· Tipos de datos y Clases Wrapper en Java

### **Tutorial Interactivo: Tipos de datos y Clases Wrapper en Java**

¡Bienvenido al tutorial interactivo! En este recorrido aprenderás sobre los **tipos de datos** en Java y las **clases Wrapper**, fundamentales para trabajar con valores básicos y su manipulación en programación orientada a objetos.

### **Objetivos del tutorial:**

1. Comprender los tipos de datos en Java.
2. Conocer las clases Wrapper y su utilidad.
3. Aplicar los conocimientos en ejercicios prácticos.

## **1. Tipos de datos en Java**

### **¿Qué son los tipos de datos?**

En Java, los **tipos de datos** determinan qué tipo de valor puede almacenar una variable y cuánta memoria ocupará. Hay dos categorías principales:

* **Tipos primitivos**: Datos simples, no objetos.
* **Tipos de referencia**: Objetos o instancias de clases.

### **Principales tipos primitivos:**

1. **byte** (1 byte)
2. **short** (2 bytes)
3. **int** (4 bytes)
4. **long** (8 bytes)
5. **float** (4 bytes, punto flotante)
6. **double** (8 bytes, punto flotante)
7. **char** (2 bytes, carácter Unicode)
8. **boolean** (1 bit, verdadero/falso)

#### **Ejemplo básico:**

java

Copiar código

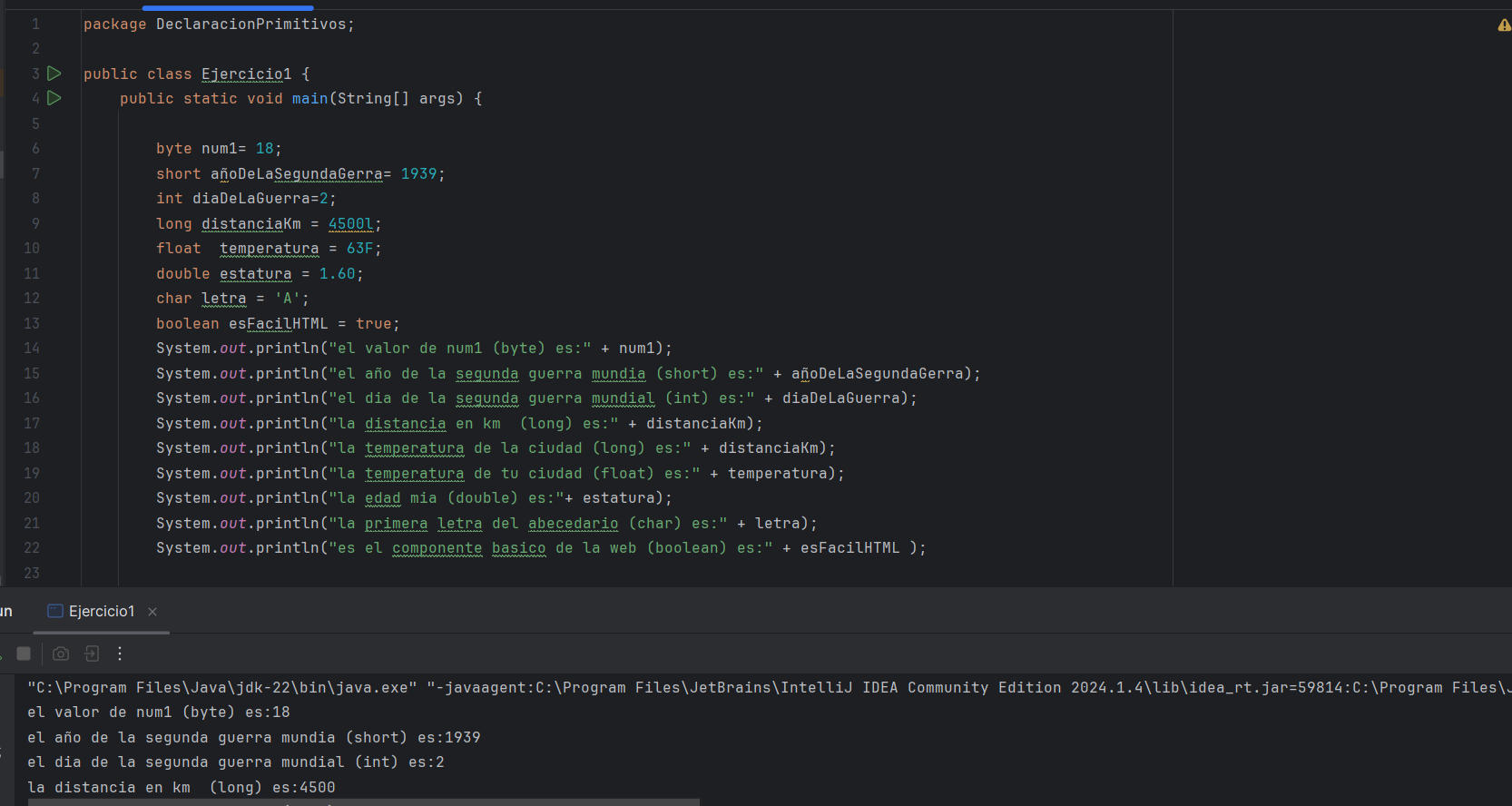
int numero = 10;

char letra = 'A';

boolean esVerdadero = true;

### **Ejercicio 1:**

1. Declara una variable de cada uno de los tipos de datos primitivos.
2. Asigna valores a cada variable.
3. Imprime los valores en la consola.



**Pregunta:** ¿Por qué crees que Java necesita diferentes tamaños para almacenar números enteros?

Se necesitan tamaños para almacenar números enteros o decimales para optimizar el uso de memoria y eficiencia según su contexto.

## **2. Clases Wrapper (envolventes)**

### **¿Qué son las clases Wrapper?**

En Java, cada tipo de dato primitivo tiene una **clase envolvente (Wrapper)** que lo convierte en un objeto. Esto es útil cuando necesitamos que un valor primitivo se comporte como un objeto, por ejemplo, al trabajar con colecciones como ArrayList que solo admiten objetos.

| **Tipo primitivo** | **Clase Wrapper** |
| --- | --- |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |
| char | Character |
| boolean | Boolean |

#### **Ejemplo:**

java

Copiar código

int numero = 10;

Integer numeroWrapper = Integer.valueOf(numero); // Conversión a Integer (Clase Wrapper)

#### **Autoboxing y Unboxing:**

* **Autoboxing**: Conversión automática de un valor primitivo en su correspondiente clase Wrapper.
* **Unboxing**: Conversión automática de una clase Wrapper a su valor primitivo.

java

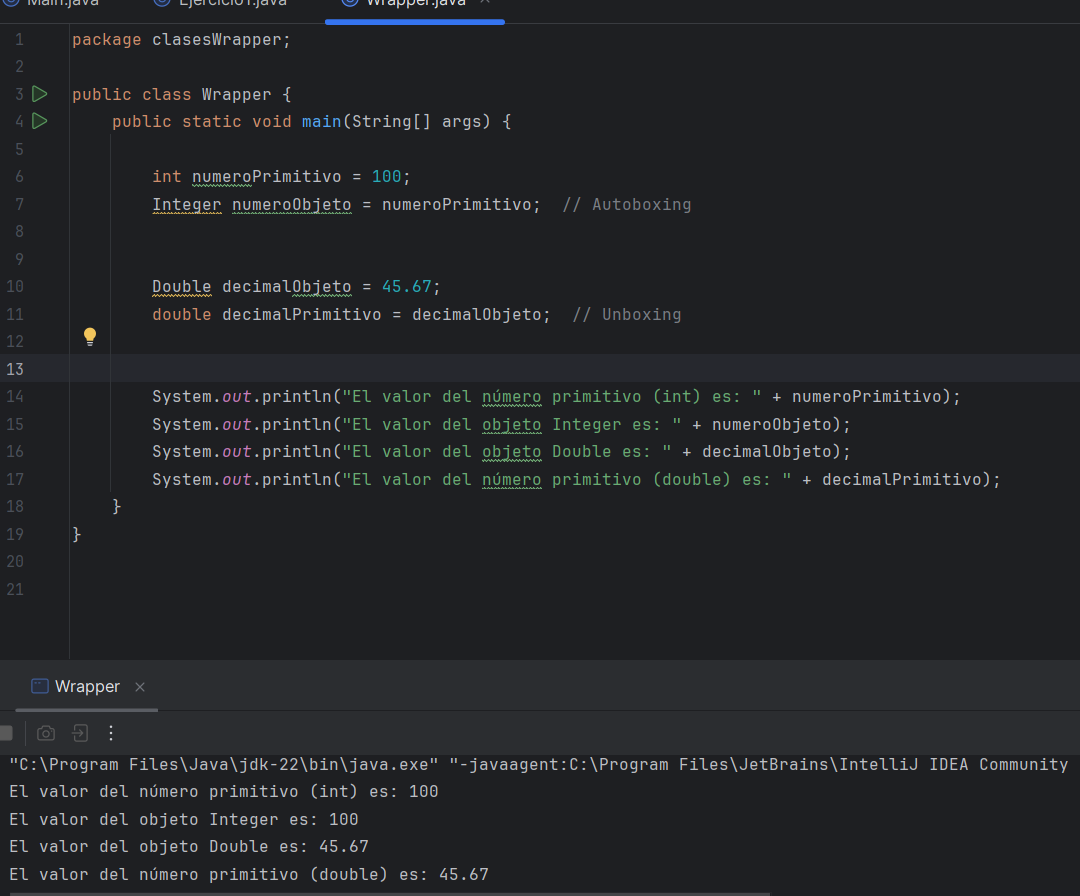
Copiar código

Integer num = 5; // Autoboxing: int a Integer

int primitivo = num; // Unboxing: Integer a int

### **Ejercicio 2:**

1. Declara un int y conviértelo a su clase Integer usando autoboxing.
2. Convierte un Double en su valor primitivo double usando unboxing.
3. Imprime los resultados.



**Pregunta:** ¿Qué ventajas crees que ofrecen las clases Wrapper en comparación con los tipos primitivos?

Las clases Wrapper permiten almacenar tipos primitivos en estructuras de datos que solo admiten objetos, como las colecciones (ArrayList, HashMap, etc.)

## **3. Aplicación práctica de Clases Wrapper**

### **Ejercicio práctico 3: Listado de enteros**

Ahora que conoces las clases Wrapper, es momento de aplicarlas en un entorno más práctico.

1. Crea un ArrayList de enteros utilizando la clase Integer.
2. Agrega cinco números enteros al ArrayList.
3. Suma todos los valores del ArrayList utilizando un bucle for.

#### **Código sugerido:**

java

Copiar código

import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

ArrayList<Integer> numeros = new ArrayList<>();

// Añadir elementos

numeros.add(5);

numeros.add(10);

numeros.add(20);

numeros.add(25);

numeros.add(30);

// Sumar los valores

int suma = 0;

for (Integer num : numeros) {

suma += num; // Unboxing ocurre aquí

}

System.out.println("La suma de los números es: " + suma);

}

}

### **Reflexión:**

¿Por qué es necesario utilizar la clase Integer en lugar de int al trabajar con colecciones como ArrayList?

Al trabajar con colecciones como ArrayList radica en que las colecciones solo admiten **objetos**, y los tipos primitivos no son objetos.

## **4. Resumen y desafíos finales**

### **Resumen:**

* Los **tipos de datos primitivos** son la base del almacenamiento de información en Java.
* Las **clases Wrapper** permiten que los valores primitivos se comporten como objetos.
* **Autoboxing** y **Unboxing** hacen que las conversiones entre tipos primitivos y clases Wrapper sean automáticas y sencillas.

### **Desafíos adicionales:**

1. Crea una clase que almacene un valor boolean y su respectiva clase Boolean. Realiza operaciones lógicas con estos valores.
2. Implementa un programa que convierte un array de números primitivos (int[]) en una lista de objetos Integer utilizando autoboxing.

### **¡Felicitaciones!**

Has completado el tutorial interactivo sobre **Tipos de datos y Clases Wrapper en Java**. Ahora tienes una base sólida para trabajar con datos en Java y convertirlos entre primitivos y objetos.

## **Estructuras de Control en Java**

### **Introducción**

¡Bienvenido! Hoy aprenderás sobre **estructuras de control** en Java, un concepto clave en la programación. Las estructuras de control permiten que nuestros programas tomen decisiones y repitan acciones basadas en condiciones específicas.

Este tutorial está diseñado para que aprendas de manera activa. Te guiaré paso a paso con ejemplos, preguntas y ejercicios para que pongas en práctica lo que aprendes en tiempo real.

### **Objetivo**

Al finalizar este tutorial, podrás:

* Entender los conceptos de las estructuras de control.
* Escribir y aplicar condicionales (if, else, switch).
* Implementar bucles (for, while, do-while).
* Resolver problemas prácticos que involucran estructuras de control.

### **1. Condicionales en Java**

Las condicionales nos permiten tomar decisiones en el código. Las más comunes en Java son if, else, y else if.

#### **Concepto clave:**

* if: Evalúa una condición. Si es verdadera, ejecuta el bloque de código.
* else: Se ejecuta si la condición del if es falsa.
* else if: Evalúa una nueva condición si la primera fue falsa.

#### **Ejemplo:**

java

Copiar código

int numero = 10;

if (numero > 5) {

System.out.println("El número es mayor que 5");

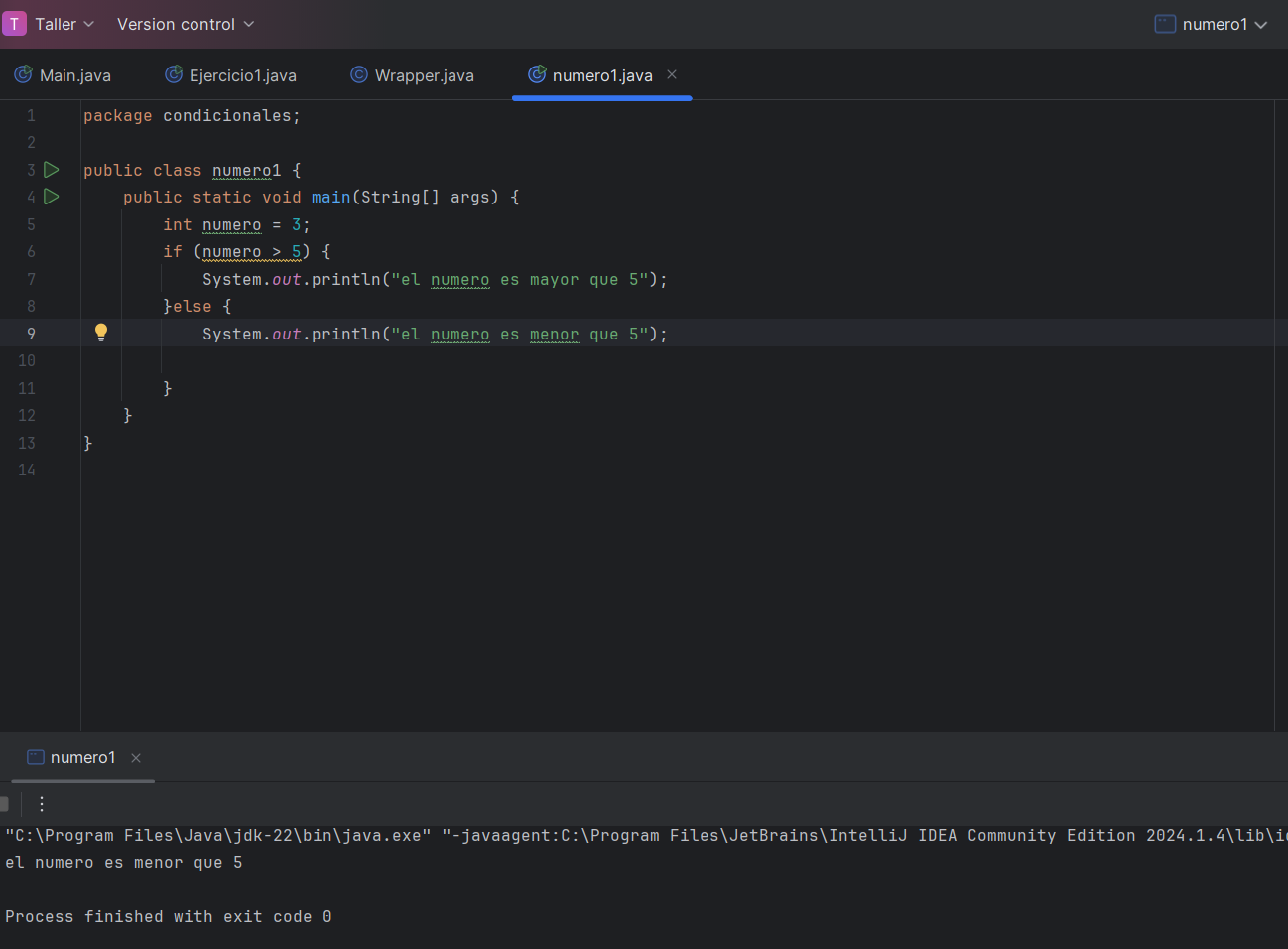
} else {

System.out.println("El número es menor o igual a 5");

}

#### **Pregunta:**

¿Qué crees que imprimirá este código si numero = 3? (Escríbelo y pruébalo).



### **Tarea interactiva 1:**

Escribe un programa en Java que pregunte la edad del usuario y:

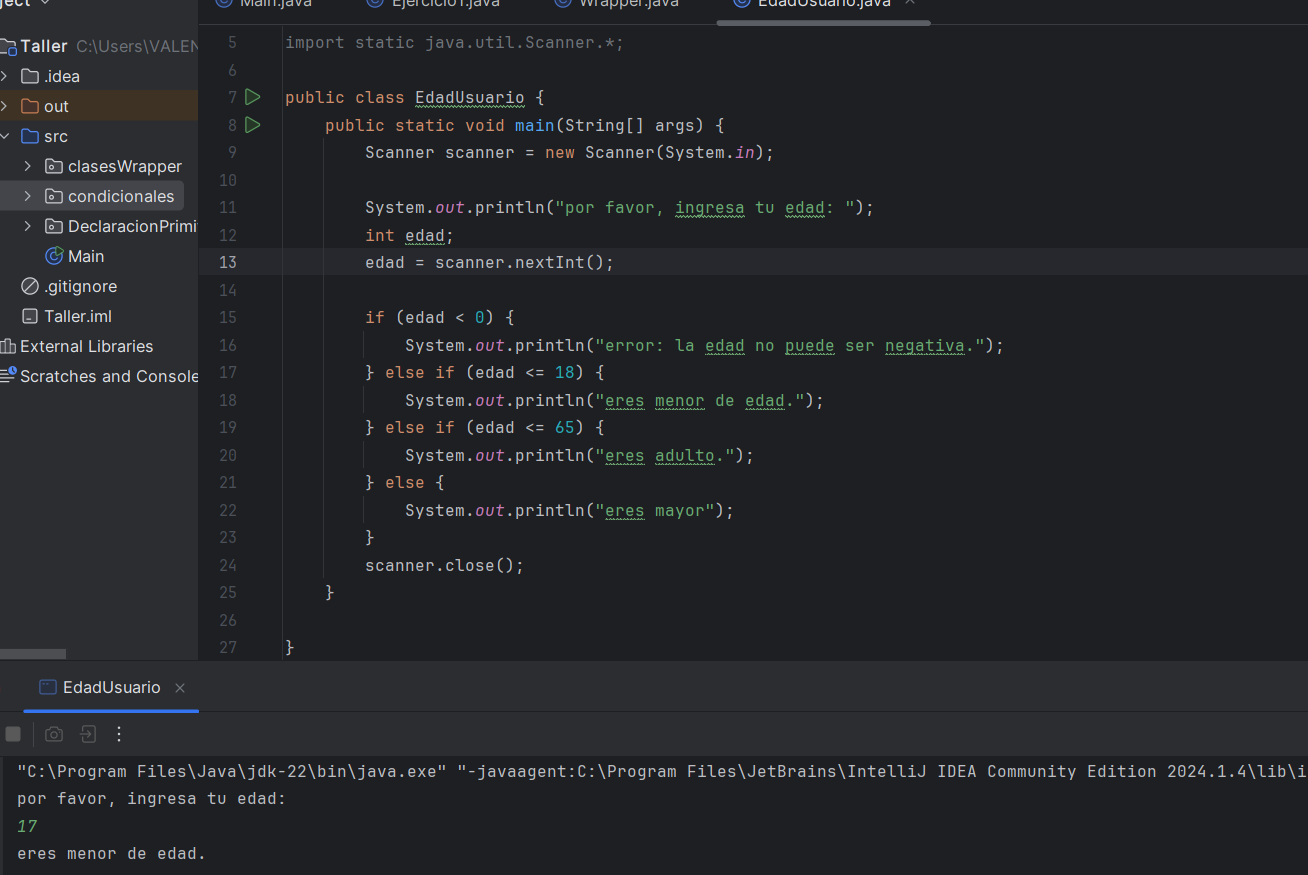
* Si tiene menos de 18 años, imprime "Eres menor de edad".
* Si tiene entre 18 y 65 años, imprime "Eres adulto".
* Si tiene más de 65 años, imprime "Eres mayor".

