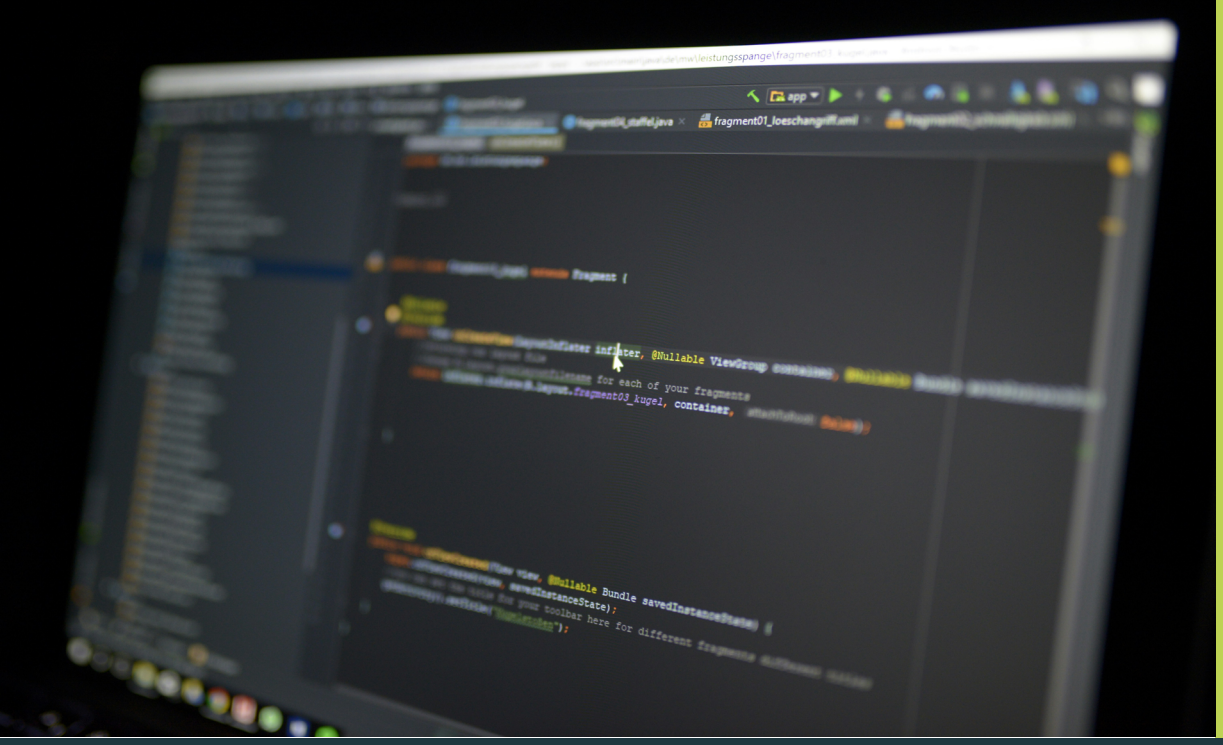

Unidad 5

Introducción a la orientación de objetos

Contenidos

Parte 1: Clases. Atributos, métodos y visibilidad

1. ¿Qué es una clase?

En Java, una **clase** es una plantilla o modelo a partir de la cual se crean objetos. Define los **atributos** (características) y **métodos** (comportamientos) que tendrán los objetos basados en esa clase.

Estructura básica de una clase en Java

```
public class NombreClase {
    // Atributos
    private Tipo atributo1;
    private Tipo atributo2;
    // Constructor
    public NombreClase(Tipo atributo1, Tipo atributo2) {
          this.atributo1 = atributo1;
          this.atributo2 = atributo2;
    }
    // Métodos
    public void metodoEjemplo() {
          System.out.println("Esto es un método.");
    }
}
```

Ejemplo: Clase Persona

```
public class Persona {
    // Atributos
    private String nombre;
    private int edad;
    // Constructor
    public Persona(String nombre, int edad) {
          this.nombre = nombre;
          this.edad = edad;
    }
    // Método para mostrar información
    public void mostrarInformacion() {
          System.out.println("Nombre: " + nombre + ", Edad: " + edad);
    }
}
```

2. Atributos

Los **atributos** son las variables que almacenan el estado de un objeto. Se definen dentro de una clase y pueden ser de diferentes tipos: primitivos (`int`, `double`, `boolean`) o referencias (como objetos de otras clases).

Características de los atributos:

- **Alcance**: Determinado por el nivel de visibilidad (`public`, `private`, etc.).
- **Tipo**: Puede ser primitivo o referenciado.
- **Valor inicial**: Puede asignarse al declararlo o mediante el constructor.

Ejemplo práctico de atributos

```
public class Coche {
    // Atributos
    private String marca;
    private int anio;
    // Constructor
    public Coche(String marca, int anio) {
        this.marca = marca;
        this.anio = anio;
    }
    // Método para mostrar la información
    public void mostrarCoche() {
        System.out.println("Marca: " + marca + ", Año: " + anio);
    }
}
```

Uso:

```
public class Main {
    public static void main(String[] args) {
        Coche coche = new Coche("Toyota", 2022);
        coche.mostrarCoche();
    }
}
```

Salida:

```
Marca: Toyota, Año: 2022
```

3. Métodos

Los **métodos** son funciones que definen el comportamiento de los objetos. Pueden realizar tareas específicas, modificar atributos o devolver información.

Tipos de métodos:

1. **Métodos de instancia: Se ejecutan sobre un objeto en particular.**
2. **Métodos estáticos: Pertenecen a la clase y no requieren un objeto para ejecutarse.**
3. **Constructores: Métodos especiales que inicializan los objetos.**

Ejemplo práctico de métodos

```
public class Calculadora {
    // Métodos de instancia
    public int sumar(int a, int b) {
        return a + b;
    }
    public int restar(int a, int b) {
        return a - b;
    }
    // Método estático
    public static void mostrarBienvenida() {
        System.out.println("Bienvenido a la calculadora.");
    }
}
```

Uso:

```
public class Main {
    public static void main(String[] args) {
        // Llamar a un método estático
        Calculadora.mostrarBienvenida();
        // Crear un objeto y usar métodos de instancia
        Calculadora calc = new Calculadora();
        int suma = calc.sumar(10, 5);
        int resta = calc.restar(10, 5);
        System.out.println("Suma: " + suma);
        System.out.println("Resta: " + resta);
    }
}
```

Salida:

```
Bienvenido a la calculadora.
Suma: 15
Resta: 5
```

4. Visibilidad

La **visibilidad** define quién puede acceder a los atributos y métodos de una clase. Esto se controla mediante modificadores como `public`, `private`, `protected` y el modificador por defecto (**default**).

Modificadores de visibilidad en Java:

1. `public`: **Accesible desde cualquier clase.**
2. `private`: **Accesible solo dentro de la clase en la que se define.**
3. `protected`: **Accesible desde clases del mismo paquete y subclases.**
4. **Paquete (`default`): Accesible solo dentro del mismo paquete.**

Ejemplo de control de acceso

```java
public class CuentaBancaria {
    // Atributo privado
    private double saldo;
    // Constructor
    public CuentaBancaria(double saldoInicial) {
        this.saldo = saldoInicial;
    }
    // Método público para consultar el saldo
    public double getSaldo() {
        return saldo;
    }
    // Método público para depositar dinero
    public void depositar(double monto) {
        if (monto > 0) {
            saldo += monto;
            System.out.println("Depósito exitoso. Nuevo saldo: " + saldo);
        } else {
            System.out.println("El monto debe ser positivo.");
        }
    }
}
```

Uso:

```
public class Main {
    public static void main(String[] args) {
          CuentaBancaria cuenta = new CuentaBancaria(100.0);
          // Consultar saldo
          System.out.println("Saldo inicial: " + cuenta.getSaldo());
          // Depositar dinero
          cuenta.depositar(50.0);
    }
}
```

Salida:

```
Saldo inicial: 100.0
Depósito exitoso. Nuevo saldo: 150.0
```

Ejercicios prácticos

1. Crea una clase "Libro":

- Atributos: `titulo`, `autor`, `precio`.
- Métodos:
  - `mostrarInformacion()`: Imprime los atributos en la consola.

2. Crea una clase "Rectangulo":

- Atributos: `base` y `altura`.
- Métodos:
  - `calcularArea()`: Devuelve el área del rectángulo.
  - `calcularPerimetro()`: Devuelve el perímetro del rectángulo.

3. Implementa un sistema de control de acceso:

- Clase "Usuario" con atributos privados `nombre` y `contrasena`.
- Métodos:
  - `setContrasena(String nuevaContrasena)`: Cambia la contraseña solo si cumple con requisitos (ej., más de 6 caracteres).
  - `verificarContrasena(String contrasena)`: Verifica si la contraseña es correcta.

5. Profundizando en los Constructores

Un **constructor** es un método especial que se ejecuta automáticamente cuando se crea un objeto. Su propósito principal es inicializar los atributos de la clase. En Java, un constructor:

- Tiene el mismo nombre que la clase.
- No tiene un tipo de retorno.
- Puede sobrecargarse para aceptar diferentes conjuntos de parámetros.

