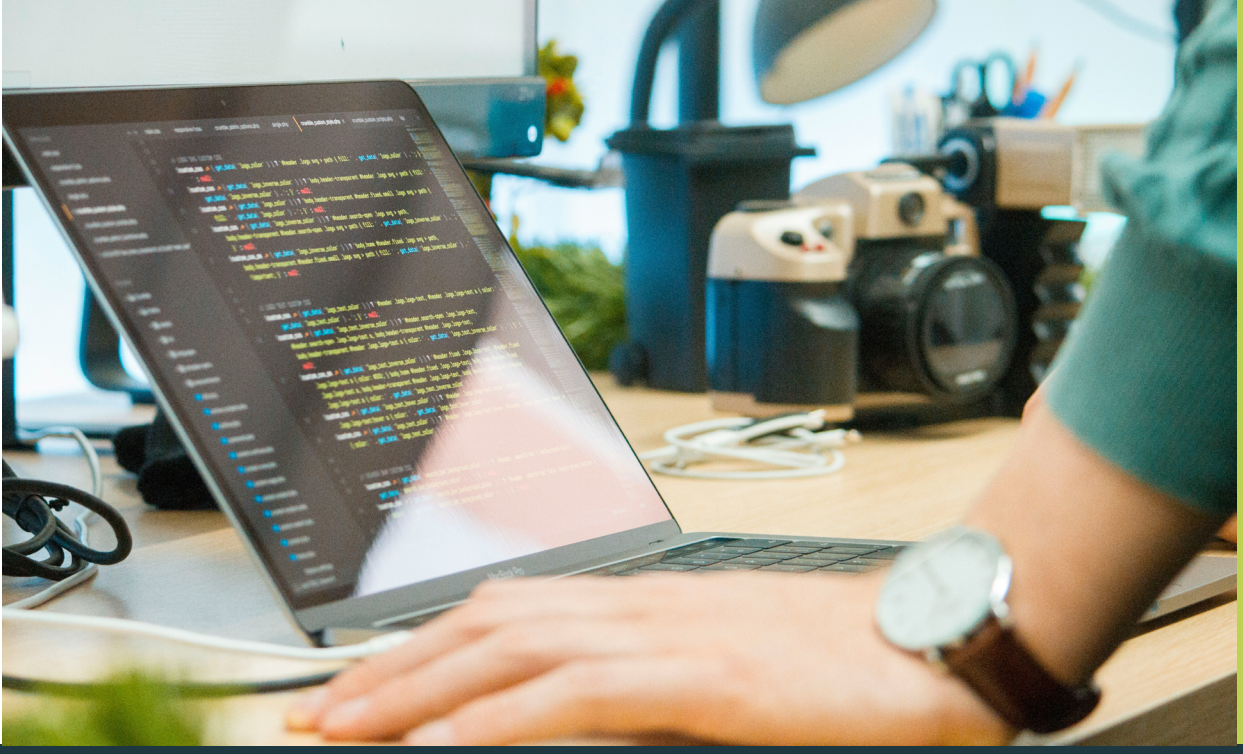

Unidad 8

Desarrollo de clases

Contenidos

Parte 1: Concepto de clase

Una **clase** es el concepto central de la programación orientada a objetos en Java. Una clase es un **molde** o **plantilla** que define las características (**atributos**) y comportamientos (**métodos**) que pueden tener los objetos creados a partir de ella.

1.Analogía con el Mundo Real

Imagina una clase como un **plano arquitectónico**:

- La **clase** define cómo será la estructura (características y comportamientos).
- Los **objetos** son las casas reales construidas usando ese plano.

Ejemplo:

- **Clase**: Automóvil.
- **Atributos**: color, marca, modelo.
- **Métodos**: acelerar, frenar, girar.

2. Definición de Clase en Java

En Java, una clase se declara usando la palabra clave `class`.

Sintaxis

```
public class NombreClase {
    // Atributos (variables de clase)
    // Métodos (funciones de clase)
}
```

3. Ejemplo Básico de Clase

Definición de una clase `Persona` con atributos y métodos:

```
  public class Persona {
      // Atributos de la clase
      String nombre;
      int edad;
      // Método de la clase
      public void saludar() {
            System.out.println("¡Hola! Mi nombre es " + nombre + " y tengo " +
edad + " años.");
      }
  }
```

Uso de la clase en el programa principal:

```
  public class Main {
      public static void main(String[] args) {
            // Crear un objeto de la clase Persona
            Persona persona1 = new Persona();
            persona1.nombre = "Juan";
            persona1.edad = 25;
            // Llamar al método saludar
            persona1.saludar();
      }
  }
```

Salida:

```
  ¡Hola! Mi nombre es Juan y tengo 25 años.
```

4. Ejercicio Práctico

Problema

Crea una clase `Libro` que tenga los siguientes atributos y métodos:

- **Atributos**: `titulo`, `autor` y `numPaginas`.
- **Método**: `mostrarInformacion` que imprima los detalles del libro.

Pistas

1. Declara la clase `Libro`.
2. Declara los atributos dentro de la clase.
3. Define el método `mostrarInformacion`.
4. Crea un objeto de la clase en el `main` y prueba el método.

5. Preguntas de Evaluación

1. ¿Qué es una clase en Java?

a) Un tipo de dato primitivo.
b) Una plantilla para crear objetos.
c) Un archivo ejecutable.
d) Una variable.

2. ¿Qué palabra clave se usa para definir una clase?

a) `class`
b) `struct`
c) `object`
d) `template`

3. ¿Qué representan los atributos de una clase?

a) Funcionalidades.
b) Datos o características.
c) Acciones del objeto.
d) Ninguna de las anteriores.

4. ¿Qué es un objeto?

a) Una función.
b) Una instancia de una clase.
c) Una variable global.
d) Un método estático.

Parte 2: Estructura y miembros de una clase

Una **clase** en Java está formada por diferentes **miembros**, que son los componentes que la definen y le permiten cumplir con su propósito. Los miembros principales de una clase son:

1. **Atributos: Variables que almacenan el estado de la clase.**
2. **Métodos: Funciones que definen el comportamiento de la clase.**
3. **Constructores: Métodos especiales que se ejecutan al crear un objeto.**
4. **Bloques de inicialización: Código que se ejecuta antes de crear un objeto.**
5. **Clases anidadas: Clases declaradas dentro de otras clases.**

1. Estructura de una Clase en Java

La estructura básica de una clase en Java incluye los miembros mencionados anteriormente y sigue un orden general:

Sintaxis

```
[modificadores] class NombreClase {
    // Atributos (variables de instancia o clase)
    // Constructores
    // Métodos (funciones de clase)
    // Bloques de inicialización
    // Clases anidadas (opcional)
}
```

2. Atributos

Los **atributos** representan las características o datos de un objeto y se declaran como variables dentro de la clase.