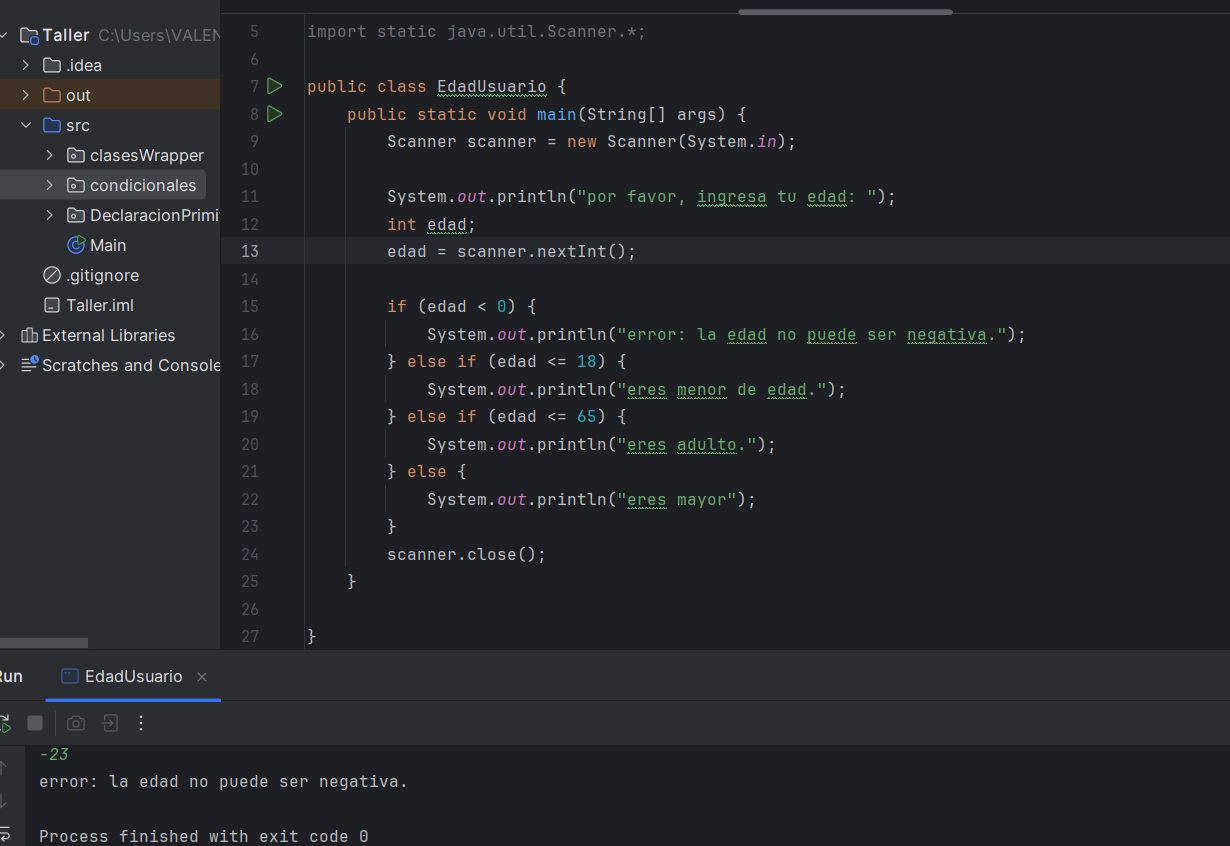
#### **Pista:**

Utiliza if, else if y else.

#### **Revisa tu código:**

¿Funciona correctamente? ¿Qué sucede si ingresas una edad negativa? ¿Cómo podrías manejar este caso? Intenta agregar un bloque adicional para verificar si la entrada es válida.





### **2. La estructura switch en Java**

El switch es otra estructura de control que se utiliza para seleccionar entre varias opciones. Es ideal cuando tienes varias posibles condiciones para un valor único.

#### **Concepto clave:**

* switch: Evalúa una variable y ejecuta el bloque de código correspondiente a la opción que coincide.

#### **Ejemplo:**

java

int dia = 2;

switch (dia) {

case 1:

System.out.println("Lunes");

break;

case 2:

System.out.println("Martes");

break;

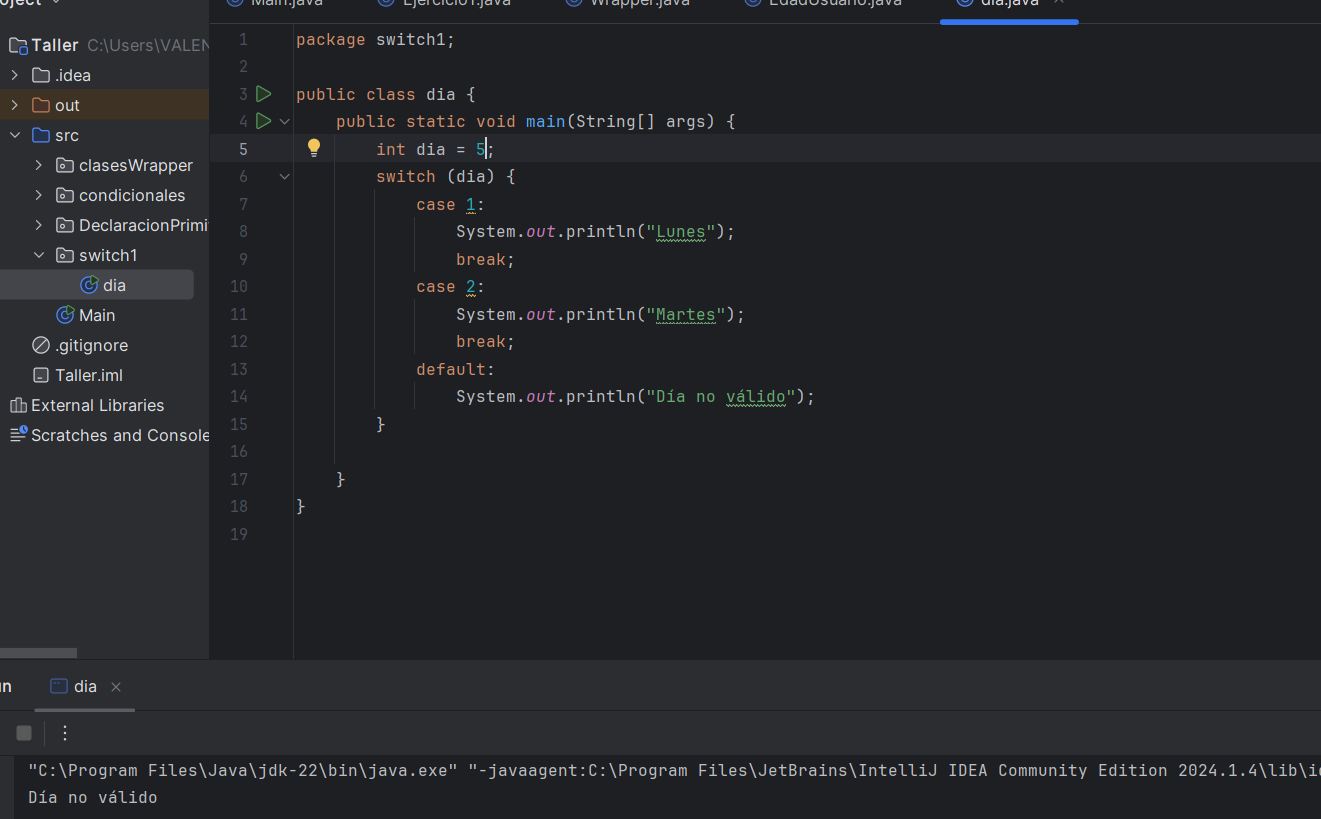
default:

System.out.println("Día no válido");

}

#### **Pregunta:**

¿Qué crees que imprimirá el código si dia = 5?

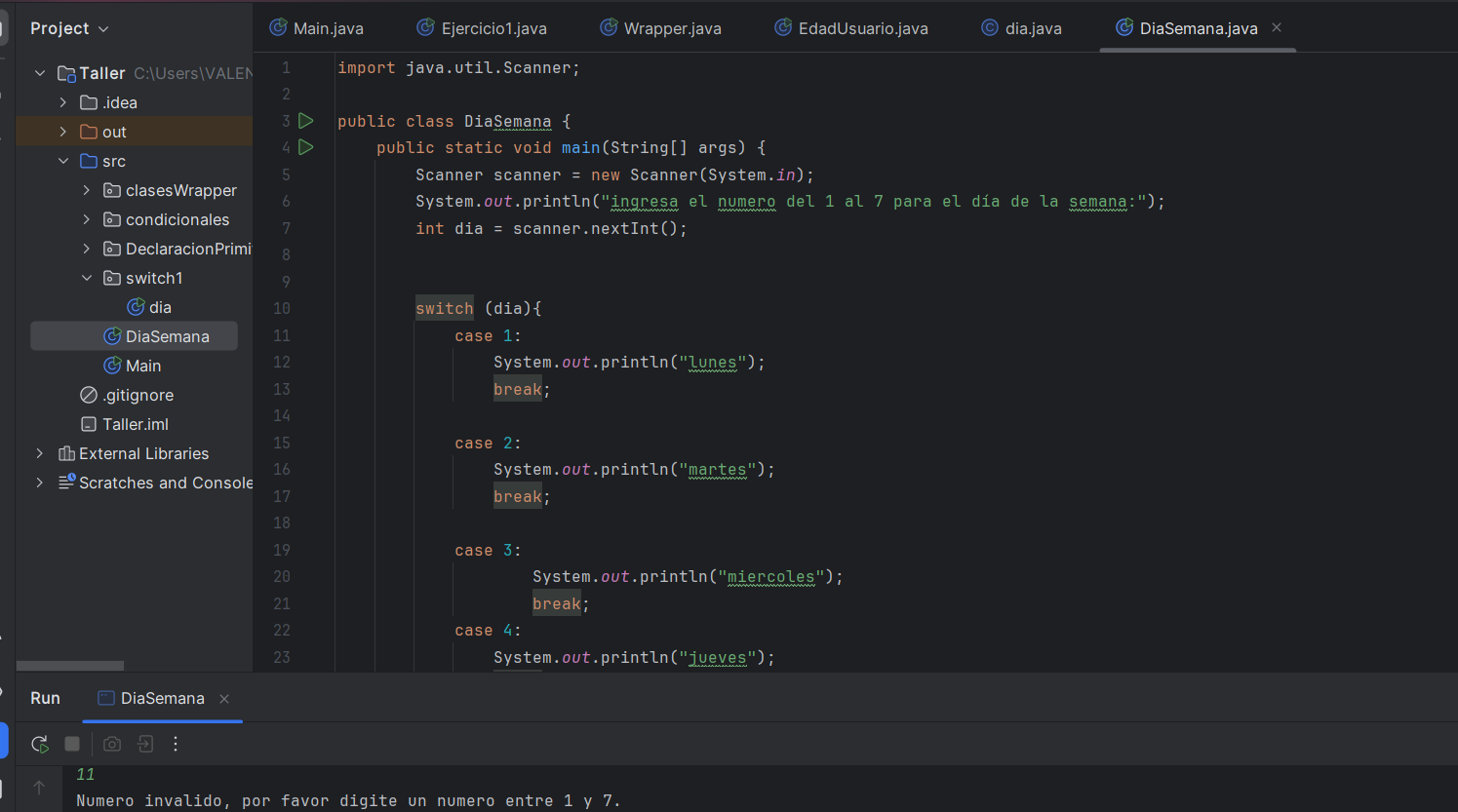


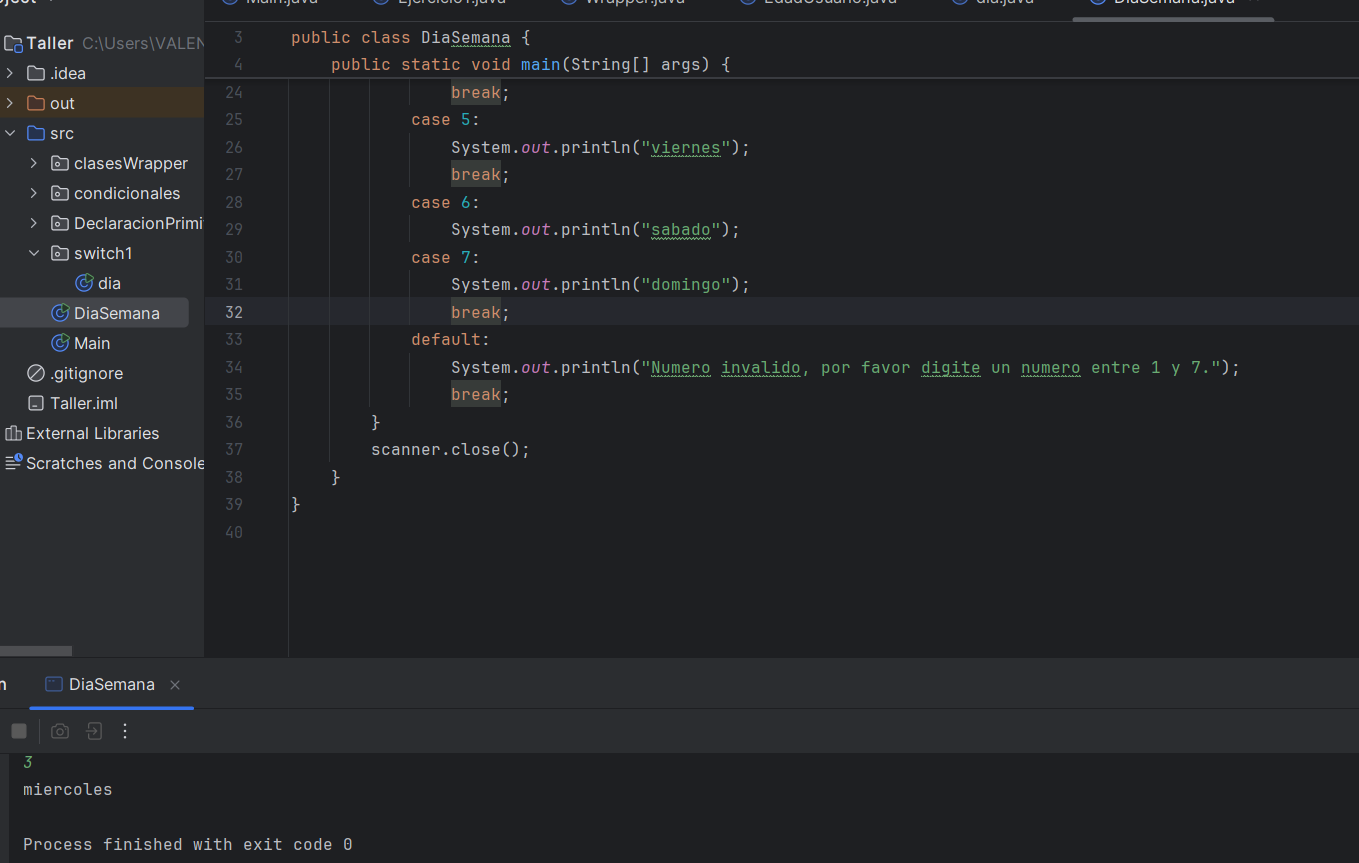
### **Tarea interactiva 2:**

Escribe un programa que pregunte al usuario el día de la semana (1-7) y utilice un switch para imprimir el nombre del día.

#### **Pista:**

No olvides agregar un default para manejar el caso en que el número no esté entre 1 y 7.





### **3. Bucles en Java**

Los bucles permiten repetir un bloque de código varias veces. Java tiene tres tipos principales: for, while, y do-while.

#### **3.1. El bucle for**

Se usa cuando sabes cuántas veces quieres repetir un bloque de código.

#### **Concepto clave:**

* for: Tiene tres partes: inicialización, condición, y actualización.

#### **Ejemplo:**

java

Copiar código

for (int i = 0; i < 5; i++) {

System.out.println("Iteración " + i);

}

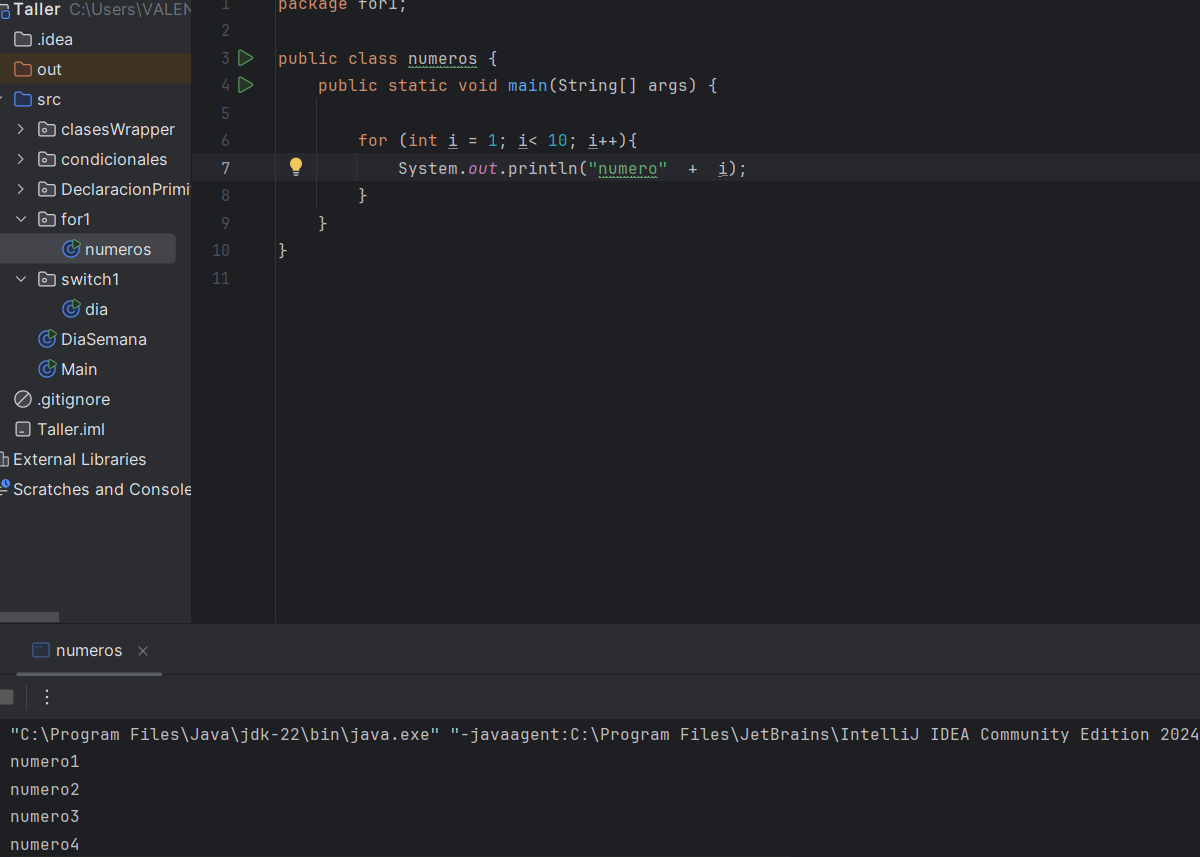
#### **Pregunta:**

¿Cuántas veces se imprimirá "Iteración"?