Ejemplo básico de constructor

```
public class Persona {
    private String nombre;
    private int edad;
    // Constructor
    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
    public void mostrarInformacion() {
        System.out.println("Nombre: " + nombre + ", Edad: " + edad);
    }
}
```

Uso:

```
public class Main {
    public static void main(String[] args) {
        // Crear un objeto utilizando el constructor
        Persona persona1 = new Persona("Ana", 25);
        persona1.mostrarInformacion();
    }
}
```

Salida:

```
Nombre: Ana, Edad: 25
```

Sobrecarga de constructores

Puedes definir múltiples constructores con diferentes parámetros para inicializar objetos de varias formas.

Ejemplo:

```
public class Coche {
    private String marca;
    private int anio;
    // Constructor 1: Inicializa todos los atributos
    public Coche(String marca, int anio) {
          this.marca = marca;
          this.anio = anio;
    }
    // Constructor 2: Inicializa solo la marca, el año es predeterminado
    public Coche(String marca) {
          this.marca = marca;
          this.anio = 2000; // Año predeterminado
    }
    public void mostrarCoche() {
          System.out.println("Marca: " + marca + ", Año: " + anio);
    }
}
```

Uso:

```
public class Main {
    public static void main(String[] args) {
          Coche coche1 = new Coche("Toyota", 2022);
          Coche coche2 = new Coche("Honda");
          coche1.mostrarCoche();
          coche2.mostrarCoche();
    }
}
```

Salida:

```
Marca: Toyota, Año: 2022
Marca: Honda, Año: 2000
```

6. Métodos Estáticos vs Métodos de Instancia

Los **métodos de instancia** se ejecutan sobre objetos específicos, mientras que los **métodos estáticos** pertenecen a la clase y no requieren un objeto.

Métodos de instancia

- Se utilizan cuando el método depende de los atributos del objeto.

Ejemplo:

```
public class Rectangulo {
    private int base;
    private int altura;
    public Rectangulo(int base, int altura) {
          this.base = base;
          this.altura = altura;
    }
    public int calcularArea() {
          return base * altura;
    }
}
```

Métodos estáticos

- Se utilizan para operaciones que no dependen de atributos específicos de objetos.

Ejemplo:

```
public class Calculadora {
    public static int sumar(int a, int b) {
          return a + b;
    }
    public static int multiplicar(int a, int b) {
          return a * b;
    }
}
```

Uso de ambos tipos de métodos:

```
  public class Main {
      public static void main(String[] args) {
            // Uso de método de instancia
            Rectangulo rectangulo = new Rectangulo(4, 5);
            System.out.println("Área del rectángulo: " + rectangulo.calcularAr-
ea());
            // Uso de método estático
            int suma = Calculadora.sumar(3, 7);
            System.out.println("Suma: " + suma);
      }
  }
```

Salida:

```
  Área del rectángulo: 20
  Suma: 10
```

7. Getters y Setters (Control de Acceso)

Los **getters** y **setters** son métodos especiales que permiten acceder y modificar los atributos de una clase mientras mantienen la encapsulación. Esto protege los datos internos del objeto y asegura que los cambios se realicen de forma controlada.

Definición de Getters y Setters

```
public class Producto {
    private String nombre;
    private double precio;
    // Constructor
    public Producto(String nombre, double precio) {
        this.nombre = nombre;
        this.precio = precio;
    }
    // Getter para nombre
    public String getNombre() {
        return nombre;
    }
    // Setter para nombre
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    // Getter para precio
    public double getPrecio() {
        return precio;
    }
    // Setter para precio
    public void setPrecio(double precio) {
        if (precio > 0) {
            this.precio = precio;
        } else {
            System.out.println("El precio debe ser positivo.");
        }
    }
}
```

Uso:

```
public class Main {
    public static void main(String[] args) {
        Producto producto = new Producto("Laptop", 1000.0);
        // Consultar atributos con getters
        System.out.println("Producto: " + producto.getNombre() + ", Precio:
" + producto.getPrecio());
        // Modificar atributos con setters
        producto.setPrecio(1200.0);
        System.out.println("Nuevo precio: " + producto.getPrecio());
        // Intentar un cambio inválido
        producto.setPrecio(-500.0);
    }
}
```

Salida:

```
Producto: Laptop, Precio: 1000.0
Nuevo precio: 1200.0
El precio debe ser positivo.
```

8. Ejercicios Prácticos para Consolidar

1. Crear una clase `CuentaBancaria`:

- Atributos: `saldo` (privado).
- Métodos:
  - `depositar(double monto)`: Aumenta el saldo.
  - `retirar(double monto)`: Disminuye el saldo, siempre que el monto sea menor o igual al saldo actual.
  - `consultarSaldo()`: Devuelve el saldo actual.

2. Diseñar una clase `Empleado`:

- Atributos: `nombre`, `salario`.
- Métodos:
  - `aumentarSalario(double porcentaje)`: Incrementa el salario en un porcentaje dado.
  - `mostrarInformacion()`: Imprime el nombre y salario.

3. Crear una clase `Libro`:

- Atributos: `titulo`, `autor`, `precio`.
- Métodos:
  - Constructor para inicializar los atributos.
  - Getters y setters para modificar el precio.
  - Método `mostrarDetalles()` que imprima toda la información del libro.

4. Desarrollar una clase `Vehiculo`:

- Atributos: `marca`, `modelo`, `kilometraje`.
- Métodos:
  - `conducir(int km)`: Incrementa el kilometraje en la cantidad indicada.
  - `mostrarKilometraje()`: Imprime el kilometraje actual.

5. Implementar una clase `Alumno`:

- Atributos: `nombre`, `calificacion`.
- Métodos:
  - Constructor para inicializar el nombre y la calificación.
  - Método `aprobado()`: Devuelve `true` si la calificación es mayor o igual a 5.
  - Método `mostrarEstado()`: Imprime si el alumno está aprobado o no.

Ejercicios propuestos

Ejercicio 1: Crear una Clase "Producto" con Métodos Básicos

Objetivo:

Crear una clase que modele un producto con nombre y precio, y permita al usuario consultar y modificar el precio de manera controlada.

Pasos para el alumno:

1. Define la clase `Producto`:

- Crea dos atributos privados: `nombre` (String) y `precio` (double).

2. Agrega un constructor:

- Inicializa ambos atributos al crear un objeto.

3. Implementa getters y setters:

- Crea métodos `getNombre`, `getPrecio`, `setPrecio`.
- Asegúrate de validar que el precio no sea negativo.

4. Crea un método adicional:

- Método `mostrarInformacion` que imprima el nombre y precio del producto.

5. Escribe una clase principal `Main`:

- Crea un objeto de tipo `Producto`.
- Usa los métodos para consultar y modificar el precio.
- Muestra la información del producto en cada paso.

Código Completo en la siguiente página

```java
// Clase Producto
public class Producto {
    private String nombre;
    private double precio;
    // Constructor
    public Producto(String nombre, double precio) {
        this.nombre = nombre;
        this.precio = precio;
    }
    // Getters
    public String getNombre() {
        return nombre;
    }
    public double getPrecio() {
        return precio;
    }
    // Setters
    public void setPrecio(double precio) {
        if (precio >= 0) {
            this.precio = precio;
        } else {
            System.out.println("El precio no puede ser negativo.");
        }
    }
    // Método adicional
    public void mostrarInformacion() {
        System.out.println("Producto: " + nombre + ", Precio: $" + precio);
    }
}
// Clase Main para probar Producto
public class Main {
    public static void main(String[] args) {
        // Crear un objeto Producto
        Producto producto = new Producto("Laptop", 1200.0);
        // Mostrar información inicial
        producto.mostrarInformacion();
        // Cambiar el precio
        producto.setPrecio(1350.0);
        producto.mostrarInformacion();
        // Intentar establecer un precio negativo
        producto.setPrecio(-100.0);
        producto.mostrarInformacion();
    }
}
```

Salida Esperada:

```
Producto: Laptop, Precio: $1200.0
Producto: Laptop, Precio: $1350.0
El precio no puede ser negativo.
Producto: Laptop, Precio: $1350.0
```

Ejercicio 2: Sistema de Gestión de Empleados

Objetivo:

Crear un sistema básico que permita gestionar información de empleados, incluyendo nombre, salario y aumentos salariales.

Pasos para el alumno:

1. Define la clase `Empleado`:

- Crea dos atributos privados: `nombre` (String) y `salario` (double).

2. Crea el constructor:

- Inicializa los atributos al crear un objeto.

3. Implementa los métodos:

- `getNombre` y `getSalario`: Devuelven el nombre y el salario.
- `aumentarSalario`: Aumenta el salario en un porcentaje dado.
- `mostrarInformacion`: Imprime el nombre y salario del empleado.

4. Escribe la clase principal `Main`:

- Crea varios empleados.
- Usa los métodos para aumentar sus salarios y mostrar la información actualizada.