Tipos de Atributos

1. **Atributos de instancia: Pertenecen a un objeto específico de la clase.**
2. **Atributos de clase (estáticos): Compartidos por todos los objetos de la clase.**

Ejemplo: Atributos

```
public class Coche {
    // Atributos de instancia
    String color;
    String modelo;
    // Atributo de clase (estático)
    static int numCoches = 0;
}
```

3. Métodos

Los **métodos** definen el comportamiento de una clase. Pueden ser de instancia (se ejecutan en un objeto específico) o estáticos (pertenecen a la clase).

Componentes de un Método

- **Modificadores**: Controlan la visibilidad (`public`, `private`, etc.).
- **Tipo de retorno**: Tipo de dato que devuelve el método (`void` si no devuelve nada).
- **Nombre del método**: Identificador del método.
- **Parámetros**: Datos que recibe el método como entrada.
- **Cuerpo del método**: Instrucciones a ejecutar.

Ejemplo: Métodos

```
public class Coche {
    String color;
    String modelo;
    // Método de instancia
    public void mostrarInformacion() {
        System.out.println("Color: " + color + ", Modelo: " + modelo);
    }
    // Método estático
    public static void mostrarMensaje() {
        System.out.println("Este es un mensaje de la clase Coche.");
    }
}
```

4. Constructores

Los **constructores** son métodos especiales que se llaman automáticamente cuando se crea un objeto. Se utilizan para inicializar atributos.

Características de los Constructores

1. **Tienen el mismo nombre que la clase.**
2. **No tienen tipo de retorno (ni siquiera `void`).**
3. **Pueden ser sobrecargados (más adelante se profundizará en este concepto).**

Ejemplo: Constructor

```
public class Coche {
    String color;
    String modelo;
    // Constructor
    public Coche(String color, String modelo) {
        this.color = color;
        this.modelo = modelo;
    }
    public void mostrarInformacion() {
        System.out.println("Color: " + color + ", Modelo: " + modelo);
    }
}
```

Uso del constructor en el `main`:

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

Salida:

```
Color: Rojo, Modelo: Toyota
```

5. Bloques de Inicialización

Los **bloques de inicialización** son segmentos de código que se ejecutan antes del constructor al crear un objeto.

Ejemplo: Bloque de Inicialización

```
public class Coche {
    String color;
    // Bloque de inicialización
    {
        System.out.println("Se está creando un objeto Coche.");
    }
    public Coche(String color) {
        this.color = color;
        System.out.println("Color: " + color);
    }
}
```

Salida al crear un objeto:

```
Se está creando un objeto Coche.
Color: Azul
```

6. Clases Anidadas

Una **clase anidada** es una clase definida dentro de otra clase. Se utiliza para organizar el código y establecer relaciones más fuertes entre clases.

Ejemplo: Clase Anidada

```
public class Coche {
    String color;
    public class Motor {
        String tipoMotor;
        public Motor(String tipoMotor) {
            this.tipoMotor = tipoMotor;
        }
        public void mostrarMotor() {
            System.out.println("Tipo de motor: " + tipoMotor);
        }
    }
}
```

7. Ejercicio Práctico

Problema

Crea una clase `Rectangulo` que:

1. **Tenga atributos `base` y `altura`.**
2. **Tenga un constructor para inicializar los atributos.**
3. **Tenga un método para calcular el área (`calcularArea`).**
4. **Tenga un método estático que indique cuántos rectángulos se han creado.**

8. Preguntas de Evaluación

1. ¿Qué componente de una clase inicializa los atributos al crear un objeto?

a) Método
b) Constructor
c) Bloque estático
d) Clase anidada

2. ¿Qué atributo es compartido entre todos los objetos de una clase?

a) De instancia
b) Estático
c) Privado
d) Final

3. ¿Qué palabra clave se usa para acceder a miembros de la clase desde un objeto?

a) `new`
b) `class`
c) `this`
d) `static`

Parte 3: Creación de atributos. Declaración e inicialización.

Los **atributos** de una clase son las variables que definen el **estado** de los objetos creados a partir de esa clase. Un atributo puede ser una **variable de instancia** (propia de cada objeto) o una **variable estática** (compartida por todos los objetos de la clase).

1. Declaración de Atributos

La declaración de un atributo incluye:

1. Modificador de visibilidad (`public`, `private`, `protected` o sin modificador).

2. Tipo de dato: El tipo del atributo (primitivo u objeto).

3. Nombre del atributo: Identificador que se utilizará para acceder al atributo.

1.1. Ejemplo Básico de Declaración

```
public class Coche {
    // Atributos de instancia
    private String color;
    private String modelo;
    private int velocidad;
    // Atributo estático (compartido entre todos los objetos)
    public static int numCoches = 0;
}
```

2. Inicialización de Atributos

Los atributos pueden inicializarse de varias formas:

1. Inicialización directa: Se asigna un valor al declarar el atributo.

2. Inicialización en el constructor: Se asignan valores al crear el objeto.

3. Inicialización mediante métodos (setters): Se asignan valores después de crear el objeto.

4. Inicialización en bloques de inicialización.

2.1. Inicialización Directa

Los atributos pueden ser inicializados al momento de declararlos.

```
  public class Coche {
      private String color = “Blanco”;
      private String modelo = “Genérico”;
      private int velocidad = 0;
      public void mostrarInformacion() {
            System.out.println(“Color: “ + color + “, Modelo: “ + modelo + “,
Velocidad: “ + velocidad);
      }
  }
```

Uso:

```
  public class Main {
      public static void main(String[] args) {
            Coche coche1 = new Coche();
            coche1.mostrarInformacion();
      }
  }
```

Salida:

```
  Color: Blanco, Modelo: Genérico, Velocidad: 0
```

2.2. Inicialización en el Constructor

La inicialización más común de los atributos se realiza a través de un **constructor**, permitiendo asignar valores al momento de crear el objeto.

```
  public class Coche {
      private String color;
      private String modelo;
      private int velocidad;
      // Constructor
      public Coche(String color, String modelo, int velocidad) {
            this.color = color;
            this.modelo = modelo;
            this.velocidad = velocidad;
      }
      public void mostrarInformacion() {
            System.out.println("Color: " + color + ", Modelo: " + modelo + ",
Velocidad: " + velocidad);
      }
  }
```

Uso:

```
  public class Main {
      public static void main(String[] args) {
            Coche coche1 = new Coche("Rojo", "Toyota", 120);
            Coche coche2 = new Coche("Azul", "Honda", 100);
            coche1.mostrarInformacion();
            coche2.mostrarInformacion();
      }
  }
```

Salida:

```
  Color: Rojo, Modelo: Toyota, Velocidad: 120
  Color: Azul, Modelo: Honda, Velocidad: 100
```