5 veces va a imprimir esa iteración.

### **Tarea interactiva 3:**

Escribe un programa que imprima los números del 1 al 10 utilizando un for.

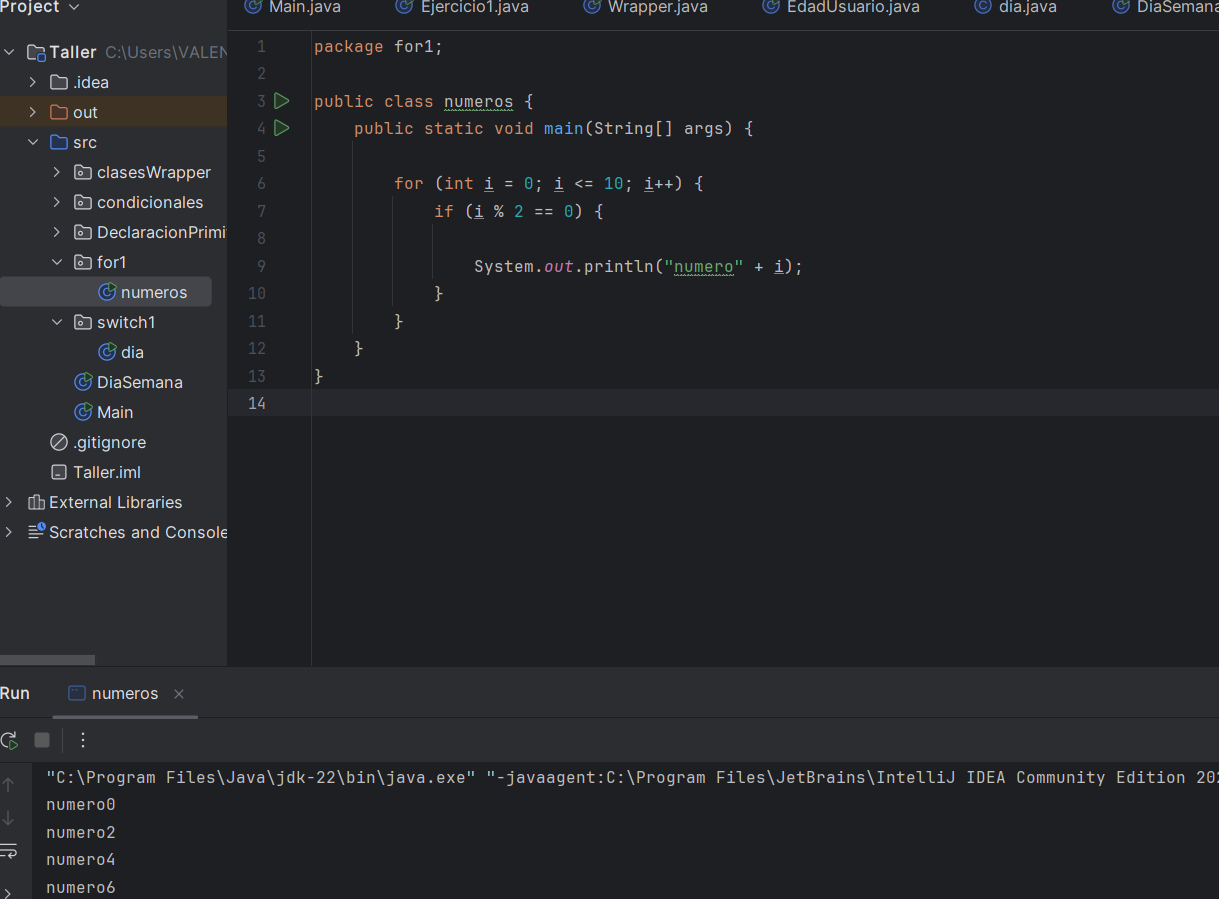


#### **Pista:**

Cambia el valor inicial y la condición del bucle.

#### **Desafío adicional:**

¿Qué sucede si quieres que se impriman solo los números pares? Intenta modificar el código para lograr esto.



#### **3.2. El bucle while**

Se utiliza cuando no sabes cuántas veces se ejecutará el bloque de código, pero quieres repetirlo mientras una condición sea verdadera.

#### **Concepto clave:**

* while: Ejecuta el código mientras la condición sea true.

#### **Ejemplo:**

int contador = 0;

while (contador < 5) {

System.out.println("Contador: " + contador);

contador++;

}

#### **Pregunta:**

¿Qué crees que pasará si se olvida de incrementar el valor de contador dentro del bucle?

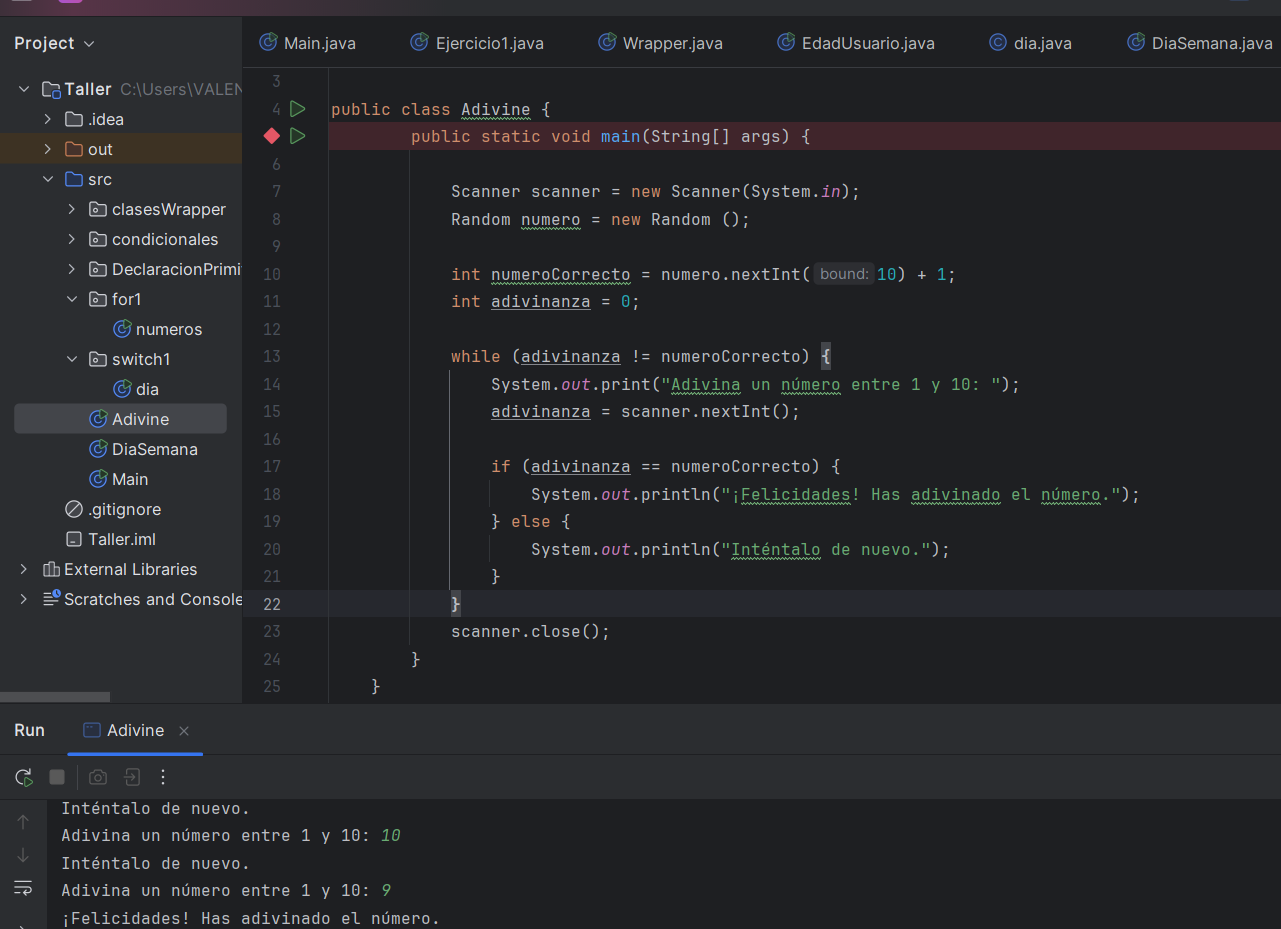
Esto provocará que el ciclo se repita indefinidamente, creando un bucle infinito

### **Tarea interactiva 4:**

Escribe un programa que pida al usuario que adivine un número (entre 1 y 10) hasta que lo adivine correctamente.

#### **Pista:**

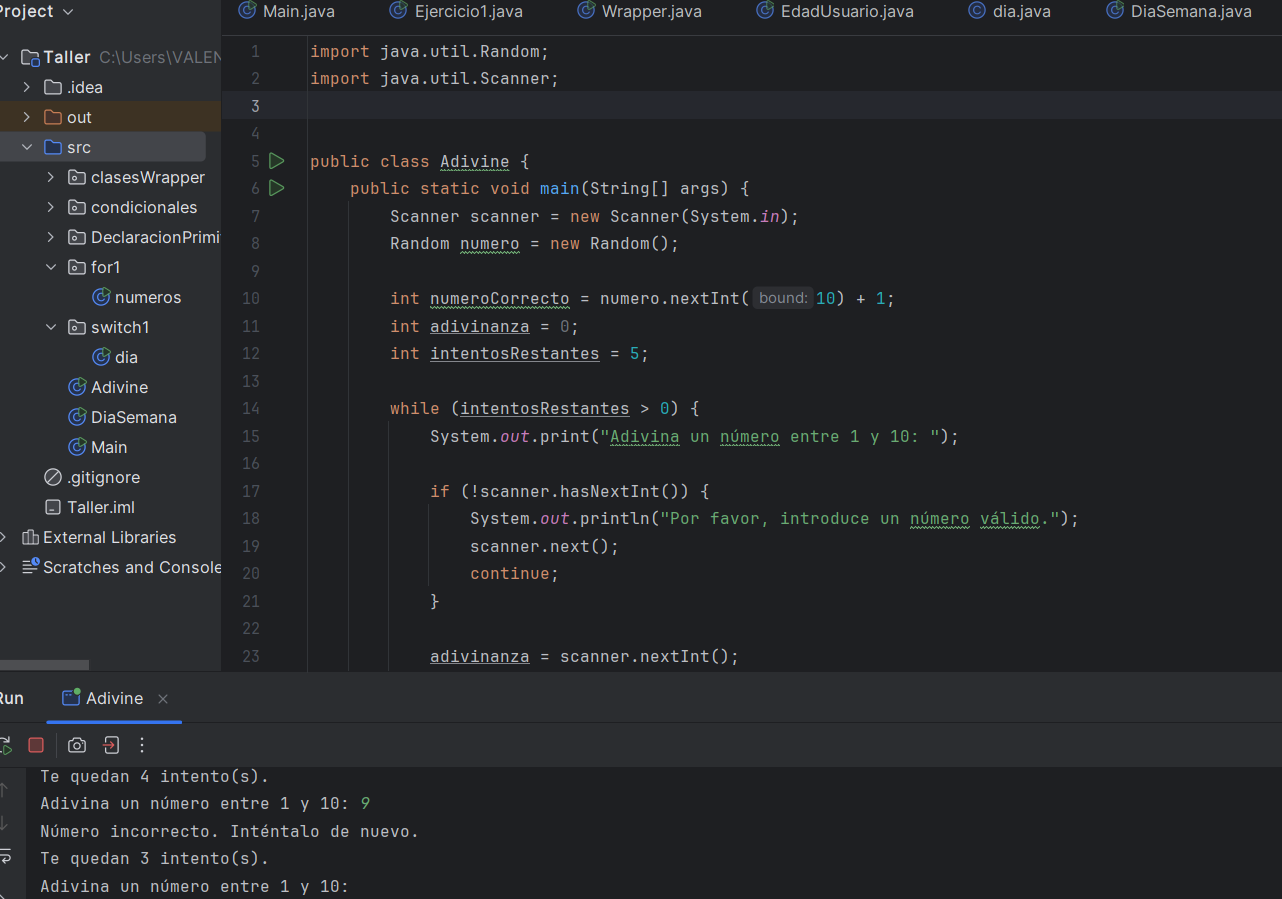
Utiliza un while para mantener el bucle hasta que la condición de "número correcto" sea verdadera.

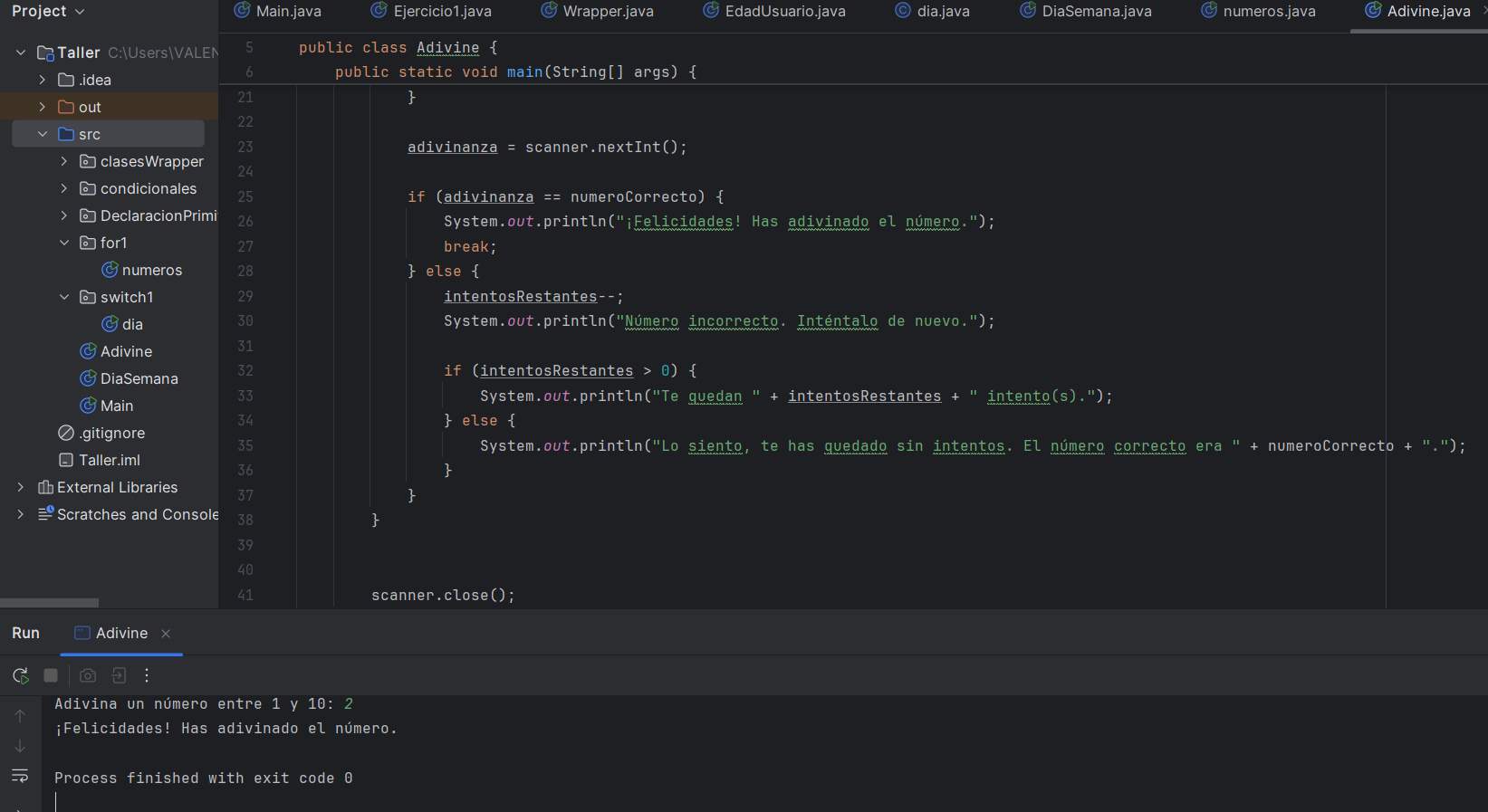


#### **Reflexión:**

¿Qué sucede si el número nunca es adivinado? ¿Cómo podrías mejorar el programa para evitar que se repita indefinidamente?

se podría colocar un límite.





#### **3.3. El bucle do-while**

Este bucle es similar al while, pero se asegura de que el código se ejecute al menos una vez antes de verificar la condición.

#### **Concepto clave:**

* do-while: Primero ejecuta el código y luego verifica la condición.

#### **Ejemplo:**

java

Copiar código

int num = 10;

do {

System.out.println("Número: " + num);

num--;

} while (num > 0);

#### **Pregunta:**

¿Cuál es la principal diferencia entre while y do-while?

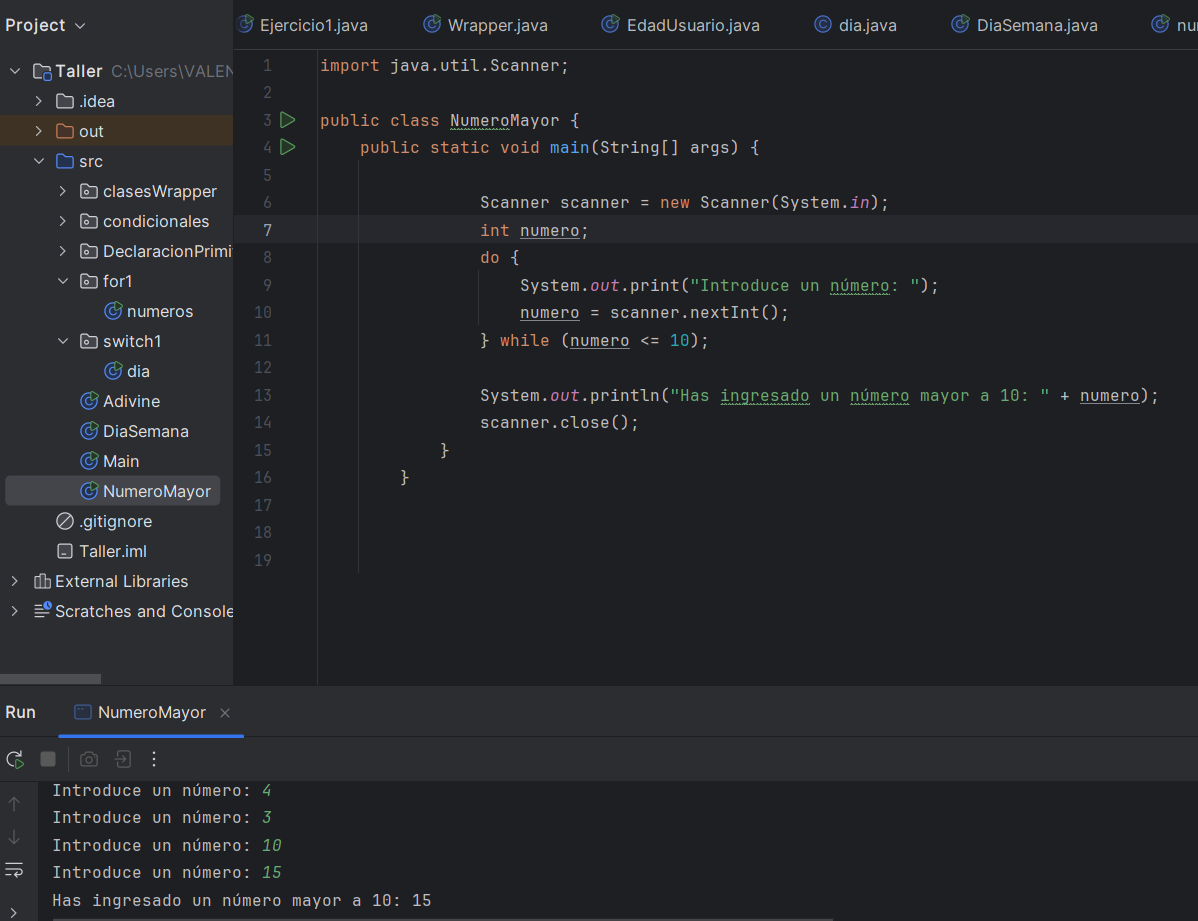
Bucle while Evaluación previa a la ejecución: La condición se evalúa antes de entrar en el bucle. Si la condición es verdadera se ejecuta, do while se asegura antes que el código se ejecute al menos una vez antes de la verificación del código.