Código Completo en la siguiente página

```
// Clase Empleado
public class Empleado {
    private String nombre;
    private double salario;
    // Constructor
    public Empleado(String nombre, double salario) {
        this.nombre = nombre;
        this.salario = salario;
    }
    // Getters
    public String getNombre() {
        return nombre;
    }
    public double getSalario() {
        return salario;
    }
    // Método para aumentar el salario
    public void aumentarSalario(double porcentaje) {
        if (porcentaje > 0) {
            salario += salario * porcentaje / 100;
        } else {
            System.out.println("El porcentaje debe ser positivo.");
        }
    }
    // Método para mostrar información
    public void mostrarInformacion() {
        System.out.println("Empleado: " + nombre + ", Salario: $" + sala-
rio);
    }
}
// Clase Main para probar Empleado
public class Main {
    public static void main(String[] args) {
        // Crear objetos Empleado
        Empleado empleado1 = new Empleado("Ana", 3000.0);
        Empleado empleado2 = new Empleado("Carlos", 4000.0);
        // Mostrar información inicial
        empleado1.mostrarInformacion();
        empleado2.mostrarInformacion();
        // Aumentar salarios
        empleado1.aumentarSalario(10); // 10%
        empleado2.aumentarSalario(15); // 15%
        // Mostrar información actualizada
```

```
            empleado1.mostrarInformacion();
            empleado2.mostrarInformacion();
        }
    }
```

Salida Esperada:

```
Empleado: Ana, Salario: $3000.0
Empleado: Carlos, Salario: $4000.0
Empleado: Ana, Salario: $3300.0
Empleado: Carlos, Salario: $4600.0
```

Ejercicio 3: Sistema de Reservas de Hotel

Objetivo:

Diseñar un sistema que permita crear habitaciones de hotel con precios y reservarlas.

Pasos para el alumno:

1. Define la clase `Habitacion`:

- Atributos: `numeroHabitacion`, `precio`, `reservada` (boolean).

2. Crea el constructor:

- Inicializa los atributos.

3. Implementa los métodos:

- `reservarHabitacion`: Cambia el estado de `reservada` a `true` si no está ya reservada.
- `cancelarReserva`: Cambia el estado de `reservada` a `false` si está reservada.
- `mostrarInformacion`: Muestra el número de habitación, precio y estado.

4. Escribe la clase principal `Main`:

- Crea varias habitaciones.
- Reserva y cancela reservas, mostrando el estado actualizado.

Código Completo en la siguiente página

```
// Clase Habitacion
public class Habitacion {
    private int numeroHabitacion;
    private double precio;
    private boolean reservada;
    // Constructor
    public Habitacion(int numeroHabitacion, double precio) {
        this.numeroHabitacion = numeroHabitacion;
        this.precio = precio;
        this.reservada = false;
    }
    // Método para reservar
    public void reservarHabitacion() {
        if (!reservada) {
            reservada = true;
            System.out.println("Habitación " + numeroHabitacion + " reservada con éxito.");
        } else {
            System.out.println("La habitación " + numeroHabitacion + " ya está reservada.");
        }
    }
    // Método para cancelar reserva
    public void cancelarReserva() {
        if (reservada) {
            reservada = false;
            System.out.println("Reserva de la habitación " + numeroHabitacion + " cancelada.");
        } else {
            System.out.println("La habitación " + numeroHabitacion + " no está reservada.");
        }
    }
    // Método para mostrar información
    public void mostrarInformacion() {
        String estado = reservada ? "Reservada" : "Disponible";
        System.out.println("Habitación " + numeroHabitacion + ", Precio: $" + precio + ", Estado: " + estado);
    }
}
// Clase Main para probar Habitacion
public class Main {
    public static void main(String[] args) {
        // Crear habitaciones
```

```
        Habitacion hab1 = new Habitacion(101, 150.0);
        Habitacion hab2 = new Habitacion(102, 200.0);
        // Mostrar información inicial
        hab1.mostrarInformacion();
        hab2.mostrarInformacion();
        // Reservar y cancelar habitaciones
        hab1.reservarHabitacion();
        hab1.reservarHabitacion(); // Intento de reserva doble
        hab1.cancelarReserva();
        hab1.cancelarReserva(); // Intento de cancelación doble
        // Mostrar información final
        hab1.mostrarInformacion();
        hab2.mostrarInformacion();
    }
}
```

Salida Esperada:

```
Habitación 101, Precio: $150.0, Estado: Disponible
Habitación 102, Precio: $200.0, Estado: Disponible
Habitación 101 reservada con éxito.
La habitación 101 ya está reservada.
Reserva de la habitación 101 cancelada.
La habitación 101 no está reservada.
Habitación 101, Precio: $150.0, Estado: Disponible
Habitación 102, Precio: $200.0, Estado: Disponible
```

Ejercicios Propuestos: Scanner

Ejercicio 1: Producto Interactivo

Objetivo:

Modificar la clase `Producto` para permitir al usuario introducir los datos del producto y decidir si cambiar el precio.

```
import java.util.Scanner;
public class Producto {
    private String nombre;
    private double precio;
    // Constructor
    public Producto(String nombre, double precio) {
        this.nombre = nombre;
        this.precio = precio;
    }
    // Getters y Setters
    public String getNombre() {
        return nombre;
    }
    public double getPrecio() {
        return precio;
    }
    public void setPrecio(double precio) {
        if (precio >= 0) {
            this.precio = precio;
        } else {
            System.out.println("El precio no puede ser negativo.");
        }
    }
    // Método para mostrar información
    public void mostrarInformacion() {
        System.out.println("Producto: " + nombre + ", Precio: $" + precio);
    }
    // Main para interacción con el usuario
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        // Pedir información del producto al usuario
        System.out.print("Introduce el nombre del producto: ");
        String nombre = scanner.nextLine();
        System.out.print("Introduce el precio inicial del producto: ");
        double precio = scanner.nextDouble();
        // Crear objeto Producto
        Producto producto = new Producto(nombre, precio);
        // Mostrar información inicial
        producto.mostrarInformacion();
        // Preguntar al usuario si desea cambiar el precio
        System.out.print("¿Deseas cambiar el precio? (sí/no): ");
        scanner.nextLine(); // Consumir salto de línea
        String respuesta = scanner.nextLine();
```

```
            if (respuesta.equalsIgnoreCase("sí")) {
                System.out.print("Introduce el nuevo precio: ");
                double nuevoPrecio = scanner.nextDouble();
                producto.setPrecio(nuevoPrecio);
            }
            // Mostrar información final
            producto.mostrarInformacion();
            scanner.close();
        }
    }
```

Salida Ejemplo 1 (Interacción):

```
Introduce el nombre del producto: Monitor
Introduce el precio inicial del producto: 250.5
Producto: Monitor, Precio: $250.5
¿Deseas cambiar el precio? (sí/no): sí
Introduce el nuevo precio: 300.75
Producto: Monitor, Precio: $300.75
```

Ejercicio 2: Sistema de Gestión de Empleados

Objetivo:

Modificar la clase `Empleado` para que los datos de los empleados se introduzcan desde la consola y permitir ajustes dinámicos del salario.

Código Mejorado en la siguiente página

```
import java.util.Scanner;
public class Empleado {
    private String nombre;
    private double salario;
    // Constructor
    public Empleado(String nombre, double salario) {
        this.nombre = nombre;
        this.salario = salario;
    }
    // Métodos
    public String getNombre() {
        return nombre;
    }
    public double getSalario() {
        return salario;
    }
    public void aumentarSalario(double porcentaje) {
        if (porcentaje > 0) {
            salario += salario * porcentaje / 100;
        } else {
            System.out.println("El porcentaje debe ser positivo.");
        }
    }
    public void mostrarInformacion() {
        System.out.println("Empleado: " + nombre + ", Salario: $" + sala-
rio);
    }
    // Main con interacción del usuario
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        // Introducir datos del empleado
        System.out.print("Introduce el nombre del empleado: ");
        String nombre = scanner.nextLine();
        System.out.print("Introduce el salario inicial del empleado: ");
        double salario = scanner.nextDouble();
        // Crear objeto Empleado
        Empleado empleado = new Empleado(nombre, salario);
        // Mostrar información inicial
        empleado.mostrarInformacion();
        // Ajustar salario
        System.out.print("Introduce el porcentaje de aumento salarial: ");
        double porcentaje = scanner.nextDouble();
        empleado.aumentarSalario(porcentaje);
```

```
            // Mostrar información final
            empleado.mostrarInformacion();
            scanner.close();
        }
    }
```

Salida Ejemplo 2 (Interacción):

```
Introduce el nombre del empleado: Laura
Introduce el salario inicial del empleado: 2000
Empleado: Laura, Salario: $2000.0
Introduce el porcentaje de aumento salarial: 10
Empleado: Laura, Salario: $2200.0
```

1. Validar entradas del usuario:

- Asegurarse de que los valores introducidos sean válidos (ej., no permitir salarios negativos).
- Usar un bucle para solicitar entradas hasta que sean correctas.

