n/categor%C3%ADa-de-variables/>

### 3.2 Tipos de variables en POO

En Java se utilizan principalmente los siguientes tipos de datos básicos:

#### Strings

- Representan **cadenas de texto** o caracteres alfanuméricos.
- Se escriben entre comillas dobles "" y se declaran usando la palabra clave **String**.
- **Ejemplo:**

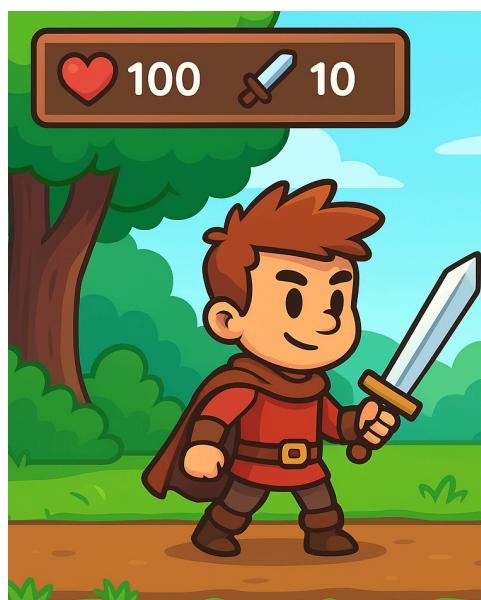
```
package EjemVideoJuego;

public class Variables {

    public static void main(String[] args) {

        // -----
        // Tipos de variables básicos
        // -----
        |

        // String: cadena de texto
        String nombrePersonaje = "Thor";
        System.out.println("Nombre del personaje: " + nombrePersonaje);
```



## Enteros (int)

- Almacenan **números enteros**, positivos o negativos, sin decimales.
- Se declaran con la palabra clave **int**.
- **Ejemplo:**

```
// int: número entero
int vida = 100;
System.out.println("Vida: " + vida);
```

## Flotantes (float)

- Almacenan **números decimales**.
- En Java, se debe agregar la letra **f** al final del número para indicar que es flotante.
- Se declaran con la palabra clave **float**.
- **Ejemplo:**

```
// float: número decimal
float velocidad = 5.5f;
System.out.println("Velocidad: " + velocidad);
```

## Booleanos (boolean)

- Representan **valores lógicos**: **true** (verdadero) o **false** (falso).
- Son útiles para condiciones y decisiones dentro del código.
- Se declaran con la palabra clave **boolean**.

- Ejemplo:

```
// boolean: valor lógico
boolean estaVivo = true;
System.out.println("¿Está vivo?: " + estaVivo);

System.out.println(); // Salto de línea para separar ejemplos
```

**Nota:** Conocer el tipo de dato de cada variable es importante, ya que determina cómo se puede manipular (operaciones matemáticas, lógicas, comparaciones, etc.).

### 3.3 Variables en videojuegos: ejemplos prácticos

En los videojuegos, las variables permiten representar propiedades y estados de los personajes, objetos o escenarios.

Ejemplos comunes:

```
// Nombre del personaje
String nombrePersonaje = "Thor";
```

```
// Vida del personaje
int vida = 100;
```

```
// Velocidad de movimiento
float velocidad = 5.5f;
```

```
// Estado de acción
boolean estaVivo = true;
```

```
// Imprimir información en consola
```

```
System.out.println("Personaje: " + nombrePersonaje);
System.out.println("Vida: " + vida);
System.out.println("Velocidad: " + velocidad);
System.out.println("¿Está vivo? " + estaVivo);
```

### 3.4 Variables en acción: ejemplo práctico en videojuegos

```
// -----
// Operaciones con variables
// -----
```

#### a. Incrementar vida

```
// Incrementar vida
vida += 20; // Suma 20 a la vida actual
System.out.println("Vida después de recibir vida extra: " + vida);
```

El fragmento muestra cómo modificar el valor de una variable utilizando el operador de asignación compuesto `+=`. En el contexto de un videojuego, se simula que el personaje recibe un bono de vida, actualizando su estado de manera dinámica.