### **Tarea interactiva 5:**

Escribe un programa que siga pidiendo un número al usuario hasta que ingrese un número mayor a 10. Utiliza un do-while.

#### **Pista:**

Coloca la lectura del número dentro del bloque do para que el código se ejecute al menos una vez.



### **Conclusión**

¡Excelente trabajo! Ahora comprendes las estructuras de control en Java y cómo usarlas para tomar decisiones y repetir acciones.

#### **Ejercicio Final:**

Crea un programa que simule un cajero automático. El usuario debe poder:

1. Ver el saldo.
2. Depositar dinero.
3. Retirar dinero.
4. Salir.

Utiliza un bucle para mostrar el menú repetidamente hasta que el usuario elija salir.

### **Reflexión Final**

¿Qué fue lo más fácil de aprender?

la iteración con for.

¿Qué parte te resultó más difícil?

el while colocando límite

¿Cómo podrías mejorar en el uso de estructuras de control en futuros programas?

con

¡Sigue practicando y mejorando! Las estructuras de control son esenciales en la programación, y cuanto más las practiques, más fácil te resultará crear programas complejos.

### **Tutorial Interactivo: Clases, Métodos, Atributos e Instancias en Java**

Bienvenido al tutorial de **Programación Orientada a Objetos en Java**. En este recorrido aprenderás los conceptos clave de clases, métodos, atributos e instancias a través de ejemplos y ejercicios interactivos que te ayudarán a aplicar los conocimientos adquiridos. ¡Empecemos!

## **Parte 1: ¿Qué es una Clase?**

### **Concepto:**

Una **clase** es una plantilla o blueprint que define cómo será un objeto. En una clase se especifican los **atributos** (características) y **métodos** (acciones o comportamientos) que los objetos de esa clase tendrán.

Piensa en una clase como una receta para crear objetos. Por ejemplo, una clase Persona puede tener atributos como nombre, edad, y altura, y métodos como caminar() o hablar().

### **Ejemplo de Código:**

java

Copiar código

public class Persona {

// Atributos

String nombre;

int edad;

// Método

void hablar() {

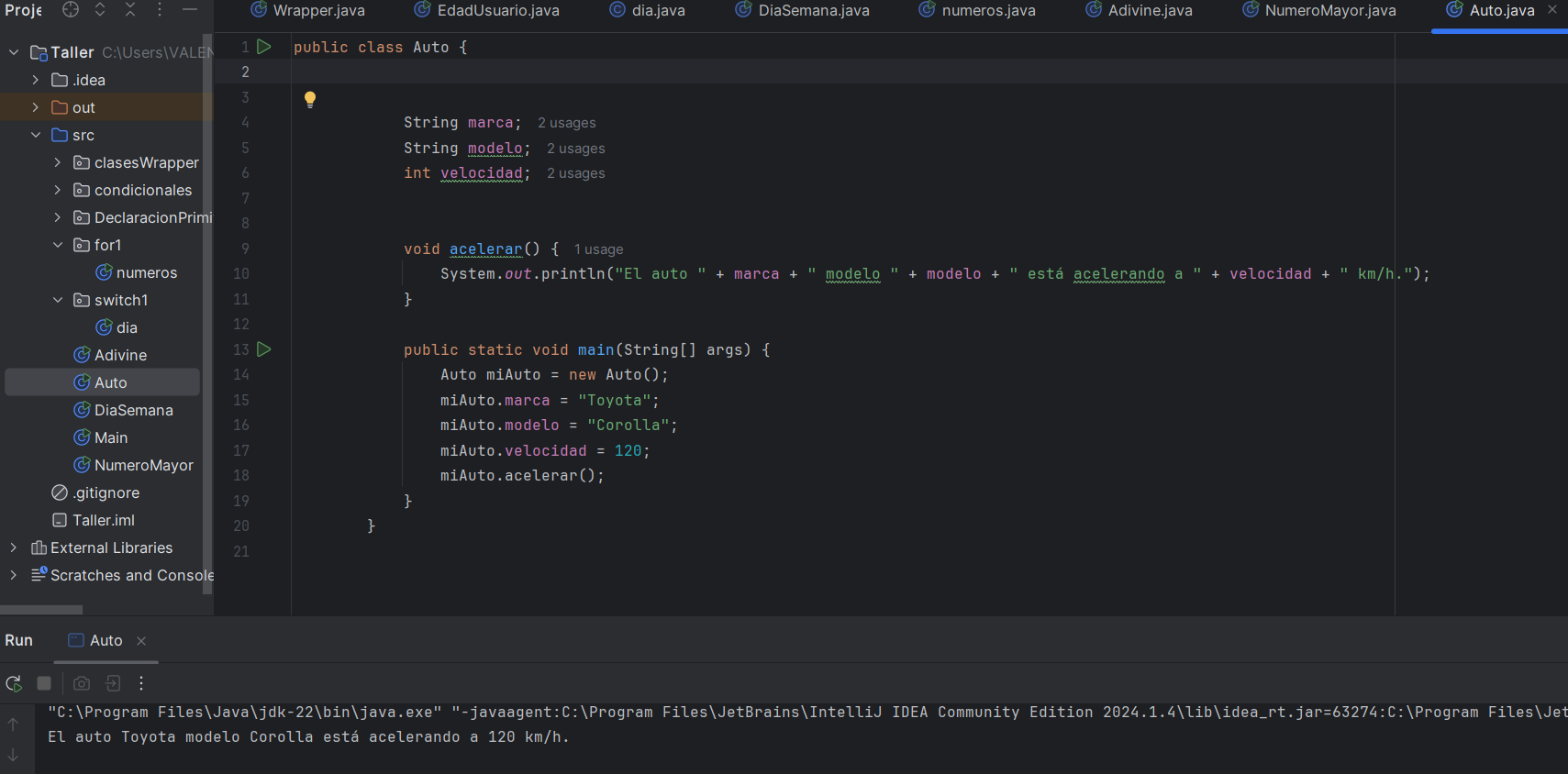
System.out.println(nombre + " está hablando.");

}

}

### **Tarea 1: Crea tu propia clase**

* Define una clase llamada Auto que tenga los atributos marca, modelo y velocidad.
* Añade un método llamado acelerar() que imprima un mensaje indicando que el auto está acelerando.



## **Parte 2: ¿Qué son los Atributos?**

### **Concepto:**

Los **atributos** son variables que representan las propiedades de los objetos de una clase. Cada objeto puede tener diferentes valores para estos atributos.

Por ejemplo, en una clase Libro, los atributos podrían ser título, autor y numeroPaginas.

### **Ejemplo de Código:**

java

Copiar código

public class Libro {

// Atributos

String titulo;

String autor;

int numeroPaginas;

}

### **Tarea 2: Personaliza tus atributos**

* Modifica la clase Auto que creaste anteriormente y define los atributos marca, modelo y velocidad.
* Luego, crea un objeto de la clase Auto y asigna valores a esos atributos.

java

Copiar código

Auto miAuto = new Auto();

miAuto.marca = "Toyota";

miAuto.modelo = "Corolla";

miAuto.velocidad = 120;

## **Parte 3: ¿Qué es un Método?**

### **Concepto:**

Los **métodos** son funciones definidas dentro de una clase que permiten a los objetos realizar acciones. Un método puede tomar parámetros y devolver un valor o simplemente ejecutar una tarea.

### **Ejemplo de Código:**

java

public class Calculadora {

// Método para sumar dos números

int sumar(int a, int b) {

return a + b;

}

}

### **Tarea 3: Define tus propios métodos**

* Añade un método frenar() en la clase Auto que reduzca la velocidad del auto.
* Luego, crea una instancia de Auto y llama al método frenar().

java

Copiar código

public class Auto {

String marca;

String modelo;

int velocidad;

void acelerar() {

velocidad += 10;

System.out.println("El auto está acelerando. Velocidad actual: " + velocidad);

}

void frenar() {

velocidad -= 10;

System.out.println("El auto está frenando. Velocidad actual: " + velocidad);

}

}

## **Parte 4: ¿Qué es una Instancia?**

### **Concepto:**

Una **instancia** es un objeto creado a partir de una clase. Cada vez que creas un nuevo objeto, estás instanciando una clase.

Por ejemplo, a partir de la clase Persona, puedes crear múltiples instancias (objetos) que representen personas específicas, como Juan o Maria.

### **Ejemplo de Código:**

java

Copiar código

public class Main {

public static void main(String[] args) {

// Creando una instancia de la clase Persona

Persona persona1 = new Persona();

persona1.nombre = "Juan";

persona1.edad = 25;

// Llamando al método

persona1.hablar();

}

}

### **Tarea 4: Crea instancias**

* Crea dos instancias de la clase Auto, con diferentes valores para los atributos marca, modelo y velocidad.
* Llama a los métodos acelerar() y frenar() para cada instancia.

java

Copiar código

Auto auto1 = new Auto();

auto1.marca = "Honda";

auto1.modelo = "Civic";

auto1.velocidad = 100;

Auto auto2 = new Auto();

auto2.marca = "Ford";

auto2.modelo = "Fiesta";

auto2.velocidad = 90;

auto1.acelerar();

auto2.frenar();

## **Parte 5: Interacción Estudiante-Tutor**

### **Pregunta 1:**

¿Qué crees que pasaría si no inicializas los atributos de un objeto antes de llamar a un método que los utilice?

1. A) Se produce un error.
2. B) El programa imprime valores por defecto.
3. C) No ocurre nada.

Respuesta B

### **Pregunta 2:**

¿Cuál es la diferencia entre un método con valor de retorno y uno sin valor de retorno (void)?

Un **método con valor de retorno** devuelve un valor cuando se completa su ejecución. Ese valor puede ser de cualquier tipo de dato.

## **Parte 6: Ejercicio Final**

### **Descripción:**

Crea una clase llamada Estudiante que tenga los atributos nombre, edad, grado y el método estudiar(). Luego, crea 3 instancias de la clase Estudiante y llama al método estudiar() para cada uno. Asegúrate de asignarles valores a los atributos.

### **Pistas:**

* Usa la sintaxis Estudiante estudiante1 = new Estudiante(); para crear instancias.
* Usa estudiante1.nombre = "Ana"; para asignar valores a los atributos.

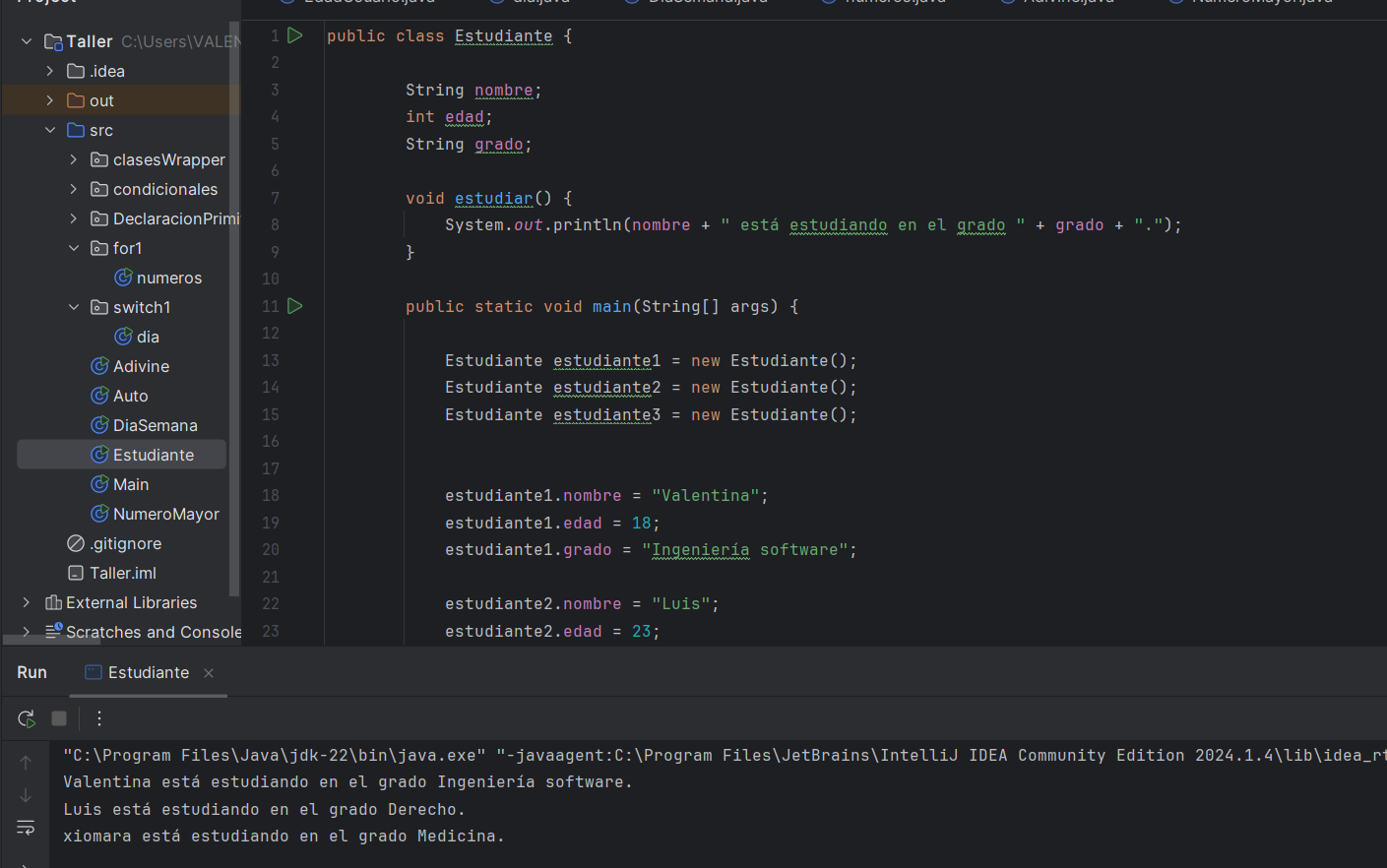
### **Ejemplo de salida esperada:**

java

Ana está estudiando.

Carlos está estudiando.

Luis está estudiando.





## **Conclusión:**

En este tutorial interactivo, has aprendido los conceptos fundamentales de la Programación Orientada a Objetos en Java: **clases, atributos, métodos e instancias**. Recuerda que la práctica constante te ayudará a entender mejor estos conceptos. ¡Sigue practicando y experimentando con tus propias clases y objetos!

### **Tutorial Interactivo: Modificadores de Acceso en Java (public, protected, private)**

¡Bienvenido al tutorial sobre los **modificadores de acceso** en Java! En este recorrido interactivo, aprenderás cómo controlar la visibilidad de las clases, métodos y atributos de tu código utilizando public, protected y private. Siguiendo los pasos propuestos, tendrás la oportunidad de aplicar los conceptos mediante ejercicios prácticos.

### **Introducción**

En Java, los **modificadores de acceso** nos permiten definir qué tan accesibles son los elementos (atributos, métodos, clases) dentro y fuera de una clase. Esto es esencial para el encapsulamiento, uno de los principios clave de la **programación orientada a objetos**.

Vamos a profundizar en los tres modificadores más importantes:

* **public**: Acceso desde cualquier parte del programa.
* **protected**: Acceso dentro del mismo paquete o clases derivadas.
* **private**: Acceso solo dentro de la misma clase.

#### **Pregunta para el estudiante:**

**¿Por qué crees que es importante controlar el acceso a los atributos y métodos de una clase en Java?**

Protege los datos, Mejora la seguridad y Hace el código más fácil de mantener

### **1. Modificador public**

El modificador public permite que un miembro de la clase (atributo o método) sea accesible desde cualquier lugar del programa, incluso fuera del paquete donde se definió la clase.

#### **Ejemplo:**

java

Copiar código

class Persona {

public String nombre;

public void saludar() {

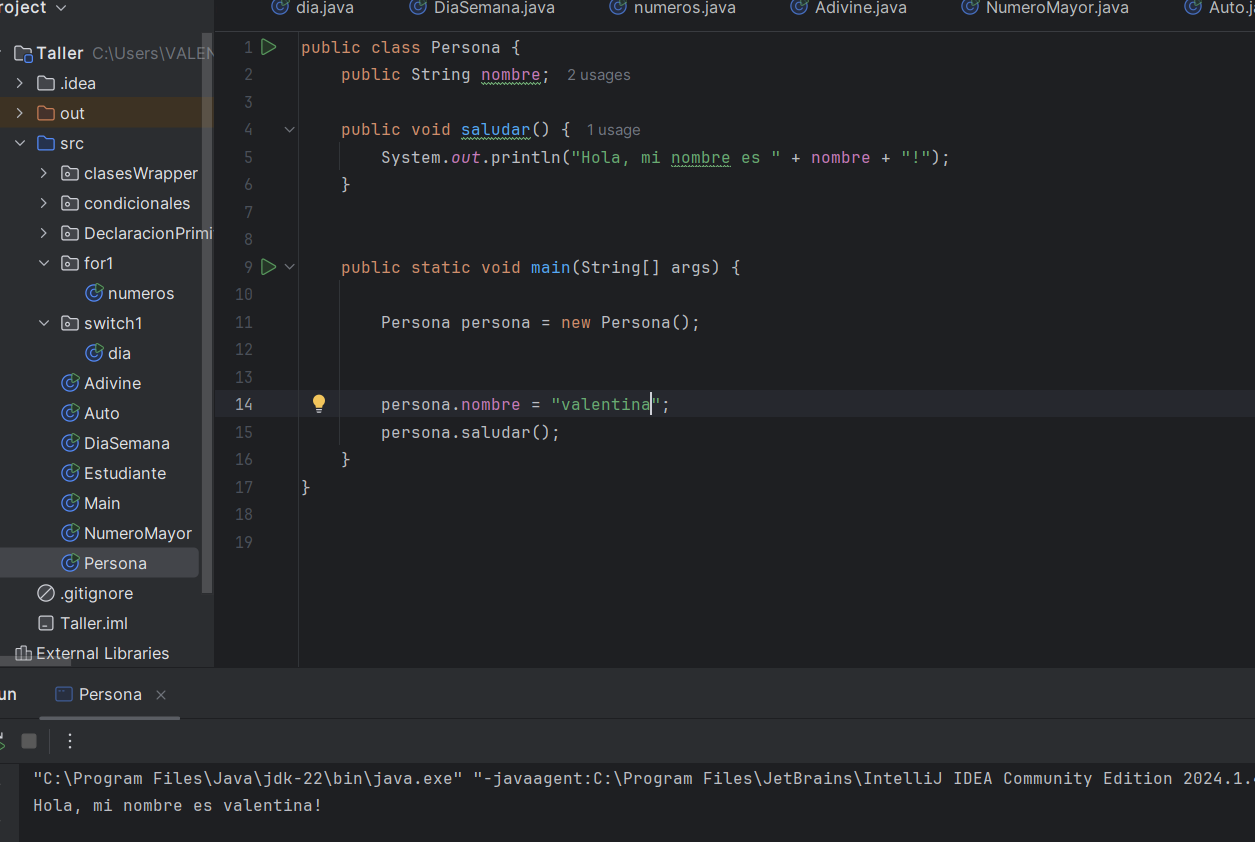
System.out.println("¡Hola, mi nombre es " + nombre + "!");

}

}

#### **Tarea:**

1. Crea una clase llamada Persona.
2. Define un atributo public llamado nombre.
3. Implementa un método public llamado saludar que imprima un mensaje con el nombre.