2.3. Inicialización mediante Métodos (Setters)

Se pueden inicializar o modificar los valores de los atributos usando **métodos setters**.

```
public class Coche {
    private String color;
    private String modelo;
    // Setter para el color
    public void setColor(String color) {
        this.color = color;
    }
    // Setter para el modelo
    public void setModelo(String modelo) {
        this.modelo = modelo;
    }
    public void mostrarInformacion() {
        System.out.println("Color: " + color + ", Modelo: " + modelo);
    }
}
```

Uso:

```
public class Main {
    public static void main(String[] args) {
        Coche coche = new Coche();
        coche.setColor("Verde");
        coche.setModelo("Ford");
        coche.mostrarInformacion();
    }
}
```

Salida:

```
Color: Verde, Modelo: Ford
```

2.4. Bloques de Inicialización

Los bloques de inicialización permiten ejecutar código adicional al inicializar los atributos.

```
public class Coche {
    private String color;
    private String modelo;
    // Bloque de inicialización
    {
          color = “Blanco”;
          modelo = “Genérico”;
          System.out.println(“Bloque de inicialización ejecutado.”);
    }
    public void mostrarInformacion() {
          System.out.println(“Color: “ + color + “, Modelo: “ + modelo);
    }
}
```

Uso:

```
public class Main {
    public static void main(String[] args) {
          Coche coche = new Coche();
          coche.mostrarInformacion();
    }
}
```

Salida:

```
Bloque de inicialización ejecutado.
Color: Blanco, Modelo: Genérico
```

3. Ejercicio Práctico

Problema

Crea una clase `Alumno` que tenga los siguientes atributos y características:

1. Atributos: `nombre`, `edad` **y** `notaMedia`**.**
2. Inicializa los atributos usando:

- Un **constructor**.
- Métodos **setters**.

3. Crea un método `mostrarInformacion` **que imprima los atributos del alumno.**

4. Preguntas de Evaluación

1. ¿Cómo se inicializan los atributos en un constructor?

a) Usando la palabra clave `this`.
b) Con el modificador `static`.
c) Llamando a métodos privados.
d) Usando el método `main`.

2. ¿Qué ventaja tiene usar setters para los atributos?

a) Evitar la encapsulación.
b) Validar los valores antes de asignarlos.
c) Incrementar la complejidad del código.
d) No tiene ninguna ventaja.

3. ¿Qué es un bloque de inicialización?

a) Un método especial para inicializar atributos.

- b) Código que se ejecuta antes del constructor.

c) Una clase anidada.
d) Un método estático.

4. ¿Qué ocurre si no se inicializa un atributo?

a) Se lanza un error en tiempo de ejecución.
b) Java asigna valores predeterminados.
c) El programa se detiene.
d) Los valores se inicializan en cero manualmente.

Parte 4: Creación de métodos. Declaración, argumentos y valores de retorno

Los **métodos** son bloques de código que ejecutan una serie de instrucciones y permiten definir el **comportamiento** de una clase. Los métodos son esenciales en Java, ya que organizan el código, promueven la reutilización y facilitan el mantenimiento.

1. Concepto de Método

Un método puede recibir **argumentos**, realizar operaciones y devolver **valores**. Los métodos encapsulan acciones o funciones específicas dentro de una clase.

1.2. Estructura de un Método

La estructura de un método en Java sigue esta sintaxis:

```
java
Copiar código
[modificadores] tipoDeRetorno nombreMetodo(parámetros) {
    // Cuerpo del método: instrucciones a ejecutar
    return valor; // Opcional, si no es void
}
```

Componentes del Método

1. **Modificadores: Controlan la visibilidad (`public`, `private`, etc.).**
2. **Tipo de retorno: El tipo de dato que devuelve el método (`void` si no devuelve nada).**
3. **Nombre del método: Identificador que describe su función.**
4. **Parámetros (argumentos): Variables que recibe el método como entrada.**
5. **Cuerpo del método: Bloque de código que realiza la acción.**
6. **`return`: Devuelve un valor al método que lo invoca (opcional si no es `void`).**

2. Declaración de Métodos

Métodos sin Retorno (`void`)

Un método puede no devolver ningún valor si su propósito es ejecutar una acción.

Ejemplo

```
public class Coche {
    String modelo;
    // Método sin retorno
    public void mostrarModelo() {
        System.out.println("El modelo del coche es: " + modelo);
    }
}
```

Uso:

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

Salida:

```
El modelo del coche es: Toyota
```

Métodos con Retorno

Un método puede devolver un valor usando la instrucción `return`. El tipo de retorno debe coincidir con el tipo declarado en la definición del método.

Ejemplo

```
public class Coche {
    int velocidad;
    // Método con retorno
    public int obtenerVelocidad() {
        return velocidad;
    }
}
```

Uso:

```
public class Main {
    public static void main(String[] args) {
        Coche coche = new Coche();
        coche.velocidad = 120;
        int v = coche.obtenerVelocidad();
        System.out.println("La velocidad del coche es: " + v);
    }
}
```

Salida:

```
La velocidad del coche es: 120
```

Métodos con Parámetros

Los métodos pueden recibir **parámetros** (datos de entrada) para realizar tareas dinámicas.

Ejemplo

```
public class Calculadora {
    // Método con parámetros y retorno
    public int sumar(int a, int b) {
        return a + b;
    }
}
```

Uso:

```
public class Main {
    public static void main(String[] args) {
        Calculadora calc = new Calculadora();
        int resultado = calc.sumar(10, 20);
        System.out.println("El resultado de la suma es: " + resultado);
    }
}
```

Salida:

```
El resultado de la suma es: 30
```

Métodos con Múltiples Parámetros

Un método puede recibir varios parámetros separados por comas.

Ejemplo

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

Uso:

```
public class Main {
    public static void main(String[] args) {
        Persona persona = new Persona();
        persona.mostrarInformacion("Ana", 25);
    }
}
```

Salida:

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

3. Sobrecarga de Métodos

La **sobrecarga de métodos** permite definir varios métodos con el mismo nombre pero diferentes parámetros (cantidad o tipo de datos).