#### b. Modificar velocidad

```
// Modificar velocidad
velocidad = velocidad + 2.0f; // Aumenta la velocidad
System.out.println("Velocidad aumentada: " + velocidad);
```

Aquí se ejemplifica cómo se pueden realizar operaciones matemáticas sobre variables tipo `float`. Se incrementa la velocidad de un personaje, demostrando cómo los atributos pueden cambiar durante la ejecución del juego.

#### c. Cambiar estado

```
// Cambiar estado
estaVivo = false; // El personaje muere
System.out.println("Estado actualizado: Esta vivo? " + (estaVivo ? "Si" : "No"));
```

Se ilustra el uso de variables booleanas para representar estados lógicos. En este caso, el personaje muere, mostrando cómo se puede controlar la lógica de juego mediante valores `true` o `false`.

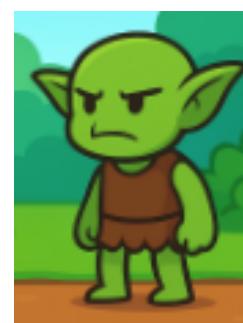
Se puede realizar un salto de línea de esta manera

```
System.out.println(); // Salto de línea
```

#### d. Ejemplo práctico: enemigo

```
// -----  
// Ejemplo practico tipo videojuego  
// -----
```

```
// Crear variables para un enemigo  
String nombreEnemigo = "Goblin";  
int vidaEnemigo = 50;  
float velocidadEnemigo = 3.2f;  
boolean enemigoVivo = true;
```



El bloque define un enemigo usando variables de distintos tipos, representando atributos típicos de un personaje en un videojuego: nombre, vida, velocidad y estado de vida.

### e. Mostrar información del enemigo

```
// Mostrar informacion en consola
System.out.println("Enemigo: " + nombreEnemigo);
System.out.println("Vida del enemigo: " + vidaEnemigo);
System.out.println("Velocidad del enemigo: " + velocidadEnemigo);
System.out.println("Enemigo vivo?: " + (enemigoVivo ? "Si" : "No"));

System.out.println(); // Salto de linea
```

Se demuestra cómo imprimir variables en consola y concatenar diferentes tipos de datos para presentar información completa de un objeto dentro del juego.

### f. Simulación de daño al enemigo

```
// Simulacion de daño al enemigo
int danio = 20;
vidaEnemigo -= danio; // Restar daño
System.out.println("Vida del enemigo despues del ataque: " + vidaEnemigo);
```

Se muestra cómo modificar la vida de un enemigo tras recibir daño, utilizando el operador `-=`. Esto refleja interacciones típicas de combate en videojuegos, donde los atributos cambian según las acciones.

### g. Verificar si el enemigo sigue vivo

```
// Verificar si el enemigo sigue vivo
if (vidaEnemigo <= 0) {
    enemigoVivo = false;
}
System.out.println("Enemigo vivo despues del ataque?: " + (enemigoVivo ? "Si" : "No"));
```

El bloque combina variables booleanas y estructuras condicionales para controlar la lógica del juego. Se verifica si el enemigo ha perdido toda su vida y actualiza su estado en consecuencia.

### 3.5 Buenas prácticas al usar variables

- **Nombres descriptivos:** Usar nombres claros y representativos del dato que almacenan, por ejemplo: `vidaJugador` en lugar de `v`. **Evitar espacios:** Los nombres de las variables no deben contener espacios.
- **Inicializar variables:** Siempre asignar un valor inicial para evitar errores en tiempo de ejecución.
- **Tipos adecuados:** Elegir el tipo de dato correcto según la información que se va a almacenar.
- **Constantes cuando sea necesario:** Si un valor no debe cambiar, usar `final` para declararlo como constante.



Salida de Consola:

```
Nombre del personaje: Thor
Vida: 100
Velocidad: 5.5
Esta vivo?: Si

Vida despues de recibir vida extra: 120
Velocidad aumentada: 7.5
Estado actualizado: Esta vivo? No

Enemigo: Goblin
Vida del enemigo: 50
Velocidad del enemigo: 3.2
Enemigo vivo?: Si

Vida del enemigo despues del ataque: 30
Enemigo vivo despues del ataque?: Si
```

## 4. Fundamentos de la Programación Orientada a Objetos

La Programación Orientada a Objetos (POO) se fundamenta en la creación de programas estructurados en torno a objetos, estos **objetos** no solo contienen datos, sino también las operaciones que pueden realizar, se logra un enfoque más natural e intuitivo para representar situaciones reales y virtuales, como las que se ven en el desarrollo de videojuegos.