#### **Pregunta:**

**Si cambias public por private en el atributo nombre, ¿qué crees que sucederá cuando intentes acceder al atributo desde fuera de la clase?**

el atributo dejará de ser accesible directamente desde fuera de la clase, obtendrás un error de compilación**.**

#### **Ejercicio Práctico:**

1. Escribe un programa en el que instancies un objeto Persona, asigna un valor a nombre y llama al método saludar.
2. Cambia el modificador de nombre de public a private. ¿Qué error obtienes?

**error: nombre has private access in Persona persona.nombre = "Juan";**

### **2. Modificador protected**

El modificador protected permite que un miembro de la clase sea accesible dentro del mismo **paquete** y por las clases **heredadas**.

#### **Ejemplo:**

java

Copiar código

class Animal {

protected String tipo;

protected void hacerSonido() {

System.out.println("El animal hace un sonido.");

}

}

class Perro extends Animal {

public void hacerSonido() {

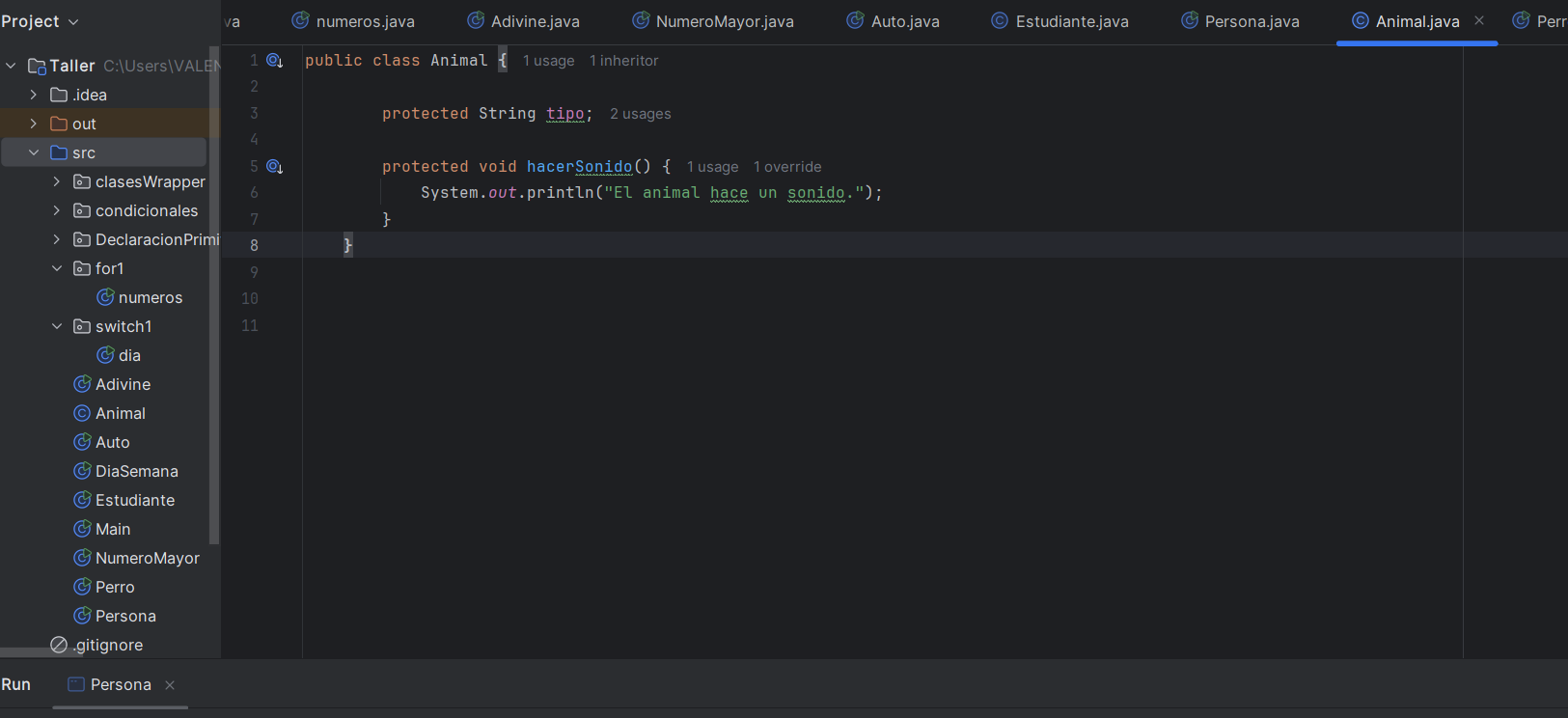
System.out.println("El perro ladra.");

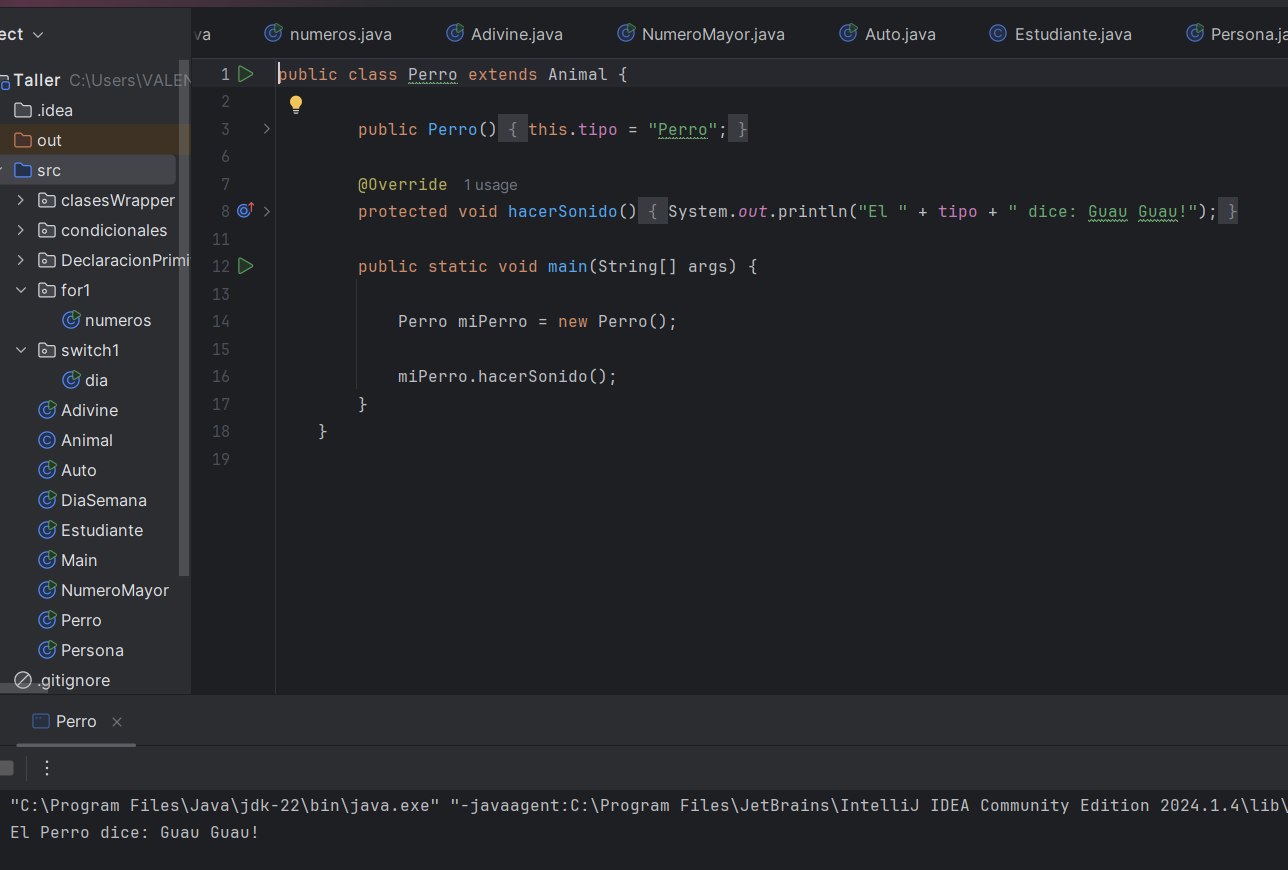
}

}

#### **Tarea:**

1. Crea una clase Animal con un atributo protected llamado tipo.
2. Define un método protected llamado hacerSonido.
3. Crea una clase Perro que herede de Animal y sobreescribe el método hacerSonido.





#### **Pregunta:**

**Si instancias un objeto de la clase Perro, ¿podrás acceder al atributo tipo directamente desde fuera de la clase? ¿Por qué?**

Esto se debe a que el modificador de acceso protected permite que el atributo sea accesible únicamente dentro de la propia clase.

Dentro de las clases que heredan de esta clase (subclases).

Dentro del mismo paquete.

generaría el siguiente erro**r**

**error: tipo has protected access in Animal**

**System.out.println(miPerro.tipo);**

#### **Ejercicio Práctico:**

1. Escribe un programa donde crees una clase que herede de Animal y sobrescriba el método hacerSonido.
2. Intenta acceder al atributo tipo desde fuera de la clase Perro y observa qué sucede.

public class Animal {

protected String tipo;

protected void hacerSonido() {

System.out.println("El animal hace un sonido.");

}

public String getTipo() {

return tipo;

}

}

public class Perro extends Animal {

public Perro() {

this.tipo = "Perro";

}

@Override

protected void hacerSonido() {

System.out.println("El " + tipo + " dice: Guau Guau!");

}

}

public class Gato extends Animal {

public Gato() {

this.tipo = "Gato";

}

@Override

protected void hacerSonido() {

System.out.println("El " + tipo + " dice: Miau Miau!");

}

}

public class Main {

public static void main(String[] args) {

Perro miPerro = new Perro();

miPerro.hacerSonido();

Gato miGato = new Gato();

miGato.hacerSonido();

System.out.println("El tipo de animal es: " + miPerro.getTipo());

System.out.println("El tipo de animal es: " + miGato.getTipo());

}

}

### **3. Modificador private**

El modificador private restringe el acceso a un miembro de la clase únicamente dentro de esa misma clase. Esto significa que ningún código fuera de la clase puede acceder a un atributo o método marcado como private.

#### **Ejemplo:**

java

Copiar código

class CuentaBancaria {

private double saldo;

public CuentaBancaria(double saldoInicial) {

saldo = saldoInicial;

}

private void mostrarSaldo() {

System.out.println("El saldo es: " + saldo);

}

public void depositar(double monto) {

saldo += monto;

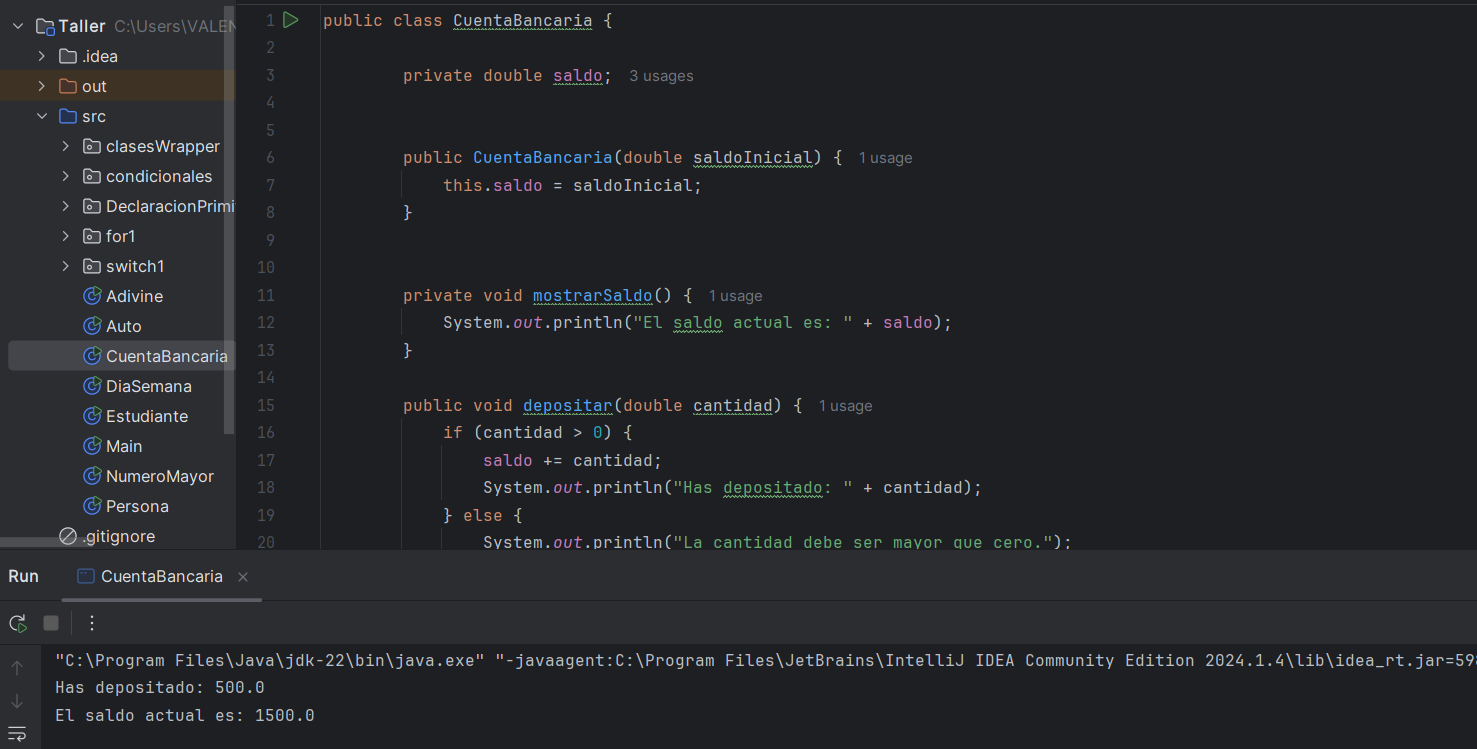
mostrarSaldo();

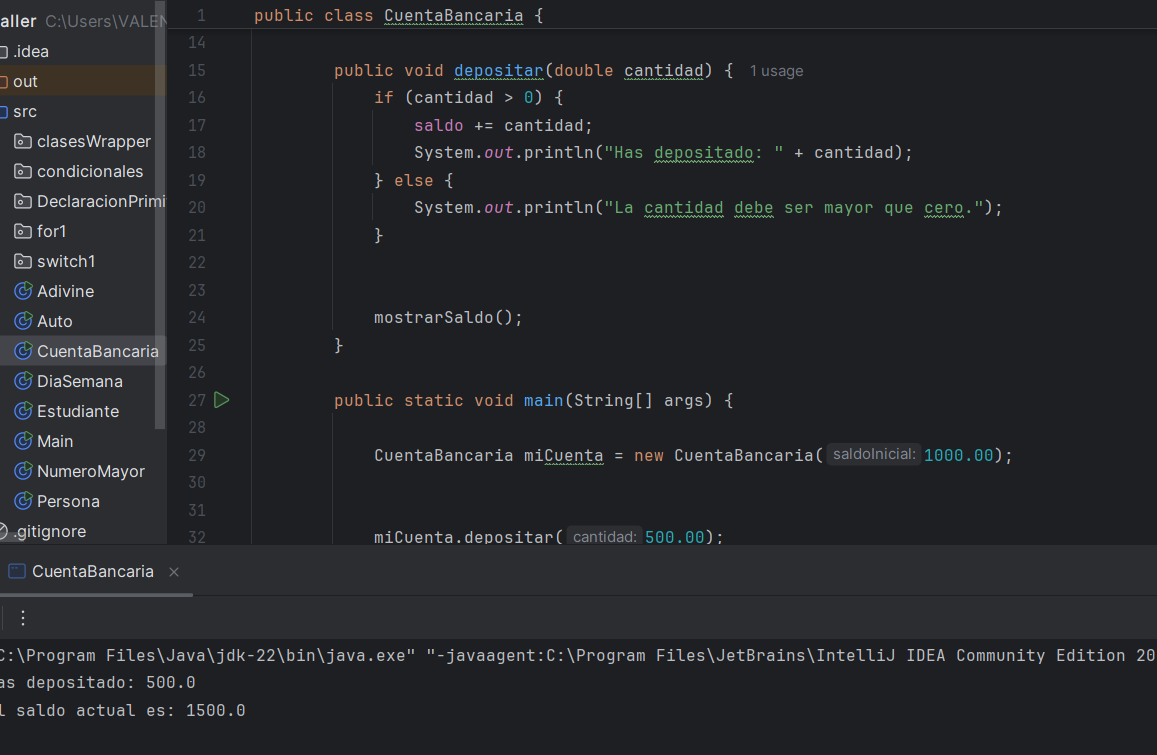
}

}

#### **Tarea:**

1. Crea una clase CuentaBancaria con un atributo private llamado saldo.
2. Define un método private llamado mostrarSaldo.
3. Crea un método public llamado depositar que permita agregar dinero al saldo y que, dentro de este método, llame al método private.





#### **Pregunta:**

**¿Por qué usarías un método private en lugar de public en este caso?**

en este caso tiene varias ventajas, principalmente relacionadas con el encapsulamiento y el control sobre cómo se accede y manipula la información dentro de la clase

#### **Ejercicio Práctico:**

1. Escribe un programa donde crees una instancia de CuentaBancaria, realices un depósito y observes cómo el método private es utilizado indirectamente.
2. Intenta llamar al método mostrarSaldo desde fuera de la clase. ¿Qué sucede?

public class CuentaBancaria

public CuentaBancaria(double saldoInicial) {

this.saldo = saldoInicial;

}

private void mostrarSaldo() {

System.out.println("El saldo actual es: " + saldo);

}

public void depositar(double cantidad) {

if (cantidad > 0) {

saldo += cantidad;

System.out.println("Has depositado: " + cantidad);

} else {

System.out.println("La cantidad debe ser mayor que cero.");

}

mostrarSaldo();

}

}

public class Main {

public static void main(String[] args) {

CuentaBancaria miCuenta = new CuentaBancaria(1000.00);

miCuenta.depositar(500.00); // El método mostrarSaldo se llamará indirectamente

// Intentar llamar al método mostrarSaldo directamente (causará error)

// miCuenta.mostrarSaldo(); // <- Esto causará un error de compilación

}

}

### **4. Resumen Comparativo**

| **Modificador** | **Acceso dentro de la clase** | **Acceso desde el mismo paquete** | **Acceso por herencia** | **Acceso desde otras clases** |
| --- | --- | --- | --- | --- |
| public | ✔ | ✔ | ✔ | ✔ |
| protected | ✔ | ✔ | ✔ | ✘ |
| private | ✔ | ✘ | ✘ | ✘ |

#### **Pregunta de Reflexión:**

**¿Cómo escogerías el modificador adecuado en función del diseño de tu clase?**

private.

### **5. Desafío Final: Diseña una Clase**

**Ejercicio Completo:** Diseña una clase llamada Vehiculo con los siguientes requerimientos:

1. Un atributo private llamado velocidadMaxima.
2. Un atributo protected llamado tipoCombustible.
3. Un método public llamado encender que imprima "El vehículo está encendido".
4. Un método private llamado mostrarVelocidadMaxima.
5. Crea una clase Auto que herede de Vehiculo y sobrescriba el método encender para imprimir "El auto está encendido".