Ejemplo de Sobrecarga

```
public class Calculadora {
    // Método sumar con 2 parámetros
    public int sumar(int a, int b) {
        return a + b;
    }
    // Sobrecarga: Método sumar con 3 parámetros
    public int sumar(int a, int b, int c) {
        return a + b + c;
    }
}
```

Uso:

```
public class Main {
    public static void main(String[] args) {
        Calculadora calc = new Calculadora();
        System.out.println("Suma de 2 números: " + calc.sumar(5, 10));
        System.out.println("Suma de 3 números: " + calc.sumar(5, 10, 15));
    }
}
```

Salida:

```
Suma de 2 números: 15
Suma de 3 números: 30
```

4. Ejercicio Práctico

Problema

Crea una clase `Triangulo` que:

1. **Tenga atributos `base` y `altura`.**
2. **Tenga un método para calcular el área (`calcularArea`).**
3. **Tenga un método sobrecargado para calcular el área usando solo la base (asumiendo un triángulo equilátero).**

Pista

- Usa la fórmula del área

5. Preguntas de Evaluación

1. ¿Qué palabra clave se utiliza para devolver un valor desde un método?

a) `return`
b) `break`
c) `void`
d) `continue`

2. ¿Qué ocurre si un método no tiene `return` y no es `void`?

a) No compila.
b) Devuelve un valor por defecto.
c) Lanza una excepción en tiempo de ejecución.
d) Ninguna de las anteriores.

3. ¿Qué permite la sobrecarga de métodos?

a) Cambiar el nombre del método.
b) Definir varios métodos con el mismo nombre pero diferentes parámetros.
c) Sobrescribir un método en una subclase.
d) Ninguna de las anteriores.

4. ¿Qué define el tipo de retorno de un método?

a) El nombre del método.
b) Los parámetros del método.
c) El valor que devuelve el método.
d) La clase del objeto.

Parte 5: Creación de constructores

Un **constructor** es un método especial de una clase que se utiliza para **inicializar objetos**. Se ejecuta automáticamente al crear un objeto y su función principal es asignar valores iniciales a los **atributos** de la clase.

1. Características de un Constructor

1. **Mismo nombre que la clase: El constructor debe tener el mismo nombre que la clase en la que está definido.**
2. **No tiene tipo de retorno: A diferencia de los métodos, un constructor no especifica un tipo de retorno, ni siquiera `void`.**
3. **Se ejecuta automáticamente: El constructor se invoca automáticamente cuando se crea un objeto con la palabra clave `new`.**
4. **Puede recibir parámetros: Los constructores permiten pasar valores iniciales a los atributos de la clase.**
5. **Sobrecarga de constructores: Es posible definir múltiples constructores en una clase (sobrecarga) con diferentes parámetros.**

2. Tipos de Constructores

En Java, existen dos tipos de constructores principales:

1. **Constructor por defecto: Es el constructor vacío que proporciona Java si no se define ningún constructor.**
2. **Constructor parametrizado: Constructor que acepta parámetros para inicializar los atributos de la clase.**

2.1. Constructor por Defecto

Si no definimos un constructor, Java proporciona un **constructor por defecto** de manera implícita. Este constructor **no realiza ninguna acción**, excepto inicializar los atributos con sus valores predeterminados.

Ejemplo: Constructor por Defecto

```
  public class Persona {
      String nombre;
      int edad;
      public void mostrarInformacion() {
            System.out.println("Nombre: " + nombre + ", Edad: " + edad);
      }
  }
  public class Main {
      public static void main(String[] args) {
            Persona persona1 = new Persona(); // Se invoca el constructor por
defecto
            persona1.mostrarInformacion();
      }
  }
```

Salida:

```
Nombre: null, Edad: 0
```

En este caso:

- Los atributos `nombre` y `edad` tienen los valores predeterminados (`null` para cadenas y `0` para enteros).

2.2. Constructor Parametrizado

Un **constructor parametrizado** permite pasar valores iniciales al crear un objeto. Esto ayuda a evitar valores predeterminados y garantiza que los atributos tengan valores significativos.

Ejemplo: Constructor Parametrizado

```
public class Persona {
    String nombre;
    int edad;
    // Constructor parametrizado
    public Persona(String nombre, int edad) {
        this.nombre = nombre; // "this" se refiere al atributo de la clase
        this.edad = edad;
    }
    public void mostrarInformacion() {
        System.out.println("Nombre: " + nombre + ", Edad: " + edad);
    }
}
public class Main {
    public static void main(String[] args) {
        // Crear objetos usando el constructor parametrizado
        Persona persona1 = new Persona("Ana", 25);
        Persona persona2 = new Persona("Carlos", 30);
        // Mostrar la información
        persona1.mostrarInformacion();
        persona2.mostrarInformacion();
    }
}
```

Salida:

```
Nombre: Ana, Edad: 25
Nombre: Carlos, Edad: 30
```

3. Sobrecarga de Constructores

La **sobrecarga de constructores** permite definir múltiples constructores en una clase con diferentes listas de parámetros. Esto proporciona flexibilidad al crear objetos.

Ejemplo: Sobrecarga de Constructores

```
public class Persona {
    String nombre;
    int edad;
    // Constructor por defecto
    public Persona() {
        this.nombre = “Desconocido”;
        this.edad = 0;
    }
    // Constructor con un parámetro
    public Persona(String nombre) {
        this.nombre = nombre;
        this.edad = 0;
    }
    // Constructor con dos parámetros
    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
    public void mostrarInformacion() {
        System.out.println(“Nombre: “ + nombre + “, Edad: “ + edad);
    }
}
public class Main {
    public static void main(String[] args) {
        Persona persona1 = new Persona();
        Persona persona2 = new Persona(“Ana”);
        Persona persona3 = new Persona(“Carlos”, 30);
        persona1.mostrarInformacion();
        persona2.mostrarInformacion();
        persona3.mostrarInformacion();
    }
}
```

Salida:

```
Nombre: Desconocido, Edad: 0
Nombre: Ana, Edad: 0
Nombre: Carlos, Edad: 30
```

4. Uso de `this` en Constructores

La palabra clave `this` se utiliza dentro de un constructor para:

1. Referirse a los atributos de la clase cuando tienen el mismo nombre que los parámetros.

2. Llamar a otro constructor dentro de la misma clase.

Ejemplo: Uso de `this`

```
public class Coche {
    String color;
    int velocidad;
    // Constructor con parámetros
    public Coche(String color, int velocidad) {
        this.color = color;
        this.velocidad = velocidad;
    }
    // Constructor que reutiliza otro constructor
    public Coche(String color) {
        this(color, 0); // Llama al constructor con dos parámetros
    }
    public void mostrarInformacion() {
        System.out.println("Color: " + color + ", Velocidad: " + velocidad +
" km/h");
    }
}
public class Main {
    public static void main(String[] args) {
        Coche coche1 = new Coche("Rojo", 120);
        Coche coche2 = new Coche("Azul");
        coche1.mostrarInformacion();
        coche2.mostrarInformacion();
    }
}
```

Salida:

```
Color: Rojo, Velocidad: 120 km/h
Color: Azul, Velocidad: 0 km/h
```

5. Ejercicio Práctico

Problema

Crea una clase `Empleado` que:

1. Tenga atributos `nombre`, `salario` y `departamento`.

2. Defina:

- Un **constructor por defecto** que inicialice los atributos con valores predeterminados.
- Un **constructor parametrizado** para asignar valores a todos los atributos.
- Un **constructor sobrecargado** que solo acepte `nombre` y `salario`.