### 4.1 Concepto de clase y objeto

En la Programación Orientada a Objetos, una **clase** es la **plantilla o modelo** que define cómo serán los objetos que se crean a partir de ella. Dentro de una clase se especifican:

Por otro lado, un **objeto** es una **instancia concreta** en la que **nace** de una clase, mientras la clase es como un “**molde**” o “**receta**”, el *objeto* es el “**producto final**” construido a partir de dicho molde.

#### → Ejemplo en la vida real

- **Clase:** Carro (es la plantilla).
  - **Atributos:** color, marca, modelo, año.
  - **Métodos:** arrancar(), acelerar(), frenar().
- **Objeto:** Un carro específico, por ejemplo:
  - Toyota Corolla, color rojo, modelo 2022.
  - Este carro en particular es **una instancia de la clase Carro**.

Con esto se entiende que la clase define la estructura general de lo que es un carro, pero cada carro real tendrá valores concretos (azul, rojo, Toyota, Renault, etc.), y todos compartirán los mismos comportamientos básicos.

## CÓDIGO:

```
package EjemCarro;

public class Main {

    // Clase Carro: representa el molde o plantilla
    static class Carro {
        String marca;
        String modelo;

        // Métodos: acciones del carro
        void arrancar() {
            System.out.println(marca + " " + modelo + " esta arrancando...");
        }

        void frenar() {
            System.out.println(marca + " " + modelo + " esta frenando...");
        }
    }

    public static void main(String[] args) {
        // Crear un objeto (instancia de Carro)
        Carro miCarro = new Carro();

        // Asignar valores a los atributos
        miCarro.marca = "Toyota";
        miCarro.modelo = "Corolla";
    }
}
```

```

// Usar los métodos

miCarro.arrancar();

miCarro.frenar();

}

}

```

**Creamos la Clase Carro con los atributos y metodos().**

```

package EjemCarro;

public class Main {

    // Clase Carro: representa el molde o plantilla
    static class Carro {
        String marca;
        String modelo;

        // Métodos: acciones del carro
        void arrancar() {
            System.out.println(marca + " " + modelo + " está arrancando...");
        }

        void frenar() {
            System.out.println(marca + " " + modelo + " está frenando...");
        }
    }
}

```

Nace el objeto de la Clase Carro

```

public static void main(String[] args) {
    // Crear un objeto (instancia de Carro)
    Carro miCarro = new Carro();

    // Asignar valores a los atributos
    miCarro.marca = "Toyota";
    miCarro.modelo = "Corolla";

    // Usar los métodos
    miCarro.arrancar();
    miCarro.frenar();
}
}

```

## 4.2 Atributos y métodos

- **Atributos (propiedades o variables):** describen las características de un objeto.

Los **atributos** representan las propiedades o variables de un objeto y permiten describir sus características, estado o información interna. Por ejemplo, en un videojuego, un personaje puede tener atributos como `vida`, `fuerza` o `nivel`.

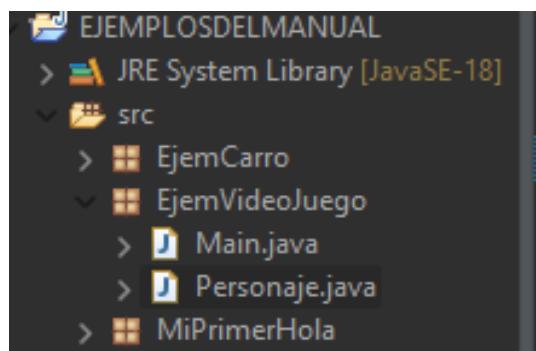
- **Métodos (funciones o acciones):** representan el comportamiento del objeto

Los **métodos**, por otro lado, definen las acciones o comportamientos que un objeto puede realizar. Siguiendo el ejemplo del personaje, los métodos podrían ser `atacar()`, `saltar()` o `recuperarVida()`.

→ Ejemplo en videojuegos:

- **Clase:** Personaje
- **Objeto:** Un personaje específico del juego (un guerrero o un mago).