**Tareas:**

1. Instancia un objeto de Auto y prueba llamar a los métodos disponibles.
2. Intenta acceder al atributo velocidadMaxima desde fuera de la clase. ¿Qué sucede?
3. Crea un método dentro de Auto que use el método private de Vehiculo a través de un método public.

public class Vehiculo {

private int velocidadMaxima;

// Atributo protegido

protected String tipoCombustible;

public Vehiculo(int velocidadMaxima, String tipoCombustible) {

this.velocidadMaxima = velocidadMaxima;

this.tipoCombustible = tipoCombustible;

}

public void encender() {

System.out.println("El vehículo está encendido.");

}

private void mostrarVelocidadMaxima() {

System.out.println("La velocidad máxima es: " + velocidadMaxima + " km/h");

}

protected void accederMostrarVelocidadMaxima() {

mostrarVelocidadMaxima(); // Llamada al método private

}

}

public class Auto extends Vehiculo {

public Auto(int velocidadMaxima, String tipoCombustible) {

super(velocidadMaxima, tipoCombustible); // Llamar al constructor de la clase base

}

@Override

public void encender() {

System.out.println("El auto está encendido.");

}

public void mostrarVelocidadDelAuto() {

accederMostrarVelocidadMaxima();

}

}

public class Main {

public static void main(String[] args) {

Auto miAuto = new Auto(180, "Gasolina");

miAuto.encender(); // Salida: "El auto está encendido."

miAuto.mostrarVelocidadDelAuto(); // Llama al método private a través de un método protegido

}

}

### **Conclusión**

Has completado el tutorial sobre **modificadores de acceso** en Java. Recuerda que elegir el modificador correcto es clave para asegurar la integridad y seguridad de tu código. Controlar el acceso a los atributos y métodos permite un diseño robusto y bien encapsulado.

### **Resumen:**

1. **public**: Acceso libre desde cualquier parte.
2. **protected**: Acceso dentro del paquete y por herencia.
3. **private**: Acceso solo dentro de la clase.

¡Felicidades por terminar el tutorial! ¿Estás listo para aplicar estos conceptos en tus propios proyectos?



## **Tutorial Interactivo: Uso de final en Clases, Métodos y Atributos en Java**

### **¡Bienvenido al tutorial interactivo sobre final en Java!**

En este tutorial, aprenderás sobre cómo y cuándo usar la palabra clave final en **clases**, **métodos** y **atributos**. Todo está diseñado para que participes activamente, resolviendo preguntas y haciendo ejercicios prácticos.

### **1. ¿Qué es final en Java?**

La palabra clave final en Java se utiliza para restringir la modificación de clases, métodos o atributos. Dependiendo de cómo la apliques, su significado cambia ligeramente:

* **Clases final**: Una clase final no puede ser extendida (no puede tener subclases).
* **Métodos final**: Un método final no puede ser sobrescrito por las subclases.
* **Atributos final**: Un atributo final se comporta como una constante, su valor no puede cambiar después de ser asignado.

### **2. final en Clases**

### **Concepto:**

Si declaras una clase como final, impides que cualquier otra clase la herede. Esto es útil cuando deseas asegurarte de que el comportamiento de esa clase no sea alterado por clases derivadas.

#### **Ejemplo:**

java

Copiar código

public final class Vehiculo {

public void encender() {

System.out.println("El vehículo está encendido.");

}

}

Aquí, Vehiculo es una clase final, por lo tanto, ninguna clase puede heredar de ella.

#### **🧠 Pregunta 1:**

¿Por qué sería útil hacer una clase final? ¿En qué situaciones querrías evitar que una clase sea heredada?

Hacer una clase final en Java es útil en varias situaciones específicas, principalmente relacionadas con la **seguridad**, la **integridad del diseño** y el **rendimiento.**

#### **📝 Tarea 1:**

Crea una clase llamada Banco y hazla final. Intenta heredar de esta clase creando otra clase llamada BancoOnline. ¿Qué sucede?

java

Copiar código

// Tu código aquí

public final class Banco {

// Métodos y atributos del banco

}

// Intenta heredar de Banco

public class BancoOnline extends Banco {

// Métodos de BancoOnline

}

Cuando intentes compilar este código, ¿qué error aparece?

### **3. final en Métodos**

### **Concepto:**

Un método declarado como final no puede ser sobrescrito en una subclase. Esto es útil si deseas que el comportamiento de un método sea consistente en toda la jerarquía de herencia.

#### **Ejemplo:**

java

Copiar código

public class Animal {

public final void hacerSonido() {

System.out.println("El animal hace un sonido.");

}

}

public class Perro extends Animal {

// Esto causará un error si intentas sobrescribir hacerSonido

/\*

@Override

public void hacerSonido() {

System.out.println("El perro ladra.");

}

\*/

}

El método hacerSonido de la clase Animal es final, por lo que la clase Perro no puede sobrescribirlo.

#### **🧠 Pregunta 2:**

¿Cuáles podrían ser las razones para declarar un método como final? ¿Por qué podría ser peligroso permitir que un método importante sea sobrescrito?

Declarar un método como final en Java significa que dicho método **no puede ser sobrescrito** por subclases. Esto puede ser útil en varias situaciones, principalmente relacionadas con la **integridad del código**, **seguridad** y **rendimiento**

#### **📝 Tarea 2:**

Crea una clase Computadora con un método final llamado iniciar. Luego, crea una clase Laptop que intente sobrescribir el método iniciar. ¿Qué sucede?

java

Copiar código

public class Computadora {

public final void iniciar() {

System.out.println("La computadora está iniciando.");

}

}

public class Laptop extends Computadora {

// Intenta sobrescribir el método iniciar aquí

}

¿Qué error obtienes al intentar sobrescribir un método final? ¿Cómo solucionarías este problema?

Cuando intentas sobrescribir un método final en Java, obtienes un **error de compilación**. Esto sucede porque la palabra clave final en un método indica que no puede ser modificado por las subclases, y cualquier intento de sobrescribirlo va en contra de esa restricción.

### **4. final en Atributos**

### **Concepto:**

Cuando declaras un atributo como final, estás diciendo que su valor no podrá cambiar una vez que se le haya asignado. Este tipo de variables se comportan como **constantes**.

#### **Ejemplo:**

java

Copiar código

public class Vehiculo {

public final int VELOCIDAD\_MAXIMA = 120;

public void mostrarVelocidad() {

System.out.println("La velocidad máxima es: " + VELOCIDAD\_MAXIMA);

}

}

En este ejemplo, VELOCIDAD\_MAXIMA es una constante. Si intentas cambiar su valor más tarde en el programa, el compilador lanzará un error.

#### **🧠 Pregunta 3:**

¿Cuáles serían algunos ejemplos de atributos que siempre deben ser constantes y no deberían cambiar una vez asignados? ¿Por qué es importante tener este tipo de variables en un programa?

, los atributos que deberían ser **constantes** y no cambiar después de ser asignados se declaran con la palabra clave final. Estos atributos suelen representar valores que son **inmutables** y que tienen un significado crítico en el funcionamiento del programa.

#### **📝 Tarea 3:**

Crea una clase llamada Persona con un atributo final llamado DNI. Inicializa el atributo DNI en el constructor. Intenta cambiar el valor de DNI después de asignarlo. ¿Qué error obtienes?

java

Copiar código

public class Persona {

public final String DNI;

public Persona(String dni) {

this.DNI = dni; // Inicialización del atributo final

}

public void cambiarDNI(String nuevoDNI) {

// Intenta cambiar el valor de DNI aquí

}

}

¿Cuál es la razón por la que el compilador no permite modificar el valor de un atributo final?

La razón por la que el compilador no permite modificar el valor de un atributo declarado como final es para garantizar la **inmutabilidad** de ese atributo. Cuando se marca una variable con la palabra clave final, el compilador entiende que esa variable no puede ser reasignada una vez que ha recibido un valor inicial. Esto implica que el valor de la variable es **constante** durante el tiempo de vida del objeto o la ejecución del programa.

### **5. Resumen:**

#### **- Clases final: No pueden ser heredadas.**

#### **- Métodos final: No pueden ser sobrescritos.**

#### **- Atributos final: No pueden cambiar su valor después de ser asignados.**

### **6. Ejercicio Final: ¡Ponlo en práctica!**

#### **📝 Ejercicio:**

Crea una clase Empleado con lo siguiente:

1. Un atributo final llamado ID\_EMPLEADO.
2. Un atributo protected llamado salario.
3. Un método final llamado calcularSalario que imprima el salario.
4. Crea una clase Gerente que herede de Empleado e intente sobrescribir el método calcularSalario. Observa qué sucede.

java

Copiar código

public class Empleado {

public final String ID\_EMPLEADO;

protected double salario;

public Empleado(String id, double salario) {

this.ID\_EMPLEADO = id;

this.salario = salario;

}

public final void calcularSalario() {

System.out.println("El salario es: " + salario);

}

}

// Clase Gerente que intenta sobrescribir calcularSalario

public class Gerente extends Empleado {

// Intenta sobrescribir calcularSalario aquí

}

Después de realizar este ejercicio, reflexiona:

* ¿Por qué sería útil hacer que ciertos métodos no puedan ser sobreescritos?

Hacer que ciertos métodos no puedan ser sobrescritos utilizando el modificador final es útil por varias razones que tienen que ver con la seguridad del código, la integridad del diseño y la eficiencia

* ¿Qué tan importantes son las constantes (final en atributos) en la programación?

Las **constantes** (atributos marcados con final) son muy importantes en la programación, ya que proporcionan varios beneficios clave relacionados con la seguridad, legibilidad y rendimiento del código

### **7. Conclusión:**

En este tutorial has aprendido a usar final en clases, métodos y atributos. Este concepto es clave para garantizar que ciertas partes del código no sean modificadas o extendidas, protegiendo así la integridad del sistema y facilitando el mantenimiento.

### **🚀 Tu próximo paso:**

Prueba a aplicar lo aprendido en tu propio proyecto. Usa final en lugares estratégicos donde creas que el comportamiento de clases, métodos o atributos no debe cambiar. ¡La práctica te ayudará a dominar este concepto fundamental de la Programación Orientada a Objetos en Java!

### **Tutorial Interactivo: Constructores y Sobrecarga en Java**

**Basado en "Programación Orientada a Objetos en Java"**

¡Bienvenido al tutorial sobre **Constructores y Sobrecarga en Java**! Aquí aprenderás estos dos conceptos clave de la programación orientada a objetos, **paso a paso** y con ejemplos prácticos. Cada sección está diseñada para que interactúes y practiques activamente. ¡Comencemos!

### **Objetivos del tutorial:**

* Entender qué es un **constructor** y cómo funciona en Java.
* Aprender sobre la **sobrecarga de métodos** y la **sobrecarga de constructores**.
* Aplicar los conocimientos a través de ejemplos y ejercicios.

## **1. ¿Qué es un Constructor en Java?**

### **Concepto clave:**

Un **constructor** en Java es un **método especial** que se utiliza para inicializar objetos. Se llama automáticamente cuando creas una nueva instancia de una clase. Tiene el mismo nombre que la clase y **no tiene ningún tipo de retorno**, ni siquiera void.

### **Características de un constructor:**

* Se invoca cuando se crea un objeto con new.
* No tiene un valor de retorno (ni siquiera void).
* El nombre del constructor debe coincidir con el nombre de la clase.

### **Ejemplo básico:**

java

Copiar código

class Persona {

String nombre;

// Constructor de la clase Persona

public Persona(String nombre) {

this.nombre = nombre;

}

}

En este ejemplo, cuando creas un nuevo objeto Persona, el constructor se ejecuta automáticamente para asignar el nombre:

java

Copiar código

Persona persona1 = new Persona("Carlos");

### **📝 Tarea 1: Experimenta con un Constructor**

1. Crea una clase llamada Coche que tenga un atributo marca.
2. Define un constructor que tome un parámetro marca y lo asigne al atributo.
3. Crea una instancia de Coche en tu método main e imprime la marca.

class Coche {

String marca;

public Coche(String marca) {

this.marca = marca;

}

}

public class Main {

public static void main(String[] args) {

Coche miCoche = new Coche("Toyota");

System.out.println("La marca de mi coche es: " + miCoche.marca);

}

}

java

Copiar código

class Coche {

// Completa el código...

}

¡Intenta hacerlo por ti mismo antes de ver la solución!

### **Pregunta:**

¿Qué sucede si no defines un constructor en una clase? ¿Java proporciona uno de manera automática? **(Sí/No)**

Sí, Java proporciona un constructor por defecto de manera automática si no defines uno en tu clase**.**

#### **📘 Respuesta:**

Si no defines un constructor en una clase, Java proporciona automáticamente un **constructor por defecto**. Este constructor no toma parámetros y no realiza ninguna inicialización específica, excepto la asignación de valores predeterminados a los atributos (por ejemplo, null para objetos y 0 para tipos primitivos).

## **2. Sobrecarga de Constructores**

### **Concepto clave:**

La **sobrecarga** de constructores te permite definir **múltiples constructores** dentro de la misma clase, siempre y cuando cada uno tenga una **lista de parámetros diferente**. Esto te permite crear objetos de diversas maneras según los argumentos que pases al constructor.

### **Ejemplo:**

java

Copiar código

class Coche {

String marca;

int año;

// Primer constructor

public Coche(String marca) {

this.marca = marca;

}

// Segundo constructor sobrecargado

public Coche(String marca, int año) {

this.marca = marca;

this.año = año;

}

}

En este caso, puedes crear un coche solo con la marca:

java

Copiar código

Coche coche1 = new Coche("Toyota");

O con la marca y el año:

java

Copiar código

Coche coche2 = new Coche("Honda", 2020);

### **📝 Tarea 2: Crea un Constructor Sobrecargado**

1. Define una clase Libro con atributos titulo y autor.
2. Crea un constructor que solo reciba el titulo.
3. Crea otro constructor que reciba tanto el titulo como el autor.
4. En el método main, crea dos objetos de la clase Libro, uno con solo el título y otro con el título y el autor.

**java**

**Copiar código**

**class Libro {**

**}**

**class Libro {**

**String titulo;**

**String autor;**

**public Libro(String titulo) {**

**this.titulo = titulo;**

**this.autor = "Autor desconocido"; // Valor por defecto si no se proporciona el autor**

**}**

**public Libro(String titulo, String autor) {**

**this.titulo = titulo;**

**this.autor = autor;**

**}**

**}**

**public class Main {**

**public static void main(String[] args) {**

**Libro libro1 = new Libro("El Quijote");**

**Libro libro2 = new Libro("Cien años de soledad", "Gabriel García Márquez");**

**System.out.println("Libro 1: Título: " + libro1.titulo + ", Autor: " + libro1.autor);**

**System.out.println("Libro 2: Título: " + libro2.titulo + ", Autor: " + libro2.autor);**

**}**

**}**

### **Pregunta:**

¿Por qué es útil la sobrecarga de constructores en Java? ¿En qué situaciones puede ser ventajoso tener más de un constructor?.

#### **📘 Respuesta:**

La sobrecarga de constructores es útil cuando quieres ofrecer varias formas de inicializar un objeto. Puede ser útil en situaciones donde no siempre tienes toda la información necesaria al momento de crear el objeto. Por ejemplo, podrías crear un objeto con valores predeterminados o valores mínimos, y otro con más detalles según sea necesario.

## **3. Sobrecarga de Métodos**

### **Concepto clave:**

La **sobrecarga de métodos** es un principio similar a la sobrecarga de constructores, pero aplicado a **cualquier método** en una clase. Puedes definir varios métodos con el mismo nombre, pero con **diferentes tipos o números de parámetros**.

### **Ejemplo:**

java

Copiar código

class Calculadora {

// Sumar dos enteros

public int sumar(int a, int b) {

return a + b;

}

// Sumar tres enteros

public int sumar(int a, int b, int c) {

return a + b + c;

}

// Sumar dos números decimales

public double sumar(double a, double b) {

return a + b;

}

}

Ahora puedes usar el método sumar con diferentes tipos y cantidades de argumentos:

java

Copiar código

Calculadora calc = new Calculadora();

System.out.println(calc.sumar(10, 20)); // Resultado: 30

System.out.println(calc.sumar(10, 20, 30)); // Resultado: 60

System.out.println(calc.sumar(10.5, 20.3)); // Resultado: 30.8

### **📝 Tarea 3: Implementa la Sobrecarga de Métodos**

1. Crea una clase llamada Operaciones que tenga tres versiones sobrecargadas de un método multiplicar:
   * Una versión que multiplique dos enteros.
   * Una versión que multiplique tres enteros.
   * Una versión que multiplique dos números decimales.

java

Copiar código

class Operaciones {

// Completa el código...

}

Prueba el método multiplicar con diferentes tipos y cantidades de parámetros en tu main.

### **Pregunta:**

¿En qué se diferencia la sobrecarga de métodos de la sobrescritura de métodos? ¿Son conceptos similares?

#### **📘 Respuesta:**

La **sobrecarga** de métodos ocurre cuando tienes múltiples métodos con el mismo nombre en la misma clase, pero con diferentes tipos o cantidades de parámetros. En cambio, la **sobrescritura** (override) ocurre cuando una subclase proporciona una nueva implementación de un método que ya existe en la clase base, manteniendo la misma firma del método (nombre y parámetros).

## **4. Recapitulación y Desafío Final**

### **Resumen:**

* Los **constructores** se utilizan para inicializar objetos. Si no defines uno, Java crea un constructor por defecto.
* La **sobrecarga de constructores** te permite crear varias formas de inicializar objetos según los parámetros.
* La **sobrecarga de métodos** permite usar el mismo nombre de método con diferentes parámetros, proporcionando flexibilidad.

### **Desafío final:**

1. Crea una clase llamada Alumno con los siguientes atributos: nombre, edad y carrera.
2. Define tres constructores sobrecargados:
   * Uno que solo reciba el nombre.
   * Uno que reciba el nombre y la edad.
   * Uno que reciba el nombre, la edad y la carrera.
3. Define métodos sobrecargados llamados mostrarInfo que impriman diferentes cantidades de información del alumno según los datos que tengas (por ejemplo, solo el nombre, o el nombre y la edad, etc.).

String nombre;

int edad;

String carrera;

public Alumno(String nombre) {

this.nombre = nombre;

this.edad = 0; // Valor por defecto

this.carrera = "Desconocida"; // Valor por defecto

}

public Alumno(String nombre, int edad) {

this.nombre = nombre;

this.edad = edad;

this.carrera = "Desconocida"; // Valor por defecto

}

public Alumno(String nombre, int edad, String carrera) {

this.nombre = nombre;

this.edad = edad;

this.carrera = carrera;

}

public void mostrarInfo() {

System.out.println("Nombre: " + nombre);

}

public void mostrarInfo(boolean mostrarEdad) {

if (mostrarEdad) {

System.out.println("Nombre: " + nombre + ", Edad: " + edad);

} else {

mostrarInfo();

}

}

// Método sobrecargado para mostrar nombre, edad y carrera

public void mostrarInfo(boolean mostrarEdad, boolean mostrarCarrera) {

if (mostrarCarrera) {

System.out.println("Nombre: " + nombre + ", Edad: " + edad + ", Carrera: " + carrera);

} else {

mostrarInfo(mostrarEdad);

}

}

}

public class Main {

public static void main(String[] args) {

Alumno alumno1 = new Alumno("Ana");

Alumno alumno2 = new Alumno("Luis", 20);

Alumno alumno3 = new Alumno("María", 22, "Ingeniería");

System.out.println("Información del Alumno 1:");

alumno1.mostrarInfo();

System.out.println("Información del Alumno 2:");

alumno2.mostrarInfo(true); // Mostrar nombre y edad

System.out.println("Información del Alumno 3:");

alumno3.mostrarInfo(true, true); // Mostrar nombre, edad y carrera

}

}

java

Copiar código

class Alumno {

// Completa el código...

}

¡Felicidades por llegar hasta aquí! Este último desafío te permitirá practicar tanto la sobrecarga de constructores como la de métodos. Una vez que lo completes, habrás dominado estos conceptos clave de la programación orientada a objetos en Java.