3. Cree un método `mostrarInformacion` para mostrar los detalles del empleado.

6. Preguntas de Evaluación

1. ¿Qué hace un constructor?

a) Devuelve un valor.
b) Inicializa atributos de la clase.
c) Modifica los métodos de la clase.
d) Elimina objetos.

2. ¿Cuál es la diferencia entre un constructor por defecto y uno parametrizado?

a) El constructor por defecto no inicializa atributos.
b) El constructor parametrizado acepta valores para inicializar atributos.
c) El constructor por defecto no existe en Java.
d) No hay diferencia.

3. ¿Qué permite la sobrecarga de constructores?

a) Definir varios constructores con el mismo nombre pero diferentes parámetros.
b) Crear subclases de una clase.
c) Modificar atributos estáticos.
d) Llamar al constructor de otra clase.

4. ¿Qué palabra clave se utiliza para llamar a otro constructor dentro de la misma clase?

a) `this`
b) `super`
c) `return`
d) `new`

Parte 6: Ámbito de atributos y variables

El **ámbito** de los atributos y variables determina **dónde** pueden ser accedidos o utilizados dentro de un programa en Java. El ámbito también está relacionado con el **ciclo de vida** de las variables, es decir, cuándo se crean y cuándo dejan de existir.

1. Tipos de Ámbito en Java

En Java, el ámbito de una variable puede clasificarse en tres niveles principales:

1. Ámbito de clase (Variables de clase / atributos estáticos).

2. Ámbito de instancia (Variables de instancia).

3. Ámbito local (Variables locales y parámetros).

1.1. Ámbito de Clase (Variables Estáticas)

Las **variables de clase** son variables declaradas con el modificador `static` dentro de una clase, pero fuera de los métodos. Estas variables:

- **Pertenecen a la clase** en lugar de a un objeto específico.
- Son **compartidas** por todos los objetos de la clase.
- Existen durante toda la ejecución del programa, ya que son cargadas en memoria una sola vez.

Ejemplo: Variable de Clase

```
public class Contador {
    // Variable de clase (estática)
    public static int totalObjetos = 0;
    // Constructor
    public Contador() {
        totalObjetos++; // Incrementar la variable estática
    }
    public void mostrarTotal() {
        System.out.println("Total de objetos creados: " + totalObjetos);
    }
}
public class Main {
    public static void main(String[] args) {
        Contador c1 = new Contador();
        Contador c2 = new Contador();
        Contador c3 = new Contador();
        c1.mostrarTotal(); // Acceso desde objeto
        System.out.println("Acceso directo: " + Contador.totalObjetos); //
Acceso desde clase
    }
}
```

Salida:

```
Total de objetos creados: 3
Acceso directo: 3
```

1.2. Ámbito de Instancia (Variables de Instancia)

Las **variables de instancia** son atributos que pertenecen a un **objeto específico** de la clase. Estas variables:

- Se declaran fuera de los métodos.
- Se inicializan en el momento en que se crea un objeto con `new`.
- Tienen un **ciclo de vida** asociado al objeto: existen mientras el objeto exista.

Ejemplo: Variables de Instancia

```
public class Coche {
    // Variables de instancia
    private String color;
    private int velocidad;
    public Coche(String color, int velocidad) {
        this.color = color;
        this.velocidad = velocidad;
    }
    public void mostrarInformacion() {
        System.out.println("Color: " + color + ", Velocidad: " + velocidad);
    }
}
public class Main {
    public static void main(String[] args) {
        Coche coche1 = new Coche("Rojo", 120);
        Coche coche2 = new Coche("Azul", 150);
        coche1.mostrarInformacion();
        coche2.mostrarInformacion();
    }
}
```

Salida:

```
Color: Rojo, Velocidad: 120
Color: Azul, Velocidad: 150
```

1.3. Ámbito Local (Variables Locales)

Las **variables locales** son declaradas **dentro de un método, bloque o constructor**. Tienen las siguientes características:

- Solo existen dentro del bloque donde fueron declaradas.
- No tienen valores predeterminados; deben inicializarse explícitamente.
- Se destruyen automáticamente cuando el método termina.

Ejemplo: Variables Locales

```
public class Calculadora {
    public int sumar(int a, int b) { // 'a' y 'b' son parámetros locales
        int resultado = a + b; // Variable local
        return resultado;
    }
    public void mostrarMensaje() {
        String mensaje = "Hola, soy una variable local";
        System.out.println(mensaje);
    }
}
public class Main {
    public static void main(String[] args) {
        Calculadora calc = new Calculadora();
        int suma = calc.sumar(5, 10);
        System.out.println("Suma: " + suma);
        calc.mostrarMensaje();
    }
}
```

Salida:

```
Suma: 15
Hola, soy una variable local
```

2. Parámetros de Método

Los **parámetros de un método** también tienen **ámbito local**. Son variables que reciben valores cuando el método es invocado.

Ejemplo: Parámetros de Método

```
public class Multiplicador {
    public int multiplicar(int x, int y) { // 'x' y 'y' son parámetros
        return x * y;
    }
}
public class Main {
    public static void main(String[] args) {
        Multiplicador m = new Multiplicador();
        int resultado = m.multiplicar(3, 4);
        System.out.println("Resultado de la multiplicación: " + resultado);
    }
}
```

Salida:

```
Resultado de la multiplicación: 12
```

3. Ejercicio Práctico

Problema

Crea una clase `Empleado` que contenga lo siguiente:

1. **Atributos de instancia:** `nombre`, `salario`.
2. **Atributo estático:** `numEmpleados` **(número total de empleados).**
3. **Un constructor para inicializar** `nombre` **y** `salario`.
4. **Un método que muestre la información del empleado.**
5. **Un método que muestre el número total de empleados.**

4. Preguntas de Evaluación

1. ¿Dónde se almacenan las variables de instancia?

a) Stack
b) Heap
c) Método principal
d) Clase estática

2. ¿Qué sucede con una variable local después de que el método termina?

a) Se mantiene en la memoria.
b) Se destruye automáticamente.
c) Se convierte en una variable de instancia.
d) Nada.

3. ¿Cuál es la diferencia entre una variable estática y una de instancia?

a) La variable estática pertenece a la clase y la de instancia a un objeto.
b) Ambas son iguales.
c) La variable estática no puede ser modificada.
d) La de instancia es compartida entre todos los objetos.