2. Añadir menús interactivos:

- Ofrecer al usuario opciones para realizar diferentes operaciones, como modificar atributos, consultar información, etc.

3. Extender a sistemas más complejos:

- Crear una lista de empleados o productos que permita manejar múltiples objetos en una sola ejecución.

Parte 2: Objetos. Estado, comportamiento e identidad. Mensajes.

1. ¿Qué es un Objeto?

Un objeto es una **entidad concreta** que combina datos y comportamientos. Es una representación de algo del mundo real o abstracto, modelado mediante una clase. En Java, un objeto es una instancia de una clase.

- **Ejemplo del mundo real**:

Un coche tiene:

- **Estado**: Su marca, modelo, color, nivel de combustible.
- **Comportamiento**: Puede acelerar, frenar, girar.
- **Identidad**: Cada coche tiene un número de serie único.

1.1. Estado

El **estado** de un objeto se define mediante sus **atributos**. Es el conjunto de valores que describen al objeto en un momento dado. El estado puede cambiar durante la ejecución del programa mediante la interacción con sus métodos.

Ejemplo: Estado Dinámico

Clase que representa una bombilla con atributos que describen si está encendida o apagada.

```
public class Bombilla {
    private boolean encendida; // Estado
    // Constructor: Inicialmente apagada
    public Bombilla() {
          this.encendida = false;
    }
    // Métodos para cambiar el estado
    public void encender() {
          encendida = true;
          System.out.println("La bombilla está encendida.");
    }
    public void apagar() {
          encendida = false;
          System.out.println("La bombilla está apagada.");
    }
    // Método para mostrar el estado actual
    public void mostrarEstado() {
          String estado = encendida ? "encendida" : "apagada";
          System.out.println("La bombilla está " + estado + ".");
    }
}
```

Uso de la clase:

```
public class Main {
    public static void main(String[] args) {
          Bombilla bombilla = new Bombilla();
          // Mostrar estado inicial
          bombilla.mostrarEstado();
          // Cambiar estado
          bombilla.encender();
          bombilla.mostrarEstado();
          bombilla.apagar();
          bombilla.mostrarEstado();
    }
}
```

Salida:

```
La bombilla está apagada.
La bombilla está encendida.
La bombilla está apagada.
```

Estado y Métodos Mutadores

Los métodos mutadores (como `encender` o `apagar`) son responsables de cambiar el estado del objeto. Es una **buena práctica** usar métodos en lugar de acceder directamente a los atributos.

Ejemplo Extendido: Ventilador con Velocidades

```
  public class Ventilador {
      private boolean encendido;
      private int velocidad; // 0: Apagado, 1: Baja, 2: Media, 3: Alta
      // Constructor
      public Ventilador() {
            this.encendido = false;
            this.velocidad = 0;
      }
      // Métodos para cambiar el estado
      public void encender() {
            if (!encendido) {
                  encendido = true;
                  velocidad = 1; // Comienza en baja
                  System.out.println("El ventilador está encendido en velocidad
baja.");
            } else {
                  System.out.println("El ventilador ya está encendido.");
            }
      }
      public void apagar() {
            encendido = false;
            velocidad = 0;
            System.out.println("El ventilador está apagado.");
      }
      public void cambiarVelocidad(int nuevaVelocidad) {
            if (encendido && nuevaVelocidad >= 0 && nuevaVelocidad <= 3) {
                  velocidad = nuevaVelocidad;
                  System.out.println("Velocidad del ventilador cambiada a " +
velocidad);
            } else if (!encendido) {
                  System.out.println("No puedes cambiar la velocidad con el ven-
tilador apagado.");
            } else {
                  System.out.println("Velocidad no válida.");
            }
      }
      // Mostrar estado actual
      public void mostrarEstado() {
            String estado = encendido ? "encendido" : "apagado";
            System.out.println("Ventilador " + estado + ", velocidad: " + ve-
locidad);
      }
  }
```

Uso de la clase:

```
public class Main {
    public static void main(String[] args) {
          Ventilador ventilador = new Ventilador();
          // Cambiar y mostrar estado
          ventilador.mostrarEstado();
          ventilador.encender();
          ventilador.cambiarVelocidad(2);
          ventilador.mostrarEstado();
          ventilador.apagar();
          ventilador.mostrarEstado();
    }
}
```

1.2. Comportamiento

El **comportamiento** de un objeto define lo que puede hacer o cómo puede interactuar con otros objetos. En Java, el comportamiento se implementa mediante **métodos**.

Ejemplo: Caja Fuerte

```
public class CajaFuerte {
    private String clave;
    private boolean abierta;
    // Constructor
    public CajaFuerte(String clave) {
        this.clave = clave;
        this.abierta = false;
    }
    // Método para abrir
    public void abrir(String intentoClave) {
        if (clave.equals(intentoClave)) {
            abierta = true;
            System.out.println("Caja fuerte abierta.");
        } else {
            System.out.println("Clave incorrecta.");
        }
    }
    // Método para cerrar
    public void cerrar() {
        if (abierta) {
            abierta = false;
            System.out.println("Caja fuerte cerrada.");
        } else {
            System.out.println("La caja fuerte ya está cerrada.");
        }
    }
    // Mostrar estado
    public void mostrarEstado() {
        String estado = abierta ? "abierta" : "cerrada";
        System.out.println("La caja fuerte está " + estado + ".");
    }
}
```

Uso:

```
public class Main {
    public static void main(String[] args) {
          CajaFuerte caja = new CajaFuerte("1234");
          // Mostrar estado inicial
          caja.mostrarEstado();
          // Intentar abrir con clave incorrecta
          caja.abrir("1111");
          caja.mostrarEstado();
          // Abrir con clave correcta
          caja.abrir("1234");
          caja.mostrarEstado();
          // Cerrar
          caja.cerrar();
          caja.mostrarEstado();
    }
}
```

Salida:

```
La caja fuerte está cerrada.
Clave incorrecta.
La caja fuerte está cerrada.
Caja fuerte abierta.
La caja fuerte está abierta.
Caja fuerte cerrada.
La caja fuerte está cerrada.
```

1.3. Identidad

Dos objetos pueden tener el mismo estado y comportamiento, pero son **diferentes** debido a su identidad. En Java, la identidad está asociada con la dirección de memoria.

Ejemplo: Objetos con el mismo estado pero diferente identidad

```
public class Persona {
    private String nombre;
    private int edad;
    public Persona(String nombre, int edad) {
          this.nombre = nombre;
          this.edad = edad;
    }
    public void mostrarInformacion() {
          System.out.println("Persona: " + nombre + ", Edad: " + edad);
    }
}
public class Main {
    public static void main(String[] args) {
          Persona persona1 = new Persona("Carlos", 30);
          Persona persona2 = new Persona("Carlos", 30);
          // Mostrar información
          persona1.mostrarInformacion();
          persona2.mostrarInformacion();
          // Comparar identidad
          if (persona1 == persona2) {
                System.out.println("Ambos son el mismo objeto.");
          } else {
                System.out.println("Son objetos diferentes.");
          }
    }
}
```

Salida:

```
Persona: Carlos, Edad: 30
Persona: Carlos, Edad: 30
Son objetos diferentes.
```

Explicación

- Aunque `persona1` y `persona2` tienen el mismo estado (`nombre` y `edad`), son **objetos diferentes** porque ocupan direcciones distintas en memoria.

1.4. Mensajes

Un mensaje en Java es la forma en que un objeto **invoca métodos** de otro. Este mecanismo permite que los objetos interactúen entre sí.

Ejemplo: Envío de Mensajes

Clase `Persona` que envía saludos a otras personas.

```
public class Persona {
    private String nombre;
    public Persona(String nombre) {
        this.nombre = nombre;
    }
    public void enviarSaludo(Persona otraPersona) {
        System.out.println(nombre + " saluda a " + otraPersona.nombre);
    }
}
```

Uso:

```
public class Main {
    public static void main(String[] args) {
        Persona persona1 = new Persona("Ana");
        Persona persona2 = new Persona("Luis");
        // Comunicación entre objetos
        persona1.enviarSaludo(persona2);
        persona2.enviarSaludo(persona1);
    }
}
```

Salida:

```
Ana saluda a Luis
Luis saluda a Ana
```

2. Test de comprensión

1. ¿Qué define el estado de un objeto en Java?

a) Los métodos de la clase.

b) Los atributos del objeto.

c) La relación entre clases.

d) La dirección de memoria.

Respuesta correcta: **b**

2. ¿Qué caracteriza el comportamiento de un objeto?

a) Los atributos que lo describen.
b) La cantidad de memoria que ocupa.
c) Los métodos que definen sus acciones.
d) Su identidad única.

Respuesta correcta: **c**

3. ¿Qué determina la identidad de un objeto?

a) Su estado y comportamiento.
b) Su posición en el código fuente.
c) Su dirección única en memoria.
d) Los valores de sus atributos.