### **Conclusión:**

Ahora entiendes cómo funcionan los **constructores** y la **sobrecarga** en Java, y has aplicado estos conceptos de manera práctica. Los constructores te permiten crear objetos correctamente inicializados, mientras que la sobrecarga te da flexibilidad para trabajar con métodos y constructores adaptados a diferentes situaciones. ¡Sigue practicando para fortalecer tus habilidades!

Memoria actualizada

### **Tutorial Interactivo: Herencia y Polimorfismo en Java**

**Basado en "Programación Orientada a Objetos en Java"**

¡Bienvenido al tutorial sobre **Herencia** y **Polimorfismo** en Java! Este tutorial está diseñado para que aprendas de manera activa y práctica. Vamos a explorar cómo funcionan estos conceptos clave de la programación orientada a objetos y cómo puedes aplicarlos en tus propios proyectos.

### **Objetivos del tutorial:**

* Entender el concepto de **herencia** y cómo se aplica en Java.
* Aprender sobre **polimorfismo** y cómo se utiliza para hacer que el código sea más flexible y extensible.
* Aplicar estos conceptos a través de ejemplos y ejercicios prácticos.

## **1. Herencia en Java**

### **Concepto clave:**

La **herencia** es un mecanismo que permite a una clase heredar atributos y métodos de otra clase. Esto facilita la reutilización de código y permite crear una jerarquía de clases.

### **Características de la herencia:**

* La clase que hereda se llama **subclase** o **clase derivada**.
* La clase de la cual se hereda se llama **superclase** o **clase base**.
* La subclase puede **extender** o **añadir** nuevas funcionalidades a los métodos y atributos de la superclase.

### **Ejemplo básico:**

java

Copiar código

// Superclase

class Animal {

String nombre;

public Animal(String nombre) {

this.nombre = nombre;

}

public void hacerSonido() {

System.out.println("El animal hace un sonido");

}

}

// Subclase

class Perro extends Animal {

public Perro(String nombre) {

super(nombre);

}

@Override

public void hacerSonido() {

System.out.println("El perro ladra");

}

}

public class Main {

public static void main(String[] args) {

Animal miAnimal = new Animal("Genérico");

Perro miPerro = new Perro("Rex");

miAnimal.hacerSonido(); // Salida: El animal hace un sonido

miPerro.hacerSonido(); // Salida: El perro ladra

}

}

### **📝 Tarea 1: Implementa Herencia**

1. Crea una clase Vehiculo con un atributo marca y un método mover.
2. Crea una subclase Coche que herede de Vehiculo y sobreescriba el método mover para especificar que el coche se mueve a alta velocidad.
3. En tu método main, crea un objeto de tipo Vehiculo y un objeto de tipo Coche. Llama al método mover para ambos objetos y observa la diferencia.

java

Copiar código

class Vehiculo {

// Completa el código...

}

class Coche extends Vehiculo {

// Completa el código...

}

public class Main {

public static void main(String[] args) {

// Completa el código...

}

}

### **Pregunta:**

¿Qué diferencia hay entre una subclase y una superclase? ¿Cómo se usa la palabra clave extends en Java?

#### **📘 Respuesta:**

* **Superclase**: La clase de la cual otras clases heredan. Contiene atributos y métodos que pueden ser compartidos.
* **Subclase**: La clase que hereda de otra clase. Puede usar, modificar o extender las funcionalidades de la superclase.
* La palabra clave extends se usa en Java para indicar que una clase es una subclase de otra. Por ejemplo, class Coche extends Vehiculo significa que Coche hereda de Vehiculo.

## **2. Polimorfismo en Java**

### **Concepto clave:**

El **polimorfismo** permite a los objetos de diferentes clases ser tratados como objetos de una clase base común. En Java, el polimorfismo se logra principalmente a través de la **sobrescritura de métodos** y la **referencia de objetos**.

### **Tipos de Polimorfismo:**

* **Polimorfismo de tiempo de compilación**: Se refiere a la sobrecarga de métodos (métodos con el mismo nombre pero con diferentes parámetros).
* **Polimorfismo de tiempo de ejecución**: Se refiere a la sobrescritura de métodos (una subclase proporciona una implementación específica de un método que ya está definido en la superclase).

### **Ejemplo de Polimorfismo de Tiempo de Ejecución:**

java

Copiar código

class Animal {

public void hacerSonido() {

System.out.println("El animal hace un sonido");

}

}

class Perro extends Animal {

@Override

public void hacerSonido() {

System.out.println("El perro ladra");

}

}

class Gato extends Animal {

@Override

public void hacerSonido() {

System.out.println("El gato maúlla");

}

}

public class Main {

public static void main(String[] args) {

Animal miAnimal;

miAnimal = new Perro();

miAnimal.hacerSonido(); // Salida: El perro ladra

miAnimal = new Gato();

miAnimal.hacerSonido(); // Salida: El gato maúlla

}

}

### **📝 Tarea 2: Implementa Polimorfismo**

1. Crea una clase base llamada Figura con un método dibujar.
2. Crea dos subclases: Circulo y Cuadrado. Cada una debe sobrescribir el método dibujar para imprimir un mensaje específico sobre cómo se dibuja la figura.
3. En tu método main, crea un arreglo de Figura y agrega instancias de Circulo y Cuadrado. Usa un bucle para llamar al método dibujar de cada figura y observa el resultado.

java

Copiar código

public class Main {

public static void main(String[] args) {

abstract class Figura {

// Método abstracto que debe ser sobrescrito por las subclases

public abstract void dibujar();

}

class Circulo extends Figura {

@Override

public void dibujar() {

System.out.println("Dibujando un círculo.");

}

}

class Cuadrado extends Figura {

@Override

public void dibujar() {

System.out.println("Dibujando un cuadrado.");

}

}

public class Main {

public static void main(String[] args) {

Figura[] figuras = new Figura[2];

figuras[0] = new Circulo();

figuras[1] = new Cuadrado();

figura

for (Figura figura : figuras) {

figura.dibujar();

}

}

}

}

}

### **Pregunta:**

¿Por qué es ventajoso utilizar polimorfismo en la programación orientada a objetos? ¿Qué beneficios proporciona al diseño del software?

#### **📘 Respuesta:**

El polimorfismo es ventajoso porque permite que un objeto sea tratado como una instancia de su clase base, lo que facilita el manejo de diferentes tipos de objetos de manera uniforme. Los beneficios incluyen:

* **Flexibilidad**: Puedes escribir código que funciona con objetos de diferentes clases sin saber los detalles específicos de cada clase.
* **Extensibilidad**: Puedes agregar nuevas clases sin modificar el código que ya usa la clase base.
* **Mantenimiento**: Facilita la modificación y extensión del código al permitir el uso de interfaces y clases base en lugar de clases concretas.

## **3. Recapitulación y Desafío Final**

### **Resumen:**

* **Herencia** permite reutilizar y extender funcionalidades de una clase base.
* **Polimorfismo** permite tratar objetos de diferentes clases a través de una interfaz común, facilitando el manejo de clases relacionadas y mejorando la flexibilidad y extensibilidad del código.

### **Desafío final:**

1. Crea una clase base llamada Empleado con atributos nombre y salario, y un método calcularPago.
2. Crea dos subclases: EmpleadoTiempoCompleto y EmpleadoMedioTiempo. Cada una debe sobrescribir el método calcularPago para calcular el salario según el tipo de empleado.
3. En tu método main, crea un arreglo de Empleado que contenga instancias de EmpleadoTiempoCompleto y EmpleadoMedioTiempo. Usa un bucle para calcular y mostrar el pago de cada empleado.

java

Copiar código

abstract class Empleado {

String nombre;

double salario;

public Empleado(String nombre, double salario) {

this.nombre = nombre;

this.salario = salario;

}

public abstract double calcularPago();

}

class EmpleadoTiempoCompleto extends Empleado {

public EmpleadoTiempoCompleto(String nombre, double salario) {

super(nombre, salario);

}

@Override

public double calcularPago() {

pago total

return salario;

}

}

class EmpleadoMedioTiempo extends Empleado {

int horasTrabajadas;

double tarifaPorHora;

public EmpleadoMedioTiempo(String nombre, int horasTrabajadas, double tarifaPorHora) {

super(nombre, 0);

this.horasTrabajadas = horasTrabajadas;

this.tarifaPorHora = tarifaPorHora;

}

@Override

public double calcularPago() {

return horasTrabajadas \* tarifaPorHora;

}

}

public class Main {

public static void main(String[] args) {

Empleado[] empleados = new Empleado[2];

empleados[0] = new EmpleadoTiempoCompleto("Juan", 3000);

empleados[1] = new EmpleadoMedioTiempo("Ana", 120, 15); // 120 horas a 15 por hora

empleado

for (Empleado empleado : empleados) {

System.out.println("Empleado: " + empleado.nombre + ", Pago: $" + empleado.calcularPago());

}

}

}

### **Conclusión:**

Ahora comprendes cómo la **herencia** permite reutilizar y extender código, y cómo el **polimorfismo** facilita el manejo flexible de objetos. Estos conceptos te ayudarán a escribir código más eficiente y mantenible. ¡Sigue practicando para consolidar estos conocimientos y aplicarlos en tus proyectos!

### **Tutorial Interactivo: Clases Abstractas e Interfaces en Java**

**Basado en "Programación Orientada a Objetos en Java"**

¡Bienvenido al tutorial sobre **Clases Abstractas** e **Interfaces** en Java! Este tutorial está diseñado para que aprendas de manera activa y práctica sobre estos dos conceptos clave de la programación orientada a objetos.

### **Objetivos del Tutorial:**

* Comprender el concepto de **clases abstractas** y cómo se utilizan en Java.
* Aprender sobre **interfaces** y cómo se utilizan para definir contratos en el código.
* Aplicar estos conceptos a través de ejemplos y ejercicios prácticos.

## **1. Clases Abstractas en Java**

### **Concepto clave:**

Una **clase abstracta** es una clase que no se puede instanciar directamente y que puede contener métodos abstractos, es decir, métodos sin implementación. Las subclases deben proporcionar la implementación de estos métodos.

### **Características de las Clases Abstractas:**

* **No se pueden instanciar**: No puedes crear objetos directamente de una clase abstracta.
* **Métodos abstractos**: Métodos sin cuerpo que deben ser implementados por las subclases.
* **Métodos concretos**: Pueden contener métodos con implementación que las subclases heredan.

### **Ejemplo Básico:**

java

Copiar código

// Clase abstracta

abstract class Animal {

String nombre;

public Animal(String nombre) {

this.nombre = nombre;

}

// Método abstracto

public abstract void hacerSonido();

// Método concreto

public void dormir() {

System.out.println(nombre + " está durmiendo.");

}

}

// Subclase concreta

class Perro extends Animal {

public Perro(String nombre) {

super(nombre);

}

@Override

public void hacerSonido() {

System.out.println("El perro ladra.");

}

}

public class Main {

public static void main(String[] args) {

// Crear una instancia de Perro

Animal miPerro = new Perro("Rex");

miPerro.hacerSonido(); // Salida: El perro ladra.

miPerro.dormir(); // Salida: Rex está durmiendo.

}

}

### **📝 Tarea 1: Implementa una Clase Abstracta**

1. Crea una clase abstracta llamada Vehiculo con atributos marca y velocidadMaxima.
2. Define un método abstracto mover().
3. Crea una subclase Coche que herede de Vehiculo y proporciona una implementación concreta del método mover().
4. En tu método main, crea una instancia de Coche y llama al método mover().

java

Copiar código

**abstract class Vehiculo {**

**String marca;**

**double velocidadMaxima;**

**public Vehiculo(String marca, double velocidadMaxima) {**

**this.marca = marca;**

**this.velocidadMaxima = velocidadMaxima;**

**}**

**public abstract void mover();**

**}**

**public Coche(String marca, double velocidadMaxima) {**

**super(marca, velocidadMaxima);**

**}**

**@Override**

**public void mover() {**

**System.out.println("El coche " + marca + " se está moviendo a " + velocidadMaxima + " km/h.");**

**}**

**}**

**public class Main {**

**public static void main(String[] args) {**

**Coche miCoche = new Coche("Toyota", 180);**

**miCoche.mover(); // Salida: El coche Toyota se está moviendo a 180 km/h.**

**}**

**}**

### **Pregunta:**

¿Cuál es la diferencia principal entre una clase abstracta y una clase concreta? ¿Cuándo usarías una clase abstracta en lugar de una clase concreta?

La principal diferencia entre una **clase abstracta** y una **clase concreta** en Java radica en su propósito y en cómo se utilizan en la programación orientada a objetos

## **2. Interfaces en Java**

### **Concepto clave:**

Una **interface** define un contrato que las clases deben seguir. Una interfaz puede contener métodos abstractos (sin implementación) y métodos por defecto (con implementación). Las clases que implementan una interfaz deben proporcionar implementaciones para los métodos abstractos de la interfaz.

### **Características de las Interfaces:**

* **Métodos abstractos**: Métodos sin cuerpo que deben ser implementados por las clases que implementan la interfaz.
* **Métodos por defecto**: Métodos con implementación que pueden ser utilizados por las clases que implementan la interfaz.
* **No pueden contener estado**: Las interfaces no pueden tener atributos de instancia, solo constantes.

### **Ejemplo Básico:**

java

Copiar código

// Interfaz

interface Volador {

void volar();

}

// Clase que implementa la interfaz

class Ave implements Volador {

@Override

public void volar() {

System.out.println("El ave está volando.");

}

}

public class Main {

public static void main(String[] args) {

// Crear una instancia de Ave

Volador miAve = new Ave();

miAve.volar(); // Salida: El ave está volando.

}

}

### **📝 Tarea 2: Implementa una Interfaz**

1. Define una interfaz llamada Imprimible con un método imprimir().
2. Crea una clase Documento que implemente Imprimible y proporcione una implementación del método imprimir().
3. En tu método main, crea una instancia de Documento y llama al método imprimir().

interface Imprimible {

void imprimir();

}

class Documento implements Imprimible {

private String contenido;

public Documento(String contenido) {

this.contenido = contenido;

}

@Override

public void imprimir() {

System.out.println("Imprimiendo documento: " + contenido);

}

}

public class Main {

public static void main(String[] args) {

Documento miDocumento = new Documento("Este es el contenido del documento.");

miDocumento.imprimir(); // Salida: Imprimiendo documento: Este es el contenido del documento.

}

}

### **Pregunta:**

¿Cuál es la ventaja de usar interfaces en comparación con clases abstractas? ¿En qué situaciones es más adecuado usar interfaces?

**Multiplicidad de Implementaciones**:

* **Interfaces**: Una clase puede implementar múltiples interfaces, lo que permite que una clase tenga varias capacidades o roles. Esto proporciona una forma de lograr la **composición de tipos**.
* **Clases Abstractas**: Una clase solo puede extender una única clase abstracta debido a la herencia simple en Java. Esto puede ser limitante si una clase necesita heredar comportamientos de múltiples fuentes.

## **3. Recapitulación y Desafío Final**

### **Resumen:**

* **Clases Abstractas**: Permiten definir métodos abstractos que deben ser implementados por las subclases y pueden tener métodos concretos con implementación. No se pueden instanciar directamente.
* **Interfaces**: Definen un contrato que las clases deben seguir. Las clases implementan interfaces y deben proporcionar implementación para los métodos abstractos. Pueden contener métodos por defecto.

### **Desafío Final:**

1. Crea una interfaz Responsable con un método responsabilidad().
2. Crea dos clases que implementen Responsable: Gerente y Desarrollador. Cada una debe proporcionar una implementación específica del método responsabilidad().
3. En tu método main, crea un arreglo de Responsable que contenga instancias de Gerente y Desarrollador. Usa un bucle para llamar al método responsabilidad() de cada objeto.

java

Copiar código

interface Responsable {

void responsabilidad();

}

class Gerente implements Responsable {

@Override

public void responsabilidad() {

System.out.println("El gerente supervisa y coordina al equipo.");

}

}

class Desarrollador implements Responsable {

@Override

public void responsabilidad() {

System.out.println("El desarrollador escribe y mantiene el código.");

}

}

public class Main {

public static void main(String[] args) {

Responsable[] responsables = new Responsable[2];

responsables[0] = new Gerente();

responsables[1] = new Desarrollador();

for (Responsable r : responsables) {

r.responsabilidad();

}

}

}

### **Conclusión:**

Ahora comprendes cómo las **clases abstractas** y las **interfaces** pueden ayudarte a diseñar un código más flexible y extensible. Ambos conceptos son fundamentales para la programación orientada a objetos y te permitirán crear sistemas más organizados y mantenibles.

¡Sigue practicando y aplicando estos conceptos en tus proyectos para fortalecer tu comprensión y habilidades en programación orientada a objetos!

### **Tutorial Interactivo: ArrayList en Java**

Bienvenido al tutorial sobre ArrayList en Java. En este tutorial, aprenderás qué es un ArrayList, cómo usarlo y cuándo es útil en la programación orientada a objetos. Vamos a explorar sus conceptos clave, realizar ejercicios prácticos y responder a preguntas para asegurar una comprensión sólida.

#### **1. Introducción a ArrayList**

**Concepto Clave:** ArrayList es una clase en Java que pertenece al paquete java.util. Es una implementación de la interfaz List y permite almacenar elementos dinámicamente, es decir, puedes agregar, eliminar y acceder a elementos en cualquier momento. A diferencia de los arreglos tradicionales, un ArrayList puede cambiar de tamaño automáticamente.

**Ventajas de usar ArrayList:**

* **Redimensionamiento Automático:** No necesitas definir un tamaño fijo.
* **Métodos de Manipulación de Datos:** Proporciona métodos para agregar, eliminar y buscar elementos de manera sencilla.

**Ejemplo Básico:**

java

Copiar código

import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

ArrayList<String> lista = new ArrayList<>();

lista.add("Manzana");

lista.add("Banana");

lista.add("Cereza");

System.out.println(lista); // Salida: [Manzana, Banana, Cereza]

}

}

#### **2. Creación de un ArrayList**

**Concepto Clave:** Para crear un ArrayList, necesitas importar el paquete java.util.ArrayList y luego crear una instancia de ArrayList especificando el tipo de datos que almacenará.

**Tarea:**

1. **Crea un nuevo proyecto en tu IDE favorito.**
2. **Agrega el siguiente código para crear un ArrayList de números enteros y agregar algunos elementos.**

java

Copiar código

import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

// Crear un ArrayList de enteros

ArrayList<Integer> numeros = new ArrayList<>();

// Agregar elementos

numeros.add(10);

numeros.add(20);

numeros.add(30);

// Imprimir la lista

System.out.println(numeros);

}

}

**Pregunta:** ¿Qué método se usa para agregar elementos a un ArrayList?

**Respuesta:** Se usa el método add().

#### **3. Acceso y Manipulación de Elementos**

**Concepto Clave:** Puedes acceder a los elementos en un ArrayList mediante el índice, similar a un arreglo. También puedes modificar y eliminar elementos.

**Métodos Clave:**

* get(int index): Obtiene el elemento en el índice especificado.
* set(int index, E element): Reemplaza el elemento en el índice especificado.
* remove(int index): Elimina el elemento en el índice especificado.

**Ejercicio:**

1. **Modifica el código para acceder al primer elemento, cambiar el segundo y eliminar el tercer elemento.**

java

Copiar código

import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

ArrayList<String> frutas = new ArrayList<>();

frutas.add("Manzana");

frutas.add("Banana");

frutas.add("Cereza");

// Acceder al primer elemento

System.out.println("Primer elemento: " + frutas.get(0));

// Cambiar el segundo elemento

frutas.set(1, "Pera");

// Eliminar el tercer elemento

frutas.remove(2);

// Imprimir la lista actualizada

System.out.println(frutas);

}

}

**Pregunta:** ¿Cómo puedes obtener el número de elementos en un ArrayList?

**Respuesta:** Usa el método size().

#### **4. Iteración sobre un ArrayList**

**Concepto Clave:** Puedes iterar sobre los elementos de un ArrayList utilizando un bucle for o un bucle for-each.

**Ejercicio:**

1. **Itera sobre el ArrayList de frutas utilizando ambos tipos de bucles para imprimir cada elemento.**

java

Copiar código

import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

ArrayList<String> frutas = new ArrayList<>();

frutas.add("Manzana");

frutas.add("Pera");

frutas.add("Cereza");

// Usando un bucle for

System.out.println("Usando un bucle for:");

for (int i = 0; i < frutas.size(); i++) {

System.out.println(frutas.get(i));

}

// Usando un bucle for-each

System.out.println("Usando un bucle for-each:");

for (String fruta : frutas) {

System.out.println(fruta);

}

}

}

**Pregunta:** ¿Qué diferencia hay entre un bucle for y un bucle for-each al iterar sobre un ArrayList?

**Respuesta:** El bucle for usa un índice para acceder a los elementos, mientras que el bucle for-each itera directamente sobre los elementos sin necesidad de manejar índices.