4. ¿Qué palabra clave se usa para acceder a variables de clase estáticas?

a) `this`
b) `class`
c) El nombre de la clase
d) `new`

Parte 7: Sobrecarga de métodos

La **sobrecarga de métodos** (method overloading) es una característica de Java que permite definir **varios métodos con el mismo nombre** en una misma clase, pero con **diferentes listas de parámetros**. Es una técnica fundamental para lograr mayor **flexibilidad** y **reutilización** del código.

1. ¿Qué es la Sobrecarga de Métodos?

En la **sobrecarga de métodos**, los métodos comparten el **mismo nombre**, pero deben cumplir al menos una de las siguientes condiciones:

1. Tienen diferente número de parámetros.

2. Los parámetros son del mismo número, pero de diferentes tipos.

3. Los parámetros tienen un orden distinto.

Importante

- El **tipo de retorno** del método **no puede diferenciar** una sobrecarga.
- El compilador decide qué método llamar en función de los **argumentos** proporcionados.

1.1. Ventajas de la Sobrecarga de Métodos

1. Mejora la legibilidad: Un mismo nombre describe una acción con variantes.

2. Facilita la reutilización del código: Evita nombres redundantes para métodos similares.

3. Permite adaptabilidad: El método se adapta a diferentes tipos o cantidades de datos de entrada.

1.2. Ejemplo Básico de Sobrecarga de Métodos

Código

```
public class Calculadora {
    // Método sumar con 2 parámetros
    public int sumar(int a, int b) {
        return a + b;
    }
    // Método sumar con 3 parámetros
    public int sumar(int a, int b, int c) {
        return a + b + c;
    }
    // Método sumar con parámetros de tipo double
    public double sumar(double a, double b) {
        return a + b;
    }
    public static void main(String[] args) {
        Calculadora calc = new Calculadora();
        System.out.println("Suma de 2 números enteros: " + calc.sumar(5, 10));
        System.out.println("Suma de 3 números enteros: " + calc.sumar(5, 10, 15));
        System.out.println("Suma de 2 números decimales: " + calc.sumar(5.5, 10.2));
    }
}
```

Salida

```
Suma de 2 números enteros: 15
Suma de 3 números enteros: 30
Suma de 2 números decimales: 15.7
```

2. Sobrecarga de Métodos con Diferentes Tipos de Parámetros

Podemos definir métodos con el mismo nombre que acepten **diferentes tipos de parámetros**.

Código

```
public class Impresora {
    // Método para imprimir un entero
    public void imprimir(int numero) {
        System.out.println("Número entero: " + numero);
    }
    // Método para imprimir una cadena
    public void imprimir(String texto) {
        System.out.println("Texto: " + texto);
    }
    // Método para imprimir un número decimal
    public void imprimir(double numeroDecimal) {
        System.out.println("Número decimal: " + numeroDecimal);
    }
    public static void main(String[] args) {
        Impresora impresora = new Impresora();
        impresora.imprimir(42);
        impresora.imprimir("Hola, Mundo");
        impresora.imprimir(3.1415);
    }
}
```

Salida

```
Número entero: 42
Texto: Hola, Mundo
Número decimal: 3.1415
```

3. Sobrecarga con Diferente Orden de Parámetros

El orden de los parámetros también puede ser diferente para que los métodos sean considerados como sobrecargados.

Código

```
public class Operacion {
    // Método con orden de parámetros int, double
    public void mostrarResultado(int a, double b) {
        System.out.println("Entero: " + a + ", Decimal: " + b);
    }
    // Método con orden de parámetros double, int
    public void mostrarResultado(double a, int b) {
        System.out.println("Decimal: " + a + ", Entero: " + b);
    }
    public static void main(String[] args) {
        Operacion op = new Operacion();
        op.mostrarResultado(5, 3.2);
        op.mostrarResultado(3.2, 5);
    }
}
```

Salida

```
Entero: 5, Decimal: 3.2
Decimal: 3.2, Entero: 5
```

4. Ejercicio Práctico

Problema

Crea una clase `Area` que contenga métodos sobrecargados para calcular el área de:

1. **Un círculo (con radio como parámetro).**
2. **Un rectángulo (con base y altura como parámetros).**
3. **Un triángulo (con base y altura como parámetros).**

Solución

```
public class Area {
    // Método para calcular el área de un círculo
    public double calcularArea(double radio) {
        return Math.PI * radio * radio;
    }
    // Método para calcular el área de un rectángulo
    public double calcularArea(double base, double altura) {
        return base * altura;
    }
    // Método para calcular el área de un triángulo
    public double calcularArea(int base, int altura) {
        return (base * altura) / 2.0;
    }
    public static void main(String[] args) {
        Area area = new Area();
        System.out.println("Área del círculo: " + area.calcularArea(5.0));
        System.out.println("Área del rectángulo: " + area.calcularArea(10.0,
5.0));
        System.out.println("Área del triángulo: " + area.calcularArea(10,
8));
    }
}
```

Salida:

```
Área del círculo: 78.53981633974483
Área del rectángulo: 50.0
Área del triángulo: 40.0
```

5. Preguntas de Evaluación

1. ¿Qué permite la sobrecarga de métodos?

a) Crear métodos con el mismo nombre pero diferentes parámetros.
b) Modificar un método en una subclase.
c) Usar un solo método para todas las clases.
d) Hacer que un método devuelva varios valores.

2. ¿Qué diferencia debe existir entre métodos sobrecargados?

a) Tipo de retorno.
b) Nombre del método.
c) Número o tipo de parámetros.
d) Comentarios en el código.

3. ¿Cuál de las siguientes opciones es válida para sobrecargar un método?

a) Cambiar el tipo de retorno únicamente.
b) Cambiar el nombre del método.
c) Cambiar el número o tipo de parámetros.
d) Usar `static` para diferenciar los métodos.

4. ¿Qué ocurre si llamas a un método sobrecargado con parámetros que coinciden con varios métodos?

a) Se produce un error de compilación.
b) El compilador elige el método más específico.
c) Se ejecutan todos los métodos.
d) Ninguna de las anteriores.

Parte 8: Visibilidad. Modificadores de clase, de atributos y de métodos

En Java, la **visibilidad** de clases, atributos y métodos se controla mediante **modificadores de acceso**. Estos modificadores determinan **dónde** y **cómo** se puede acceder a un miembro de la clase.

1. Modificadores de Acceso

Java proporciona **cuatro niveles de acceso**:

Modificador	Acceso en la misma clase	Acceso en el mismo paquete	Acceso en subclases	Acceso desde fuera
public	√ Sí	√ Sí	√ Sí	√ Sí
protected	√ Sí	√ Sí	√ Sí	× No
(default)	√ Sí	√ Sí	× No	× No
private	√ Sí	× No	× No	× No

- `public`: Accesible desde cualquier lugar.
- `protected`: Accesible desde clases del mismo paquete y subclases.
- **Sin modificador (default)**: Accesible únicamente dentro del mismo paquete.
- `private`: Accesible **solo dentro de la misma clase**.