Tenemos la siguiente Creacion de carpetas



/EJEMPLOSMANUAL/src/EjemVideoJuego/Personaje.java o Main.java

## Notas:

- Ambos archivos están en la carpeta EjemVideoJuego.
- Por eso **arriba de cada archivo** va la línea package EjemVideoJuego;.
- Desde Main puedes crear instancias de Personaje sin problema porque están en el mismo paquete.

## ARCHIVO CLASS Personaje.java

En esta clase se definen los atributos y métodos de un personaje dentro del videojuego. Se incluyen propiedades como el nombre y la vida, junto con acciones básicas como mostrar información, atacar y recibir daño

```
package EjemVideoJuego;

public class Personaje {
    // Atributos
    String nombre;
    int vida;

    // Método para mostrar información
    public void mostrarInfo() {
        System.out.println("El personaje es " + nombre + " con vida " + vida);
    }

    // Método para atacar
    public void atacar(String enemigo) {
        System.out.println(nombre + " ataca a " + enemigo + " con su espada.");
    }

    // Método para recibir daño
    public void recibirDanio(int puntos) {
        vida -= puntos;
        System.out.println(nombre + " ha recibido " + puntos + " de daño. Vida restante: " + vida);
    }
}
```

## ARCHIVO CLASS Main.java

En este archivo se crea un objeto de la clase Personaje y se utilizan sus métodos para simular interacciones, como atacar y recibir daño. De esta forma, se demuestra cómo instanciar una clase y aplicar sus comportamientos en un programa práctico.

```
package EjemVideoJuego;

public class Main {
    public static void main(String[] args) {
        // Crear un personaje
        Personaje guerrero = new Personaje();
        guerrero.nombre = "Thor";
        guerrero.vida = 100;

        // Usar los métodos
        guerrero.mostrarInfo();
        guerrero.atacar("Loki");
        guerrero.recibirDanio(20);
        guerrero.mostrarInfo();
    }
}
```

## 4.4 Encapsulamiento

El encapsulamiento es un principio de la Programación Orientada a Objetos que protege los atributos internos de un objeto para que no sean modificados directamente desde el exterior. En su lugar, se utilizan métodos públicos que controlan cómo se accede o modifica la información. Esto garantiza mayor seguridad, control y coherencia en el manejo de los datos.

### Ejemplo aplicado a los videojuegos:

La vida de un personaje no puede alterarse de forma directa, sino únicamente mediante métodos que representen acciones del juego, como recibir daño o curarse.

```

package EjemVideoJuego;

// Clase que representa a un personaje con vida encapsulada
class PersonajeEncapsulado {

    private int vida = 100;

    // Método público que permite modificar la vida de manera controlada
    // Aquí se resta la cantidad de vida cuando el personaje recibe daño
    public void recibirDaño(int cantidad) {
        vida -= cantidad;
    }

    // Método público para consultar la vida actual del personaje
    public int getVida() {
        return vida;
    }
}

// Clase principal para ejecutar el ejemplo de encapsulamiento
public class EjemploEncapsulamiento {
    public static void main(String[] args) {
        PersonajeEncapsulado p = new PersonajeEncapsulado();

        p.recibirDaño(30);

        System.out.println("Vida actual del personaje: " + p.getVida());
    }
}

```

### Salida de Consola:

```

Console X
<terminated> EjemploEncapsulamiento [Java Application]
Vida actual del personaje: 70

```

## 4.5 Herencia

La herencia permite que una clase (subclase) tome atributos y métodos de otra clase (superclase). Este mecanismo evita la duplicación de código y facilita la extensión de funcionalidades. Gracias a la herencia, se pueden crear diferentes tipos de personajes que comparten características básicas, pero que también agregan sus propias particularidades.