Respuesta correcta: **c**

4. Si dos objetos tienen el mismo estado y comportamiento, pero ocupan direcciones de memoria diferentes, entonces:

a) Son objetos diferentes.
b) Son el mismo objeto.
c) No pueden tener el mismo estado.
d) Depende del compilador de Java.

Respuesta correcta: **a**

5. ¿Qué son los mensajes en la programación orientada a objetos?

a) Variables que almacenan información.
b) Llamadas a métodos de un objeto.
c) El flujo de control del programa.
d) La identidad de un objeto.

Respuesta correcta: **b**

6. ¿Qué método de los siguientes cambiaría el estado de un objeto?

a) Un método que imprime información en la consola.
b) Un método que incrementa un atributo privado.
c) Un método que verifica la igualdad de dos objetos.
d) Ninguna de las anteriores.

Respuesta correcta: **b**

Ejercicio Guiado: Sistema de Gestión de Tareas

Objetivo

Crear un programa que modele un sistema básico de gestión de tareas, permitiendo que los usuarios:

1. **Creen tareas.**
2. **Cambien su estado (completada o no completada).**
3. **Muestren todas las tareas con su estado actual.**

Pasos

1. **Define la clase `Tarea`:**
   - Atributos: `descripcion` (String) y `completada` (boolean).
   - Métodos:
     - Constructor para inicializar la tarea como no completada.
     - `marcarCompletada`: Cambia el estado de la tarea a completada.
     - `mostrarEstado`: Imprime la descripción y si está completada.
2. **Crea una clase principal `Main`:**
   - Usa un `ArrayList<Tarea>` para manejar una lista de tareas.
   - Implementa un menú para añadir, completar y mostrar tareas.

Código Completo

```
import java.util.ArrayList;
import java.util.Scanner;
// Clase Tarea
class Tarea {
    private String descripcion;
    private boolean completada;
    // Constructor
    public Tarea(String descripcion) {
        this.descripcion = descripcion;
        this.completada = false; // Inicialmente, no está completada
    }
    // Método para marcar como completada
    public void marcarCompletada() {
        if (!completada) {
            completada = true;
            System.out.println("La tarea \"" + descripcion + "\" ha sido
marcada como completada.");
        } else {
            System.out.println("La tarea \"" + descripcion + "\" ya estaba
completada.");
        }
    }
    // Método para mostrar el estado actual
    public void mostrarEstado() {
        String estado = completada ? "Completada" : "No completada";
        System.out.println("Tarea: \"" + descripcion + "\", Estado: " + es-
tado);
    }
}
// Clase principal
public class Main {
    public static void main(String[] args) {
        ArrayList<Tarea> listaTareas = new ArrayList<>();
        Scanner scanner = new Scanner(System.in);
        boolean continuar = true;
        // Menú interactivo
        while (continuar) {
            System.out.println("\n--- Menú de Gestión de Tareas ---");
            System.out.println("1. Añadir una tarea");
            System.out.println("2. Marcar una tarea como completada");
            System.out.println("3. Mostrar todas las tareas");
```

```
                System.out.println("4. Salir");
                System.out.print("Selecciona una opción: ");
                int opcion = scanner.nextInt();
                scanner.nextLine(); // Consumir salto de línea
                switch (opcion) {
                    case 1:
                        System.out.print("Introduce la descripción de la
nueva tarea: ");
                        String descripcion = scanner.nextLine();
                        listaTareas.add(new Tarea(descripcion));
                        System.out.println("Tarea añadida.");
                        break;
                    case 2:
                        System.out.println("Selecciona la tarea que quieres
marcar como completada:");
                        for (int i = 0; i < listaTareas.size(); i++) {
                            System.out.print((i + 1) + ". ");
                            listaTareas.get(i).mostrarEstado();
                        }
                        int indice = scanner.nextInt() - 1;
                        if (indice >= 0 && indice < listaTareas.size()) {
                            listaTareas.get(indice).marcarCompletada();
                        } else {
                            System.out.println("Índice no válido.");
                        }
                        break;
                    case 3:
                        System.out.println("\n--- Lista de Tareas ---");
                        for (Tarea tarea : listaTareas) {
                            tarea.mostrarEstado();
                        }
                        break;
                    case 4:
                        continuar = false;
                        System.out.println("Saliendo del sistema...");
                        break;
                    default:
                        System.out.println("Opción no válida. Intenta de
nuevo.");
                }
            }
            scanner.close();
        }
    }
```

Salida Esperada (Ejemplo de Interacción)

```
--- Menú de Gestión de Tareas ---
1. Añadir una tarea
2. Marcar una tarea como completada
3. Mostrar todas las tareas
4. Salir
Selecciona una opción: 1
Introduce la descripción de la nueva tarea: Estudiar Java
Tarea añadida.
--- Menú de Gestión de Tareas ---
1. Añadir una tarea
2. Marcar una tarea como completada
3. Mostrar todas las tareas
4. Salir
Selecciona una opción: 3
--- Lista de Tareas ---
Tarea: “Estudiar Java”, Estado: No completada
--- Menú de Gestión de Tareas ---
1. Añadir una tarea
2. Marcar una tarea como completada
3. Mostrar todas las tareas
4. Salir
Selecciona una opción: 2
Selecciona la tarea que quieres marcar como completada:
1. Tarea: “Estudiar Java”, Estado: No completada
1
La tarea “Estudiar Java” ha sido marcada como completada.
--- Menú de Gestión de Tareas ---
1. Añadir una tarea
2. Marcar una tarea como completada
3. Mostrar todas las tareas
4. Salir
Selecciona una opción: 4
Saliendo del sistema...
```

Ampliaciones para el Alumno

1. Añadir una opción para eliminar tareas del sistema.

2. Establecer una prioridad (baja, media, alta) para las tareas.

3. Implementar un sistema de ordenamiento donde las tareas completadas se muestren al final.

Parte 3: Encapsulado. Visibilidad.

El **encapsulamiento** es uno de los principios fundamentales de la programación orientada a objetos (POO). Se refiere a **restringir el acceso directo a los atributos y métodos de una clase** y permitir su interacción únicamente a través de métodos controlados, como getters y setters. Este principio ayuda a proteger los datos y garantizar la integridad del objeto.

1. ¿Qué es el Encapsulado?

El encapsulado permite:

1. **Ocultar detalles internos de la implementación de una clase.**
2. **Proveer una interfaz controlada para acceder y modificar sus datos.**
3. **Proteger la integridad de los atributos, evitando que sean manipulados directamente de manera no controlada.**

Ejemplo de clase no encapsulada (mala práctica):

```
public class Persona {
    String nombre; // Acceso público
    int edad;
    public void mostrarInformacion() {
          System.out.println("Nombre: " + nombre + ", Edad: " + edad);
    }
}
```

En este ejemplo, cualquier clase puede modificar los atributos directamente:

```
Persona persona = new Persona();
persona.nombre = "Carlos";
persona.edad = -5; // Dato inconsistente
```

Problema: No hay control sobre los valores asignados.

2. Encapsulación con Atributos Privados y Métodos Públicos

La solución es **hacer privados** los atributos y proporcionar métodos públicos para interactuar con ellos.

Ejemplo: Clase encapsulada

```
public class Persona {
    private String nombre; // Acceso restringido
    private int edad;
    // Getter para obtener el nombre
    public String getNombre() {
        return nombre;
    }
    // Setter para asignar el nombre
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    // Getter para obtener la edad
    public int getEdad() {
        return edad;
    }
    // Setter para asignar la edad con validación
    public void setEdad(int edad) {
        if (edad >= 0) {
            this.edad = edad;
        } else {
            System.out.println("La edad no puede ser negativa.");
        }
    }
    // Método para mostrar información
    public void mostrarInformacion() {
        System.out.println("Nombre: " + nombre + ", Edad: " + edad);
    }
}
```

Uso en la clase principal:

```
public class Main {
    public static void main(String[] args) {
        Persona persona = new Persona();
        persona.setNombre("Ana");
        persona.setEdad(25);
        persona.mostrarInformacion();
        persona.setEdad(-5); // Intento de asignar un valor inválido
        persona.mostrarInformacion();
    }
}
```

Salida:

```
Nombre: Ana, Edad: 25
La edad no puede ser negativa.
Nombre: Ana, Edad: 25
```

3. Modificadores de Visibilidad

Los **modificadores de visibilidad** controlan el acceso a los atributos y métodos de una clase desde otras clases o paquetes. En Java, existen cuatro niveles de visibilidad:

Modificador	Clase	Paquete	Subclases	Global
`public`	√	√	√	√
`protected`	√	√	√	×
`default`	√	√	×	×
`private`	√	×	×	×

Ejemplo de Modificadores

```
public class Persona {
    public String nombre;       // Acceso global
    protected int edad;           // Acceso dentro del paquete y subclases
    String direccion;                // Acceso dentro del paquete (default)
    private String dni;           // Acceso solo dentro de la clase
    // Métodos para interactuar con los atributos privados
    public String getDni() {
          return dni;
    }
    public void setDni(String dni) {
          this.dni = dni;
    }
}
```

4. Ventajas del Encapsulado

1. Protección de datos:

- Los datos sensibles se ocultan y no pueden ser manipulados directamente.