2. Modificadores de Clase

1. Clases públicas: Se declaran con la palabra clave `public` y pueden ser accedidas desde cualquier lugar.

```
public class MiClase {
    // Código de la clase
}
```

1. Clases sin modificador (default): Solo son accesibles dentro del mismo paquete.

```
class MiClase {
    // Código de la clase
}
```

Nota: No se puede usar private o protected en clases de nivel superior.

3. Modificadores de Atributos

Los **atributos** pueden declararse con cualquiera de los modificadores de acceso:

- `private`: Encapsula los datos, permitiendo el acceso solo a través de métodos (setters y getters).
- `public`: Permite acceso directo.
- `protected`: Accesible desde el mismo paquete y subclases.
- **Sin modificador**: Accesible solo desde el mismo paquete.

Ejemplo: Modificadores de Atributos

```
public class Persona {
    // Atributos con diferentes niveles de acceso
    public String nombre;       // Accesible desde cualquier lugar
    protected int edad;           // Accesible en el paquete y subclases
    private String dni;           // Solo accesible dentro de la clase
    // Métodos públicos para acceder a atributos privados (encapsulación)
    public void setDni(String dni) {
          this.dni = dni;
    }
    public String getDni() {
          return dni;
    }
}
```

Uso de la Clase con Modificadores

```
public class Main {
    public static void main(String[] args) {
          Persona persona = new Persona();
          // Acceso a atributos públicos
          persona.nombre = “Ana”;
          // Acceso a atributos protegidos y privados mediante métodos
          persona.setDni(“12345678A”);
          System.out.println(“Nombre: “ + persona.nombre);
          System.out.println(“DNI: “ + persona.getDni());
    }
}
```

Salida:

```
Nombre: Ana
DNI: 12345678A
```

4. Modificadores de Métodos

Los **métodos** también pueden usar modificadores de acceso para controlar su visibilidad.

1. `public`: **El método puede ser accedido desde cualquier clase.**
2. `protected`: **Accesible en el mismo paquete y subclases.**
3. `private`: **El método solo puede ser accedido dentro de la misma clase.**
4. **Sin modificador: Accesible solo dentro del mismo paquete.**

Ejemplo: Modificadores de Métodos

```
public class Calculadora {
    // Método público
    public int sumar(int a, int b) {
        return a + b;
    }
    // Método protegido
    protected int restar(int a, int b) {
        return a - b;
    }
    // Método privado
    private int multiplicar(int a, int b) {
        return a * b;
    }
    // Método para acceder al método privado
    public void mostrarMultiplicacion(int a, int b) {
        System.out.println("Multiplicación: " + multiplicar(a, b));
    }
}
```

Uso de la clase:

```
public class Main {
    public static void main(String[] args) {
        Calculadora calc = new Calculadora();
        System.out.println("Suma: " + calc.sumar(10, 5));
        System.out.println("Resta: " + calc.restar(10, 5));
        // Acceso al método privado a través de un método público
        calc.mostrarMultiplicacion(10, 5);
    }
}
```

Salida:

```
Suma: 15
Resta: 5
Multiplicación: 50
```

5. Ejercicio Práctico

Problema

1. Crea una clase `CuentaBancaria` con los siguientes atributos:

- `numeroCuenta` (público)
- `saldo` (privado)
- `titular` (protegido)

2. Crea los métodos necesarios para:

- Consultar el saldo.
- Realizar un depósito.
- Realizar un retiro (verifica si el saldo es suficiente).

3. Encapsula el atributo `saldo` usando setters y getters.

6. Preguntas de Evaluación

1. ¿Qué modificador permite el acceso a un atributo desde cualquier clase?

a) `private`
b) `protected`
c) `public`
d) Sin modificador

2. ¿Cuál de los siguientes modificadores limita el acceso solo a la misma clase?

a) `public`
b) `protected`
c) `private`
d) Sin modificador

3. ¿Qué modificador de acceso permite que un atributo sea visible solo en el mismo paquete?

a) `public`
b) `protected`
c) `default` (sin modificador)
d) `private`

4. ¿Cómo se accede a un atributo privado?

a) Usando la palabra clave `this`.
b) Mediante un método setter o getter.
c) Directamente desde otra clase.
d) Con la palabra clave `public`.

5. ¿Qué modificador permite el acceso a métodos en subclases, pero no desde otras clases externas?

a) `private`
b) `protected`
c) `public`
d) Sin modificador

Parte 9: Paso de parámetros. Paso por valor y paso por referencia

En Java, cuando se llama a un método y se pasan parámetros, estos pueden ser tratados de dos maneras: **paso por valor** y **paso por referencia**. Es fundamental comprender cómo Java maneja los parámetros para evitar confusiones y errores.

1. Paso de Parámetros en Java

Java **siempre** utiliza el **paso por valor** para enviar parámetros a un método. Esto significa que:

1. Para tipos primitivos: Se pasa una copia del valor.

2. Para objetos: Se pasa una copia de la referencia al objeto, pero no el objeto en sí.

1.1. Paso por Valor: Tipos Primitivos

Cuando se pasa un **tipo primitivo** a un método, se envía una **copia del valor**. Por lo tanto, cualquier modificación que se realice dentro del método **no afecta el valor original** fuera del método.

Ejemplo: Paso por Valor con Tipos Primitivos

```
public class EjemploPrimitivo {
    public static void modificar(int numero) {
          numero = numero * 2;
          System.out.println("Valor dentro del método: " + numero);
    }
    public static void main(String[] args) {
          int valor = 10;
          System.out.println("Valor antes del método: " + valor);
          modificar(valor);
          System.out.println("Valor después del método: " + valor);
    }
}
```

Salida:

```
Valor antes del método: 10
Valor dentro del método: 20
Valor después del método: 10
```

Explicación:

1. En el método `modificar`, se recibe una copia del valor de `valor`.

2. Aunque el valor se modifica dentro del método, la copia del valor no afecta a la variable original `valor` en `main`.

1.2. Paso por Valor: Referencias a Objetos

Cuando se pasa un **objeto** a un método, se envía una **copia de la referencia** que apunta al objeto en memoria. Aunque la referencia se copia, ambas referencias (la original y la copia) apuntan al mismo objeto. Por lo tanto:

1. **Modificar el estado interno del objeto afecta al objeto original.**
2. **Si se asigna una nueva referencia dentro del método, la original no cambia.**

Ejemplo: Paso por Valor con Objetos

```
public class Persona {
    String nombre;
    public Persona(String nombre) {
        this.nombre = nombre;
    }
    public static void cambiarNombre(Persona persona) {
        persona.nombre = "Carlos"; // Modifica el estado interno del objeto
        System.out.println("Nombre dentro del método: " + persona.nombre);
    }
    public static void main(String[] args) {
        Persona persona1 = new Persona("Ana");
        System.out.println("Nombre antes del método: " + persona1.nombre);
        cambiarNombre(persona1);
        System.out.println("Nombre después del método: " + persona1.nombre);
    }
}
```

Salida:

```
Nombre antes del método: Ana
Nombre dentro del método: Carlos
Nombre después del método: Carlos
```

Explicación:

1. **Se pasa una copia de la referencia del objeto `persona1` al método `cambiarNombre`.**
2. **El método modifica el estado interno del objeto (`nombre`), lo cual afecta al objeto original.**

Importante

Si intentamos asignar una **nueva referencia** al objeto dentro del método, la referencia original no se verá afectada.