#### **5. Otros Métodos Útiles**

**Concepto Clave:** ArrayList proporciona otros métodos útiles como clear() para eliminar todos los elementos, contains(Object o) para verificar la existencia de un elemento, y isEmpty() para comprobar si la lista está vacía.

**Ejercicio:**

1. **Usa los métodos clear(), contains(), e isEmpty() en el siguiente código.**

java

Copiar código

import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

ArrayList<String> frutas = new ArrayList<>();

frutas.add("Manzana");

frutas.add("Pera");

// Verificar si la lista contiene un elemento

System.out.println("Contiene Pera: " + frutas.contains("Pera"));

// Vaciar la lista

frutas.clear();

// Verificar si la lista está vacía

System.out.println("La lista está vacía: " + frutas.isEmpty());

}

}

**Pregunta:** ¿Qué método usarías para verificar si un ArrayList contiene un elemento específico?

**Respuesta:** Usa el método contains().

### **Resumen y Conclusión**

Has aprendido cómo trabajar con ArrayList en Java. Ahora sabes cómo:

* Crear y manipular ArrayList.
* Acceder, modificar y eliminar elementos.
* Iterar sobre los elementos.
* Usar métodos adicionales para verificar y gestionar la lista.

**Próximos Pasos:**

* Experimenta creando ArrayList de diferentes tipos de datos.
* Intenta combinar ArrayList con otras estructuras de datos y clases en tus proyectos.

¡Continúa practicando y aplicando lo aprendido en tus propios proyectos para reforzar tus conocimientos sobre ArrayList!

### **Tutorial Interactivo: Agregación de Clases en Java**

¡Bienvenido al tutorial sobre la **agregación de clases** en Java! En este tutorial, aprenderás qué es la agregación, cómo se utiliza en la programación orientada a objetos y cómo implementarla en Java. Vamos a explorar los conceptos clave, realizar ejercicios prácticos y responder a preguntas para asegurarnos de que comprendas bien el tema.

#### **1. Introducción a la Agregación**

**Concepto Clave:** La **agregación** es un tipo de relación entre clases en la que una clase (la clase contenedora) tiene una referencia a una o más instancias de otra clase (la clase contenida). La relación de agregación indica una relación "tiene un" donde la clase contenedora "tiene" o "usa" la clase contenida.

La diferencia principal entre **agregación** y **composición** es que en la agregación, la vida del objeto contenido no está directamente ligada a la vida del objeto contenedor.

**Ejemplo Conceptual:**

* Un **Departamento** puede tener muchos **Empleados**.
* Un **Coche** puede tener muchos **Ruedas**.

**Diagrama UML Básico:**

markdown

Copiar código

Departamento

----------------

- nombre: String

- empleados: List<Empleado>

Empleado

----------------

- nombre: String

#### **2. Definición de Clases y Agregación**

**Tarea:**

1. **Crea una clase Empleado con un atributo nombre.**
2. **Crea una clase Departamento que tenga una lista de Empleado como atributo.**

**Código:**

java

Copiar código

import java.util.ArrayList;

import java.util.List;

// Clase Empleado

class Empleado {

private String nombre;

public Empleado(String nombre) {

this.nombre = nombre;

}

public String getNombre() {

return nombre;

}

}

// Clase Departamento

class Departamento {

private String nombre;

private List<Empleado> empleados;

public Departamento(String nombre) {

this.nombre = nombre;

this.empleados = new ArrayList<>();

}

public void agregarEmpleado(Empleado empleado) {

empleados.add(empleado);

}

public void mostrarEmpleados() {

System.out.println("Empleados en el departamento " + nombre + ":");

for (Empleado e : empleados) {

System.out.println("- " + e.getNombre());

}

}

}

**Pregunta:** ¿Qué tipo de relación entre Departamento y Empleado estamos modelando aquí?

**Respuesta:** Estamos modelando una relación de agregación, donde un Departamento contiene una lista de Empleado.

#### **3. Uso de Agregación en el Método Main**

**Ejercicio:**

1. **En tu método main, crea instancias de Empleado y un Departamento.**
2. **Agrega los empleados al departamento y muestra la lista de empleados.**

**Código:**

java

Copiar código

public class Main {

public static void main(String[] args) {

Empleado emp1 = new Empleado("Ana");

Empleado emp2 = new Empleado("Luis");

Departamento dep = new Departamento("Recursos Humanos");

dep.agregarEmpleado(emp1);

dep.agregarEmpleado(emp2);

dep.mostrarEmpleados();

}

}

**Resultado esperado:**

diff

Copiar código

Empleados en el departamento Recursos Humanos:

- Ana

- Luis

#### **4. Modificación y Eliminación de Elementos**

**Concepto Clave:** Puedes modificar y eliminar elementos en una relación de agregación de manera que los cambios en la lista de empleados no afectan a los empleados en sí mismos, ya que los empleados existen independientemente del departamento.

**Tarea:**

1. **Agrega un método en la clase Departamento para eliminar un empleado por nombre.**
2. **Modifica el método main para eliminar un empleado y mostrar la lista actualizada.**

**Código:**

java

class Departamento {

private String nombre;

private List<Empleado> empleados;

public Departamento(String nombre) {

this.nombre = nombre;

this.empleados = new ArrayList<>();

}

public void agregarEmpleado(Empleado empleado) {

empleados.add(empleado);

}

public void eliminarEmpleado(String nombre) {

empleados.removeIf(e -> e.getNombre().equals(nombre));

}

public void mostrarEmpleados() {

System.out.println("Empleados en el departamento " + nombre + ":");

for (Empleado e : empleados) {

System.out.println("- " + e.getNombre());

}

}

}

public class Main {

public static void main(String[] args) {

Empleado emp1 = new Empleado("Ana");

Empleado emp2 = new Empleado("Luis");

Departamento dep = new Departamento("Recursos Humanos");

dep.agregarEmpleado(emp1);

dep.agregarEmpleado(emp2);

dep.mostrarEmpleados();

dep.eliminarEmpleado("Ana");

dep.mostrarEmpleados();

}

}

**Resultado esperado:**

Empleados en el departamento Recursos Humanos:

- Ana

- Luis

Empleados en el departamento Recursos Humanos:

- Luis

**Pregunta:** ¿Cómo se maneja la eliminación de un empleado en la agregación? ¿Qué pasa con el objeto Empleado después de eliminarlo de la lista?

**Respuesta:** La eliminación del empleado de la lista no afecta al objeto Empleado en sí; el objeto sigue existiendo en la memoria hasta que sea recolectado por el recolector de basura si no hay otras referencias a él.

#### **5. Resumen y Buenas Prácticas**

**Conceptos Clave:**

* **Agregación**: Relación donde una clase contiene instancias de otra clase, pero las instancias contenidas pueden existir independientemente de la clase contenedora.
* **Modificación y Eliminación**: Puedes modificar la colección de objetos contenidos sin afectar a los objetos en sí mismos.
* **Uso Eficiente**: Utiliza agregación cuando quieras que una clase tenga referencias a otros objetos pero sin una relación de dependencia fuerte.

### **Tutorial Interactivo: Funciones de Cadena en Java**

¡Bienvenido al tutorial sobre **funciones de cadena** en Java! En este tutorial, exploraremos cómo trabajar con cadenas en Java, utilizando las funciones disponibles en la clase String. A lo largo del tutorial, aprenderás los conceptos clave, participarás en ejercicios prácticos y responderás a preguntas para reforzar tus conocimientos.

#### **1. Introducción a la Clase String**

**Concepto Clave:** En Java, las cadenas se representan con la clase String. Esta clase ofrece una variedad de métodos para manipular y analizar texto.

**Métodos Clave:**

* length(): Retorna la longitud de la cadena.
* charAt(int index): Retorna el carácter en el índice especificado.
* substring(int beginIndex, int endIndex): Retorna una subcadena desde beginIndex hasta endIndex.

**Ejemplo Básico:**

public class Main {

public static void main(String[] args) {

String texto = "Hola, Mundo!";

System.out.println("Longitud: " + texto.length()); // Salida: 12

System.out.println("Carácter en índice 7: " + texto.charAt(7));

System.out.println("Subcadena: " + texto.substring(0, 5));

}

}

**Pregunta:** ¿Qué método usarías para obtener el carácter en una posición específica de una cadena?

**Respuesta:** Usarías el método charAt(int index).

#### **2. Métodos de Manipulación de Cadenas**

**Concepto Clave:** La clase String proporciona varios métodos para manipular el contenido de las cadenas, como concat(), replace(), y trim().

**Tarea:**

1. **Crea un programa que use los siguientes métodos:**
   * concat(String str): Une dos cadenas.
   * replace(CharSequence target, CharSequence replacement): Reemplaza todas las ocurrencias de una subcadena por otra.
   * trim(): Elimina espacios en blanco al principio y al final de la cadena.

public class Main {

public static void main(String[] args) {

String saludo = " Hola, ";

String nombre = "Juan ";

String mensaje = saludo.concat(nombre);

System.out.println("Mensaje concatenado: " + mensaje.trim());

String mensajeModificado = mensaje.replace("Juan", "Pedro");

System.out.println("Mensaje modificado: " + mensajeModificado.trim());

}

}

**Pregunta:** ¿Cómo puedes eliminar los espacios en blanco al principio y al final de una cadena?

**Respuesta:** Usa el método trim().

#### **3. Comparación y Búsqueda en Cadenas**

**Concepto Clave:** Los métodos para comparar y buscar en cadenas son esenciales para manejar textos. Algunos métodos importantes son equals(), equalsIgnoreCase(), indexOf(), y contains().

**Ejercicio:**

1. **Usa los siguientes métodos para comparar y buscar en una cadena:**
   * equals(String anotherString): Compara dos cadenas para ver si son iguales.
   * indexOf(String str): Retorna el índice de la primera ocurrencia de la subcadena.
   * contains(CharSequence sequence): Verifica si la cadena contiene la subcadena.

**Código:**

public class Main {

public static void main(String[] args) {

String texto = "Java es un lenguaje de programación";

String otraCadena = "Java es un lenguaje de programación";

System.out.println("Las cadenas son iguales: " + texto.equals(otraCadena)); // Salida: true

int indice = texto.indexOf("lenguaje");

System.out.println("Índice de 'lenguaje': " + indice); // Salida: 12

boolean contiene = texto.contains("programación");

System.out.println("Contiene 'programación': " + contiene);

}

}

**Pregunta:** ¿Qué método usarías para verificar si una cadena contiene otra cadena específica?

**Respuesta:** Usarías el método contains(CharSequence sequence).

#### **4. Conversión y Formateo de Cadenas**

**Concepto Clave:** Puedes convertir y formatear cadenas usando métodos como toLowerCase(), toUpperCase(), y format().

**Ejercicio:**

1. **Crea un programa que convierta una cadena a mayúsculas y minúsculas, y formatee una cadena con variables.**

**Código:**

public class Main {

public static void main(String[] args) {

String texto = "Java es Genial!";

System.out.println("En mayúsculas: " + texto.toUpperCase());

System.out.println("En minúsculas: " + texto.toLowerCase());

String nombre = "Alice";

int edad = 30;

String mensaje = String.format("Hola, mi nombre es %s y tengo %d años.", nombre, edad);

System.out.println(mensaje); // Salida: Hola, mi nombre es Alice y tengo 30 años.

}

}

**Pregunta:** ¿Cómo puedes convertir una cadena a mayúsculas en Java?

**Respuesta:** Usa el método toUpperCase().

#### **5. Resumen y Buenas Prácticas**

**Conceptos Clave:**

* **Manipulación de Cadenas:** Usa métodos como concat(), replace(), y trim() para manipular cadenas.
* **Comparación y Búsqueda:** Usa equals(), indexOf(), y contains() para comparar y buscar dentro de cadenas.
* **Conversión y Formateo:** Usa toUpperCase(), toLowerCase(), y format() para convertir y formatear texto.

**Práctica Adicional:**

* Experimenta con la creación de métodos personalizados que utilicen las funciones de cadena para resolver problemas específicos.
* Crea un programa que procese un archivo de texto y utilice varios métodos de cadena para analizar y manipular el contenido.

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

public class ProcesadorDeTexto {

public static void main(String[] args) {

String archivo = "texto.txt";

procesarArchivo(archivo);

}

public static void procesarArchivo(String archivo) {

try (BufferedReader br = new BufferedReader(new FileReader(archivo))) {

String linea;

while ((linea = br.readLine()) != null) {

System.out.println("Línea original: " + linea);

int numeroDePalabras = contarPalabras(linea);

System.out.println("Número de palabras: " + numeroDePalabras);

String textoReemplazado = reemplazarPalabra(linea, "Java", "Python");

System.out.println("Texto con palabras reemplazadas: " + textoReemplazado);

String textoMayusculas = linea.toUpperCase();

String textoMinusculas = linea.toLowerCase();

System.out.println("Texto en mayúsculas: " + textoMayusculas);

System.out.println("Texto en minúsculas: " + textoMinusculas);

System.out.println(); // Línea en blanco para separar los resultados

}

} catch (IOException e) {

System.out.println("Error al leer el archivo: " + e.getMessage());

}

}

public static int contarPalabras(String texto) {

if (texto == null || texto.isEmpty()) {

return 0;

}

String[] palabras = texto.trim().split("\\s+");

return palabras.length;

}

public static String reemplazarPalabra(String texto, String palabraOriginal, String palabraNueva) {

if (texto == null || palabraOriginal == null || palabraNueva == null) {

return texto;

}

return texto.replace(palabraOriginal, palabraNueva);

}

}

¡Felicidades por completar el tutorial! Sigue practicando para fortalecer tu comprensión de las funciones de cadena en Java y aplicarlas en tus proyectos.

### **Tutorial Interactivo: Métodos y Variables Static en Java**

¡Bienvenido al tutorial sobre **métodos y variables static** en Java! En este tutorial, aprenderás qué son los métodos y variables static, cómo se utilizan y por qué son importantes en la programación orientada a objetos.

### **1. Conceptos Clave**

#### **Variables Static**

**Definición:**

* Una variable static es una variable de clase, compartida por todas las instancias de la clase. Se inicializa solo una vez y su valor es común para todas las instancias.

**Características:**

* **Acceso**: Se accede mediante el nombre de la clase o una instancia.
* **Inicialización**: Se inicializa una sola vez, cuando se carga la clase.
* **Uso**: Útil para valores constantes o para compartir datos entre todas las instancias de una clase.

**Ejemplo Básico:**

java

Copiar código

public class Contador {

// Variable estática

static int conteo = 0;

// Constructor

public Contador() {

conteo++;

}

// Método para mostrar el conteo

public void mostrarConteo() {

System.out.println("Conteo: " + conteo);

}

}

**Pregunta:** ¿Qué sucede con el valor de una variable static cuando se crean múltiples instancias de una clase?

**Respuesta:** El valor de una variable static es compartido entre todas las instancias de la clase. Se inicializa una vez y su valor es el mismo para todas las instancias.

#### **Métodos Static**

**Definición:**

* Un método static pertenece a la clase en lugar de a una instancia específica. Se puede llamar sin crear una instancia de la clase.

**Características:**

* **Acceso**: Se puede llamar directamente usando el nombre de la clase.
* **Uso**: Ideal para operaciones que no dependen del estado de las instancias de la clase.

**Ejemplo Básico:**

java

Copiar código

public class Calculadora {

// Método estático

public static int sumar(int a, int b) {

return a + b;

}

}

**Pregunta:** ¿Cómo puedes llamar a un método static sin crear una instancia de la clase?

**Respuesta:** Puedes llamar a un método static usando el nombre de la clase seguido del nombre del método, por ejemplo: Calculadora.sumar(5, 3);.

### **2. Ejercicios Prácticos**

#### **Ejercicio 1: Uso de Variables Static**

1. **Crea una clase llamada Banco con una variable static saldoTotal.**
2. **Cada vez que se cree una instancia de Banco, incrementa saldoTotal en una cantidad específica.**
3. **Implementa un método mostrarSaldoTotal() para imprimir el valor de saldoTotal.**

**Código:**

public class Banco {

static double saldoTotal = 0;

public Banco(double cantidad) {

saldoTotal += cantidad;

}

public static void mostrarSaldoTotal() {

System.out.println("Saldo Total en el Banco: " + saldoTotal);

}

public static void main(String[] args) {

Banco cuenta1 = new Banco(500.0);

Banco cuenta2 = new Banco(300.0);

Banco.mostrarSaldoTotal(); }

}

**Pregunta:** ¿Cómo se actualiza el valor de saldoTotal cada vez que se crea una nueva instancia de Banco?

**Respuesta:** El valor de saldoTotal se actualiza porque la variable es estática y se comparte entre todas las instancias. Cada vez que se crea una nueva instancia, el constructor agrega una cantidad al saldoTotal.

#### **Ejercicio 2: Uso de Métodos Static**

1. **Crea una clase llamada Conversor con un método static convertirKilometrosAMillas(double kilometros).**
2. **El método debe convertir kilómetros a millas usando la fórmula: millas = kilómetros \* 0.621371.**
3. **En el método main, llama al método convertirKilometrosAMillas sin crear una instancia de Conversor.**

**Código:**

public class Conversor {

// Método estático

public static double convertirKilometrosAMillas(double kilometros) {

return kilometros \* 0.621371;

}

public static void main(String[] args) {

double kilometros = 10.0;

double millas = Conversor.convertirKilometrosAMillas(kilometros);

System.out.println(kilometros + " kilómetros son " + millas + " millas.");

}

}

**Pregunta:** ¿Por qué es adecuado usar un método static para realizar conversiones o cálculos que no dependen de las instancias de la clase?

**Respuesta:** Un método static es adecuado porque no necesita acceder a datos específicos de una instancia. Realiza una operación que es independiente del estado de las instancias y puede ser utilizada directamente a través de la clase.

### **3. Resumen y Buenas Prácticas**

**Conceptos Clave:**

* **Variables Static:** Son compartidas entre todas las instancias de una clase. Útiles para valores constantes o compartidos.
* **Métodos Static:** Pertenecen a la clase y no a una instancia específica. Útiles para operaciones independientes del estado de la instancia.

**Práctica Adicional:**

* **Refactorización:** Examina tus clases existentes y considera si puedes utilizar variables o métodos static para mejorar la eficiencia.
* **Desafíos:** Crea una clase RegistroEstudiantes con un método static para contar el número de estudiantes registrados, y una variable static para almacenar la matrícula total.

private static int totalEstudiantes = 0;

private static double matriculaTotal = 0.0;

public static void registrarEstudiante(double matricula) {

totalEstudiantes++;

matriculaTotal += matricula;

}

public static void mostrarRegistro() {

System.out.println("Número total de estudiantes: " + totalEstudiantes);

System.out.println("Matrícula total: " + matriculaTotal);

}

RegistroEstudiantes.registrarEstudiante(1500.0);

RegistroEstudiantes.registrarEstudiante(1200.0);

RegistroEstudiantes.registrarEstudiante(2000.0);

RegistroEstudiantes.mostrarRegistro();

}

}

¡Felicidades por completar el tutorial sobre métodos y variables static en Java! Sigue practicando y aplicando estos conceptos para mejorar tu habilidad en programación orientada a objetos.