2. Control de acceso:

- Los métodos permiten validar los datos antes de asignarlos.

3. Mantenimiento y flexibilidad:

- Cambiar la implementación interna no afecta el resto del código si la interfaz pública se mantiene.

5. Ejemplo Completo

Crea una clase `CuentaBancaria` que implemente el concepto de encapsulamiento.

Requisitos:

1. Los atributos son privados:

- `numeroCuenta` (tipo `String`).
- `saldo` (tipo `double`).

2. Métodos:

- `depositar(double cantidad)`: Incrementa el saldo.
- `retirar(double cantidad)`: Decrementa el saldo solo si hay fondos suficientes.
- `mostrarInformacion()`: Muestra el número de cuenta y el saldo actual.

Clase `CuentaBancaria`:

```
  public class CuentaBancaria {
      private String numeroCuenta;
      private double saldo;
      // Constructor
      public CuentaBancaria(String numeroCuenta, double saldoInicial) {
            this.numeroCuenta = numeroCuenta;
            this.saldo = saldoInicial;
      }
      // Método para depositar dinero
      public void depositar(double cantidad) {
            if (cantidad > 0) {
                  saldo += cantidad;
                  System.out.println("Depósito realizado. Nuevo saldo: $" + sal-
do);
            } else {
                  System.out.println("La cantidad a depositar debe ser positi-
va.");
            }
      }
      // Método para retirar dinero
      public void retirar(double cantidad) {
            if (cantidad > 0 && cantidad <= saldo) {
                  saldo -= cantidad;
                  System.out.println("Retiro realizado. Nuevo saldo: $" + sal-
do);
            } else if (cantidad > saldo) {
                  System.out.println("Fondos insuficientes.");
            } else {
                  System.out.println("La cantidad a retirar debe ser positi-
va.");
            }
      }
      // Método para mostrar información
      public void mostrarInformacion() {
            System.out.println("Número de cuenta: " + numeroCuenta);
            System.out.println("Saldo actual: $" + saldo);
      }
  }
```

Clase principal:

```
public class Main {
    public static void main(String[] args) {
          CuentaBancaria cuenta = new CuentaBancaria("123456789", 1000);
          cuenta.mostrarInformacion();
          cuenta.depositar(500);
          cuenta.retirar(300);
          cuenta.retirar(1500); // Intento de retiro sin fondos
          cuenta.mostrarInformacion();
    }
}
```

Salida:

```
Número de cuenta: 123456789
Saldo actual: $1000.0
Depósito realizado. Nuevo saldo: $1500.0
Retiro realizado. Nuevo saldo: $1200.0
Fondos insuficientes.
Número de cuenta: 123456789
Saldo actual: $1200.0
```

6. Ejercicio

1. Crea una clase `Producto` con los atributos privados:

- `nombre` (tipo `String`).
- `precio` (tipo `double`).
- `cantidadEnStock` (tipo `int`).

2. Implementa métodos públicos:

- **Getters y setters** para acceder y modificar los atributos con validaciones.
- `calcularValorInventario()`: Devuelve el valor total del inventario (`precio * cantidadEnStock`).
- `mostrarInformacion()`: Muestra todos los datos del producto.

3. En la clase principal:

- Crea un producto.
- Modifica sus atributos utilizando los setters.
- Calcula y muestra el valor del inventario.

Parte 4: Relaciones entre clases

En Java, las **relaciones entre clases** permiten modelar cómo interactúan las entidades dentro de un programa. Estas relaciones son fundamentales en la programación orientada a objetos (POO) y se utilizan para construir sistemas complejos de manera estructurada.

1. Tipos de Relaciones entre Clases

1. Asociación: Representa una relación lógica entre dos clases.

- Puede ser **unidireccional** o **bidireccional**.
- Ejemplo: Un cliente tiene una cuenta bancaria.

2. Agregación: Una relación más fuerte donde una clase contiene a otra como parte de su estructura, pero los objetos tienen ciclos de vida independientes.

- Ejemplo: Un equipo contiene jugadores.

3. Composición: Una relación de dependencia más fuerte donde una clase contiene otra, y ambas tienen el mismo ciclo de vida.

- Ejemplo: Un coche tiene un motor.

4. Herencia: Una clase (subclase) hereda atributos y métodos de otra clase (superclase).

- Ejemplo: Un gato es un animal.

5. Dependencia: Una clase utiliza otra para realizar una acción específica.

- Ejemplo: Un pedido usa un producto para calcular el precio total.

2. Asociación

Unidireccional

Una clase conoce a la otra, pero no al revés.

Ejemplo: Cliente y Cuenta Bancaria

```
public class Cliente {
    private String nombre;
    private CuentaBancaria cuenta; // Relación unidireccional
    public Cliente(String nombre, CuentaBancaria cuenta) {
        this.nombre = nombre;
        this.cuenta = cuenta;
    }
    public void mostrarInformacion() {
        System.out.println("Cliente: " + nombre);
        cuenta.mostrarInformacion();
    }
}
public class CuentaBancaria {
    private String numeroCuenta;
    private double saldo;
    public CuentaBancaria(String numeroCuenta, double saldoInicial) {
        this.numeroCuenta = numeroCuenta;
        this.saldo = saldoInicial;
    }
    public void mostrarInformacion() {
        System.out.println("Número de cuenta: " + numeroCuenta + ", Saldo:
$" + saldo);
    }
}
public class Main {
    public static void main(String[] args) {
        CuentaBancaria cuenta = new CuentaBancaria("123456789", 1500);
        Cliente cliente = new Cliente("Carlos", cuenta);
        cliente.mostrarInformacion();
    }
}
```

Salida:

```
Cliente: Carlos
Número de cuenta: 123456789, Saldo: $1500.0
```

3. Agregación

La **agregación** implica que una clase contiene objetos de otra, pero ambos tienen ciclos de vida independientes.

Ejemplo: Equipo y Jugadores

```
import java.util.ArrayList;
public class Jugador {
    private String nombre;
    public Jugador(String nombre) {
          this.nombre = nombre;
    }
    public void mostrarInformacion() {
          System.out.println("Jugador: " + nombre);
    }
}
public class Equipo {
    private String nombre;
    private ArrayList<Jugador> jugadores;
    public Equipo(String nombre) {
          this.nombre = nombre;
          this.jugadores = new ArrayList<>();
    }
    public void agregarJugador(Jugador jugador) {
          jugadores.add(jugador);
    }
    public void mostrarInformacion() {
          System.out.println("Equipo: " + nombre);
          System.out.println("Jugadores:");
          for (Jugador jugador : jugadores) {
                jugador.mostrarInformacion();
          }
    }
}
public class Main {
    public static void main(String[] args) {
          Jugador jugador1 = new Jugador("Luis");
          Jugador jugador2 = new Jugador("Ana");
          Equipo equipo = new Equipo("Tigres");
          equipo.agregarJugador(jugador1);
          equipo.agregarJugador(jugador2);
          equipo.mostrarInformacion();
    }
}
```

Salida:

```
Equipo: Tigres
Jugadores:
Jugador: Luis
Jugador: Ana
```

4. Composición

La **composición** es una relación más fuerte. Si la clase contenedora desaparece, los objetos que contiene también lo hacen.

Ejemplo: Coche y Motor

```
public class Motor {
    private int potencia;
    public Motor(int potencia) {
        this.potencia = potencia;
    }
    public void mostrarInformacion() {
        System.out.println("Potencia del motor: " + potencia + " HP");
    }
}
public class Coche {
    private String marca;
    private Motor motor;
    public Coche(String marca, int potenciaMotor) {
        this.marca = marca;
        this.motor = new Motor(potenciaMotor); // Composición
    }
    public void mostrarInformacion() {
        System.out.println("Coche: " + marca);
        motor.mostrarInformacion();
    }
}
public class Main {
    public static void main(String[] args) {
        Coche coche = new Coche("Toyota", 120);
        coche.mostrarInformacion();
    }
}
```

Salida:

```
Coche: Toyota
Potencia del motor: 120 HP
```

5. Herencia

La **herencia** permite que una clase (subclase) herede atributos y métodos de otra clase (superclase).

Ejemplo: Animales y Perros

```
public class Animal {
    private String nombre;
    public Animal(String nombre) {
          this.nombre = nombre;
    }
    public void mostrarInformacion() {
          System.out.println("Animal: " + nombre);
    }
}
public class Perro extends Animal {
    public Perro(String nombre) {
          super(nombre); // Llama al constructor de la superclase
    }
    public void ladrar() {
          System.out.println("¡Guau, guau!");
    }
}
public class Main {
    public static void main(String[] args) {
          Perro perro = new Perro("Rex");
          perro.mostrarInformacion();
          perro.ladrar();
    }
}
```

Salida:

```
Animal: Rex
¡Guau, guau!
```

6. Ejercicio Guiado

1. Crea una clase `Libro` con atributos:

- `titulo` (tipo `String`).
- `autor` (tipo `String`).