### Ejemplo aplicado a los videojuegos:

La clase **Personaje** es la base para crear otros personajes, como **Guerrero** o **Mago**, quienes heredan atributos comunes como el nombre y la vida, pero agregan sus propias habilidades.

```

package EjemVideoJuego;

// Clase base con atributos generales
class PersonajeBase {
    String nombre;
    int vida;
}

// Subclase que hereda de PersonajeBase
class Guerrero extends PersonajeBase {
    int fuerza; // Atributo adicional
}

public class EjemploHerencia {
    public static void main(String[] args) {
        // Crear un objeto Guerrero
        Guerrero g = new Guerrero();
        g.nombre = "Ares";
        g.vida = 120;
        g.fuerza = 80;

        // Mostrar atributos del guerrero
        System.out.println("Guerrero: " + g.nombre + " - Vida: " + g.vida +
                           " - Fuerza: " + g.fuerza);
    }
}

```

### Salida de Consola:

```

Console X
<terminated> EjemploHerencia [Java Application] C:\Pro...
Guerrero: Ares - Vida: 120 - Fuerza: 80

```

## 4.6 Polimorfismo

El polimorfismo permite que un mismo método tenga diferentes comportamientos según el objeto que lo utilice. Este principio hace que el código sea más flexible y extensible, ya que diferentes clases pueden redefinir métodos heredados para adaptarlos a su propia lógica.

### Ejemplo aplicado a los videojuegos:

El método **atacar()** se ejecuta de forma distinta en cada tipo de personaje: un guerrero realiza un ataque físico, mientras que un mago lanza un hechizo.

```

package EjemVideoJuego;

// Clase base con un método genérico
class PersonajePoli {
    void atacar() {
        System.out.println("El personaje ataca.");
    }
}

// Subclase Guerrero que redefine atacar()
class GuerreroPoli extends PersonajePoli {
    @Override
    void atacar() {
        System.out.println("El guerrero ataca con su espada.");
    }
}

// Subclase Mago que redefine atacar()
class MagoPoli extends PersonajePoli {
    @Override
    void atacar() {
        System.out.println("El mago lanza un hechizo.");
    }
}

public class EjemploPolimorfismo {
    public static void main(String[] args) {
        // Se crean personajes de distinto tipo
        PersonajePoli p1 = new GuerreroPoli();
        PersonajePoli p2 = new MagoPoli();

        // Cada objeto ejecuta atacar() de forma distinta
        p1.atacar(); // Guerrero
        p2.atacar(); // Mago
    }
}

```

Salida de Consola:

```

Console X
<terminated> EjemploPolimorfismo [Java Application]
El guerrero ataca con su espada.
El mago lanza un hechizo.

```

#### 4.7 Abstracción

La abstracción consiste en enfocarse en lo esencial de un objeto y ocultar los detalles innecesarios. Este principio se implementa con **clases abstractas** e **interfaces**, que definen qué acciones deben realizar los objetos, sin especificar cómo se implementan. Esto facilita la organización del código y promueve la reutilización.

## Ejemplo aplicado a los videojuegos:

Todos los personajes pueden atacar, pero cada uno implementa su ataque de forma diferente (físico, mágico o a distancia).

```
package EjemVideoJuego;

// Clase abstracta que define un método atacar sin implementarlo
abstract class PersonajeAbs {
    abstract void atacar();
}

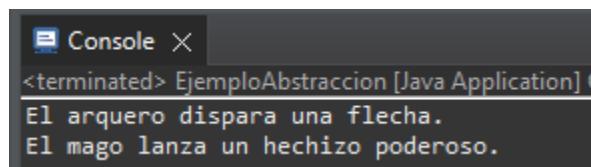
// Subclase Arquero que implementa el método atacar
class Arquero extends PersonajeAbs {
    @Override
    void atacar() {
        System.out.println("El arquero dispara una flecha.");
    }
}

// Subclase Mago que implementa el método atacar
class MagoAbs extends PersonajeAbs {
    @Override
    void atacar() {
        System.out.println("El mago lanza un hechizo poderoso.");
    }
}

public class EjemploAbstraccion {
    public static void main(String[] args) {
        // Crear objetos de tipo PersonajeAbs usando las subclases
        PersonajeAbs a = new Arquero();
        PersonajeAbs m = new MagoAbs();

        // Llamar al método atacar de cada objeto
        a.atacar();
        m.atacar();
    }
}
```

Salida de Consola:



The screenshot shows a terminal window titled "Console X" with the following text:  
<terminated> EjemploAbstraccion [Java Application] 0  
El arquero dispara una flecha.  
El mago lanza un hechizo poderoso.

**5.titulo del tema de carlos del documento de el (EMANUEL).**

**6.podemos colocar todo lo que ahora si llevamos en clase (CARLOS).**