2. Crea una clase `Biblioteca` que contenga un array de libros.

3. Métodos:

- `agregarLibro(Libro libro)`: Añade un libro a la biblioteca.
- `mostrarLibros()`: Muestra todos los libros disponibles.

Clase `Libro`:

```
public class Libro {
    private String titulo;
    private String autor;
    public Libro(String titulo, String autor) {
          this.titulo = titulo;
          this.autor = autor;
    }
    public void mostrarInformacion() {
          System.out.println("Título: " + titulo + ", Autor: " + autor);
    }
}
```

Clase `Biblioteca`:

```
import java.util.ArrayList;
public class Biblioteca {
    private ArrayList<Libro> libros;
    public Biblioteca() {
          this.libros = new ArrayList<>();
    }
    public void agregarLibro(Libro libro) {
          libros.add(libro);
    }
    public void mostrarLibros() {
          System.out.println("Libros disponibles:");
          for (Libro libro : libros) {
                libro.mostrarInformacion();
          }
    }
}
```

Clase principal:

```
  public class Main {
      public static void main(String[] args) {
            Biblioteca biblioteca = new Biblioteca();
            Libro libro1 = new Libro("1984", "George Orwell");
            Libro libro2 = new Libro("Cien Años de Soledad", "Gabriel García
Márquez");
            biblioteca.agregarLibro(libro1);
            biblioteca.agregarLibro(libro2);
            biblioteca.mostrarLibros();
      }
  }
```

Salida:

Libros disponibles:

Título: 1984, Autor: George Orwell

Título: Cien Años de Soledad, Autor: Gabriel García Márquez

Parte 5: Principios básicos de la orientación a objetos.

La programación orientada a objetos (POO) se basa en cuatro principios fundamentales que permiten estructurar programas de forma lógica, reutilizable y escalable. Estos principios son:

1. **Abstracción**
2. **Encapsulación**
3. **Herencia**
4. **Polimorfismo**

Cada uno de estos conceptos se enfoca en diferentes aspectos del diseño y desarrollo de software, y juntos constituyen el núcleo de la POO.

1. Abstracción

La **abstracción** consiste en modelar entidades del mundo real seleccionando únicamente las características esenciales que son relevantes para el programa, ignorando los detalles irrelevantes.

Ejemplo práctico: Vehículos

1. **Clase abstracta: Define los métodos y atributos generales que todas las subclases deben implementar o heredar.**
2. **Subclases: Implementan o personalizan la funcionalidad según el tipo específico de objeto.**

```
// Clase abstracta
abstract class Vehiculo {
    protected String marca;
    public Vehiculo(String marca) {
        this.marca = marca;
    }
    public abstract void mover(); // Método abstracto
    public void mostrarMarca() {
        System.out.println("Marca: " + marca);
    }
}
// Subclase
class Coche extends Vehiculo {
    public Coche(String marca) {
        super(marca);
    }
    @Override
    public void mover() {
        System.out.println("El coche se mueve en la carretera.");
    }
}
// Clase principal
public class Main {
    public static void main(String[] args) {
        Vehiculo coche = new Coche("Toyota");
        coche.mostrarMarca();
        coche.mover();
    }
}
```

Salida:

```
Marca: Toyota
El coche se mueve en la carretera.
```

2. Encapsulación

El **encapsulamiento** se refiere a proteger los datos internos de una clase restringiendo su acceso directo desde fuera de la clase, como se explicó anteriormente. Esto permite validar y controlar cómo se interactúa con los atributos.

Ventajas:

1. Oculta la implementación interna.

2. Proporciona seguridad al proteger los datos de manipulación directa.

3. Facilita el mantenimiento del código.

3. Herencia

La **herencia** permite que una clase (subclase) reutilice atributos y métodos de otra clase (superclase). Esto promueve la reutilización de código y facilita la extensión de funcionalidades.

Ejemplo: Herencia de animales

```
// Superclase
class Animal {
    protected String nombre;
    public Animal(String nombre) {
          this.nombre = nombre;
    }
    public void comer() {
          System.out.println(nombre + " está comiendo.");
    }
}
// Subclase
class Perro extends Animal {
    public Perro(String nombre) {
          super(nombre);
    }
    public void ladrar() {
          System.out.println(nombre + " está ladrando: ¡Guau, guau!");
    }
}
// Clase principal
public class Main {
    public static void main(String[] args) {
          Perro perro = new Perro("Rex");
          perro.comer();
          perro.ladrar();
    }
}
```

Salida:

```
Rex está comiendo.
Rex está ladrando: ¡Guau, guau!
```

4. Polimorfismo

El **polimorfismo** permite que un objeto tome diferentes formas, es decir, un objeto de una subclase puede ser tratado como un objeto de su superclase. Esto se logra mediante:

1. **Sobrecarga de métodos: Métodos con el mismo nombre pero diferentes parámetros.**
2. **Sobreescritura de métodos: Una subclase redefine un método de la superclase.**

Ejemplo: Sobreescritura

```
// Superclase
class Figura {
    public void dibujar() {
        System.out.println("Dibujando una figura.");
    }
}
// Subclase
class Circulo extends Figura {
    @Override
    public void dibujar() {
        System.out.println("Dibujando un círculo.");
    }
}
// Clase principal
public class Main {
    public static void main(String[] args) {
        Figura figura1 = new Figura();
        Figura figura2 = new Circulo();
        figura1.dibujar();
        figura2.dibujar(); // Polimorfismo
    }
}
```

Salida:

```
Dibujando una figura.
Dibujando un círculo.
```

5. Ejercicio Guiado: Tienda de Instrumentos

1. Crea una clase abstracta `Instrumento` con los atributos:

- `nombre` (tipo `String`).
- Método abstracto `tocar()`.

2. Crea dos subclases:

- `Guitarra`: Implementa el método `tocar()` imprimiendo un mensaje relevante.
- `Piano`: Implementa el método `tocar()` imprimiendo otro mensaje relevante.

3. En la clase principal:

- Crea un array de instrumentos.
- Llena el array con guitarras y pianos.
- Recorre el array y llama al método `tocar()` para cada instrumento.

Solución:

Clase `Instrumento`:

```
abstract class Instrumento {
    protected String nombre;
    public Instrumento(String nombre) {
          this.nombre = nombre;
    }
    public abstract void tocar();
}
```

Subclase `Guitarra`:

```
class Guitarra extends Instrumento {
    public Guitarra(String nombre) {
          super(nombre);
    }
    @Override
    public void tocar() {
          System.out.println("Tocando la guitarra: " + nombre);
    }
}
```

Subclase `Piano`:

```
class Piano extends Instrumento {
    public Piano(String nombre) {
        super(nombre);
    }
    @Override
    public void tocar() {
        System.out.println("Tocando el piano: " + nombre);
    }
}
```

Clase principal:

```
public class Main {
    public static void main(String[] args) {
        Instrumento[] instrumentos = new Instrumento[4];
        instrumentos[0] = new Guitarra("Guitarra eléctrica");
        instrumentos[1] = new Piano("Piano de cola");
        instrumentos[2] = new Guitarra("Guitarra acústica");
        instrumentos[3] = new Piano("Piano vertical");
        for (Instrumento instrumento : instrumentos) {
            instrumento.tocar();
        }
    }
}
```

Salida:

```
Tocando la guitarra: Guitarra eléctrica
Tocando el piano: Piano de cola
Tocando la guitarra: Guitarra acústica
Tocando el piano: Piano vertical
```

6. Preguntas de Evaluación

1. ¿Qué diferencia hay entre una clase abstracta y una interfaz?

2. ¿Cuándo se utiliza la sobrecarga de métodos?

3. ¿Qué ventajas ofrece el polimorfismo en el diseño de sistemas?

Conclusión del Bloque: Introducción a la orientación a objetos

La **programación orientada a objetos (POO)** es un paradigma esencial para el desarrollo de software moderno, ofreciendo un enfoque estructurado, modular y escalable. A lo largo de este bloque, hemos explorado los conceptos fundamentales que permiten modelar y resolver problemas del mundo real mediante objetos y sus interacciones. A continuación, resumimos los puntos clave:

1. Puntos Principales

1. Clases, Atributos y Métodos:

- Las **clases** son la plantilla para crear objetos.
- Los **atributos** representan las características de un objeto.
- Los **métodos** definen su comportamiento.
- **Encapsulación** asegura que los datos internos estén protegidos y sean accesibles de forma controlada mediante getters y setters.

2. Objetos, Estado, Comportamiento e Identidad:

- Un **objeto** es una instancia de una clase, con un estado único (valores de atributos) y comportamientos (métodos).
- Los **mensajes** son la forma en que los objetos interactúan, invocando métodos en otros objetos.

3. Encapsulado y Visibilidad:

- La **visibilidad** de los atributos y métodos, controlada por modificadores como `private`, `protected` y `public`, garantiza que los datos sean seguros y manejados adecuadamente.
- El **encapsulamiento** fomenta el uso de métodos públicos para exponer comportamientos y esconder detalles de implementación.

4. Relaciones entre Clases:

- Las **asociaciones** modelan las relaciones lógicas entre objetos.
- La **agregación** y la **composición** representan cómo un objeto puede contener a otros, con diferentes niveles de dependencia.
- La **herencia** permite reutilizar código y modelar jerarquías.
- Las **dependencias** ayudan a crear interacciones temporales entre clases.

5. Principios Básicos:

- **Abstracción** simplifica problemas complejos enfocándose en características esenciales.
- **Herencia** reutiliza código existente y crea relaciones jerárquicas.
- **Encapsulación** protege datos y asegura la coherencia.
- **Polimorfismo** permite que un objeto tome diferentes formas, facilitando la extensibilidad y el mantenimiento.

2. Ventajas de la Programación Orientada a Objetos

1. Reutilización del Código:

- La herencia y los principios de diseño orientado a objetos permiten que las clases sean reutilizables en diferentes contextos.

2. Mantenimiento Simplificado:

- La encapsulación facilita la modificación de clases sin afectar otras partes del sistema.

3. Escalabilidad y Modularidad:

- Los sistemas orientados a objetos son fáciles de escalar, ya que se pueden añadir nuevas clases o funcionalidades sin alterar significativamente el diseño existente.

4. Modelado del Mundo Real:

- POO permite mapear conceptos del mundo real en código, haciendo que el diseño sea más intuitivo y comprensible.

3. Ejercicio Final: Sistema de Gestión de Biblioteca

Objetivo: Aplicar todos los conceptos aprendidos sobre la orientación a objetos en un proyecto completo.

Diseña un sistema de gestión de una biblioteca que permita realizar las siguientes acciones:

1. Gestión de libros:

- Cada libro debe tener los atributos:
  - `titulo` (tipo `String`).
  - `autor` (tipo `String`).
  - `disponible` (tipo `boolean`), que indica si el libro está disponible para préstamo.
- Métodos:
  - `mostrarInformacion()`: Muestra todos los detalles del libro.
  - `prestar()`: Cambia el estado del libro a `no disponible` si está disponible.
  - `devolver()`: Cambia el estado del libro a `disponible` si está prestado.

2. Gestión de usuarios:

- Cada usuario debe tener:
  - `nombre` (tipo `String`).
  - `prestamos` (array de `Libros` con un máximo de 3).
- Métodos:
  - `mostrarInformacion()`: Muestra el nombre del usuario y los libros que tiene prestados.
  - `prestarLibro(Libro libro)`: Agrega un libro al array de préstamos si el usuario no ha alcanzado el límite y si el libro está disponible.
  - `devolverLibro(Libro libro)`: Devuelve un libro, eliminándolo de los préstamos.

3. Biblioteca:

- La biblioteca debe contener:
  - Una lista de libros (`ArrayList<Libro>`).
  - Métodos:
    - `agregarLibro(Libro libro)`: Agrega un libro a la biblioteca.
    - `mostrarLibrosDisponibles()`: Muestra todos los libros disponibles para préstamo.

4. Implementa un flujo en el programa principal que:

- Cree varios libros y usuarios.
- Realice préstamos y devoluciones.
- Muestra el estado final de los usuarios y los libros.

Solución

Clase `Libro`:

```
  public class Libro {
      private String titulo;
      private String autor;
      private boolean disponible;
      public Libro(String titulo, String autor) {
            this.titulo = titulo;
            this.autor = autor;
            this.disponible = true;
      }
      public void mostrarInformacion() {
            System.out.println("Título: " + titulo + ", Autor: " + autor + ",
Disponible: " + (disponible ? "Sí" : "No"));
      }
      public boolean isDisponible() {
            return disponible;
      }
      public void prestar() {
            if (disponible) {
                  disponible = false;
                  System.out.println("El libro '" + titulo + "' ha sido presta-
do.");
            } else {
                  System.out.println("El libro '" + titulo + "' no está dis-
ponible.");
            }
      }
      public void devolver() {
            disponible = true;
            System.out.println("El libro '" + titulo + "' ha sido devuelto.");
      }
  }
```

Clase `Usuario`:

```
import java.util.ArrayList;
public class Usuario {
    private String nombre;
    private ArrayList<Libro> prestamos;
    public Usuario(String nombre) {
        this.nombre = nombre;
        this.prestamos = new ArrayList<>();
    }
    public void mostrarInformacion() {
        System.out.println("Usuario: " + nombre);
        System.out.println("Libros prestados:");
        if (prestamos.isEmpty()) {
            System.out.println("No tiene libros prestados.");
        } else {
            for (Libro libro : prestamos) {
                System.out.println(" - " + libro);
            }
        }
    }
    public void prestarLibro(Libro libro) {
        if (prestamos.size() < 3 && libro.isDisponible()) {
            prestamos.add(libro);
            libro.prestar();
        } else if (!libro.isDisponible()) {
            System.out.println("El libro '" + libro + "' no está dis-
ponible.");
        } else {
            System.out.println("No puedes prestar más libros.");
        }
    }
    public void devolverLibro(Libro libro) {
        if (prestamos.remove(libro)) {
            libro.devolver();
        } else {
            System.out.println("El libro '" + libro + "' no está entre tus
préstamos.");
        }
    }
}
```

Clase `Biblioteca`**:**

```
import java.util.ArrayList;
public class Biblioteca {
    private ArrayList<Libro> libros;
    public Biblioteca() {
          this.libros = new ArrayList<>();
    }
    public void agregarLibro(Libro libro) {
          libros.add(libro);
    }
    public void mostrarLibrosDisponibles() {
          System.out.println("Libros disponibles:");
          for (Libro libro : libros) {
                if (libro.isDisponible()) {
                      libro.mostrarInformacion();
                }
          }
    }
}
```

Clase Principal:

```
  public class Main {
      public static void main(String[] args) {
            Biblioteca biblioteca = new Biblioteca();
            // Crear libros
            Libro libro1 = new Libro("1984", "George Orwell");
            Libro libro2 = new Libro("El principito", "Antoine de Saint-Ex-
upéry");
            Libro libro3 = new Libro("Don Quijote", "Miguel de Cervantes");
            // Agregar libros a la biblioteca
            biblioteca.agregarLibro(libro1);
            biblioteca.agregarLibro(libro2);
            biblioteca.agregarLibro(libro3);
            // Crear usuarios
            Usuario usuario1 = new Usuario("Ana");
            Usuario usuario2 = new Usuario("Carlos");
            // Mostrar libros disponibles
            System.out.println("\n--- Libros disponibles ---");
            biblioteca.mostrarLibrosDisponibles();
            // Realizar préstamos
            System.out.println("\n--- Préstamos ---");
            usuario1.prestarLibro(libro1);
            usuario2.prestarLibro(libro2);
            usuario1.prestarLibro(libro3);
            // Mostrar libros disponibles después de los préstamos
            System.out.println("\n--- Libros disponibles después de préstamos
---");
            biblioteca.mostrarLibrosDisponibles();
            // Realizar devoluciones
            System.out.println("\n--- Devoluciones ---");
            usuario1.devolverLibro(libro1);
            // Mostrar estado final
            System.out.println("\n--- Estado Final ---");
            usuario1.mostrarInformacion();
            usuario2.mostrarInformacion();
            biblioteca.mostrarLibrosDisponibles();
      }
  }
```

4. Test de Evaluación

1. ¿Qué principio de la orientación a objetos permite reutilizar atributos y métodos en una subclase?

a) Polimorfismo
b) Herencia
c) Encapsulación
d) Abstracción

2. ¿Cuál es la principal diferencia entre agregación y composición?

a) En la composición, los objetos tienen ciclos de vida independientes.
b) En la agregación, los objetos tienen ciclos de vida dependientes.
c) En la composición, los objetos comparten atributos.
d) En la agregación, los objetos desaparecen juntos.

3. ¿Qué ventaja tiene el encapsulamiento?

a) Permite cambiar el valor de los atributos directamente.
b) Proporciona seguridad al proteger los datos internos.
c) Facilita la reutilización del código.
d) Permite que una clase tome múltiples formas.

4. ¿Qué modificador de visibilidad restringe el acceso a los atributos solo dentro de la clase?

a) `public`
b) `private`
c) `protected`
d) `default`

5. **En el contexto del polimorfismo, ¿qué permite la sobreescritura de métodos?**
- a) Declarar métodos con el mismo nombre en una clase pero con diferentes parámetros.
- b) Implementar métodos heredados de forma específica en una subclase.
- c) Usar un método estático sin necesidad de instanciar la clase.
- d) Definir métodos abstractos en una clase concreta.

Soluciones del Test

1. b): La herencia permite que una subclase reutilice atributos y métodos de su superclase.
2. a): En la composición, los objetos tienen ciclos de vida dependientes.
3. b): El encapsulamiento protege los datos internos y asegura su integridad.
4. b): El modificador `private` restringe el acceso a los atributos únicamente dentro de la clase.
5. b): La sobreescritura permite implementar métodos heredados de manera específica en una subclase.