Ejemplo: Nueva Referencia

```
public class Persona {
    String nombre;
    public Persona(String nombre) {
        this.nombre = nombre;
    }
    public static void asignarNuevaReferencia(Persona persona) {
        persona = new Persona("Carlos");
        System.out.println("Nombre dentro del método: " + persona.nombre);
    }
    public static void main(String[] args) {
        Persona persona1 = new Persona("Ana");
        System.out.println("Nombre antes del método: " + persona1.nombre);
        asignarNuevaReferencia(persona1);
        System.out.println("Nombre después del método: " + persona1.nombre);
    }
}
```

Salida:

```
Nombre antes del método: Ana
Nombre dentro del método: Carlos
Nombre después del método: Ana
```

Explicación:

1. Dentro del método `asignarNuevaReferencia`, creamos una nueva referencia.

2. La referencia original (`persona1`) no se ve afectada porque solo se modificó la copia de la referencia.

2. Diferencias Clave: Tipos Primitivos vs Objetos

Aspecto	Tipos Primitivos	Objetos
Qué se pasa al método	Copia del valor	Copia de la referencia al objeto
Modificación en el método	No afecta al valor original	Modificar el estado interno afecta al original
Asignación de nueva referencia	No afecta fuera del método	No afecta la referencia original

3. Ejercicio Práctico

Problema

1. Crea una clase `Cuenta` con:

- Un atributo `saldo`.
- Un método `depositar` que aumente el saldo.
- Un método `retirar` que reduzca el saldo.

2. Escribe un método que acepte un objeto `Cuenta` como parámetro y realice un depósito y un retiro.

3. Comprueba cómo afecta al objeto original.

4. Preguntas de Evaluación

1. ¿Qué significa "paso por valor" en Java?

a) Se pasa una copia del valor o referencia.
b) Se pasa el valor original directamente.
c) Se pasa una copia completa del objeto.
d) Se crea un nuevo objeto dentro del método.

2. ¿Qué sucede si modificamos el estado interno de un objeto pasado como parámetro?

a) El objeto original se modifica.
b) No se modifica el objeto original.
c) Se crea un nuevo objeto.
d) Se lanza un error.

3. ¿Qué ocurre si reasignamos una referencia de objeto dentro de un método?

a) El objeto original también se reasigna.
b) La referencia original permanece sin cambios.
c) El objeto original se destruye.
d) Se lanza una excepción.

4. ¿Qué tipo de datos no puede modificarse en un método?

a) Objetos.
b) Tipos primitivos.
c) Referencias.
d) Ninguno de los anteriores.

Parte 10: Utilización de clases y objetos

La **utilización de clases y objetos** es la aplicación práctica de la programación orientada a objetos (POO). Una **clase** es la plantilla que define atributos y comportamientos, mientras que un **objeto** es una instancia concreta de esa clase.

En este apartado, exploraremos cómo usar clases y objetos de manera práctica, paso a paso, con múltiples ejemplos y ejercicios.

1. Creación de Objetos

Un **objeto** se crea utilizando la palabra clave `new`, que asigna memoria para el objeto y llama a un constructor para inicializar sus atributos.

Sintaxis General

```
NombreClase nombreObjeto = new NombreClase();
```

- `NombreClase`: La clase a partir de la cual se crea el objeto.
- `nombreObjeto`: El nombre que se le asigna al objeto.
- `new`: Palabra clave que asigna memoria al objeto.
- `NombreClase()`: Llamada al constructor de la clase.

Ejemplo Básico

```
public class Coche {
    String marca;
    String modelo;
    public void mostrarInformacion() {
        System.out.println("Marca: " + marca + ", Modelo: " + modelo);
    }
}
public class Main {
    public static void main(String[] args) {
        // Crear un objeto de la clase Coche
        Coche coche1 = new Coche();
        // Asignar valores a los atributos
        coche1.marca = "Toyota";
        coche1.modelo = "Corolla";
        // Llamar a un método del objeto
        coche1.mostrarInformacion();
    }
}
```

Salida:

```
Marca: Toyota, Modelo: Corolla
```

2. Acceso a los Atributos y Métodos

Podemos acceder a los atributos y métodos de un objeto utilizando el **operador punto (`.`)**.

Sintaxis

```
nombreObjeto.atributo;
nombreObjeto.metodo();
```

Ejemplo Práctico

```
public class Persona {
    String nombre;
    int edad;
    // Método para asignar nombre y edad
    public void asignarDatos(String nombre, int edad) {
          this.nombre = nombre;
          this.edad = edad;
    }
    // Método para mostrar la información
    public void mostrarInformacion() {
          System.out.println("Nombre: " + nombre + ", Edad: " + edad);
    }
}
public class Main {
    public static void main(String[] args) {
          Persona persona1 = new Persona();
          persona1.asignarDatos("Juan", 30);
          persona1.mostrarInformacion();
    }
}
```

Salida:

```
Nombre: Juan, Edad: 30
```

3. Uso de Múltiples Objetos

Podemos crear múltiples objetos de la misma clase. Cada objeto tendrá su propia copia de los atributos de instancia.

Ejemplo: Creación de Múltiples Objetos

```
public class Libro {
    String titulo;
    String autor;
    public void mostrarInformacion() {
        System.out.println("Título: " + titulo + ", Autor: " + autor);
    }
}
public class Main {
    public static void main(String[] args) {
        // Crear el primer objeto
        Libro libro1 = new Libro();
        libro1.titulo = "1984";
        libro1.autor = "George Orwell";
        // Crear el segundo objeto
        Libro libro2 = new Libro();
        libro2.titulo = "Cien años de soledad";
        libro2.autor = "Gabriel García Márquez";
        // Mostrar la información de ambos libros
        libro1.mostrarInformacion();
        libro2.mostrarInformacion();
    }
}
```

Salida:

```
Título: 1984, Autor: George Orwell
Título: Cien años de soledad, Autor: Gabriel García Márquez
```

4. Relación entre Clases

Las clases en Java pueden interactuar entre sí. Una clase puede contener referencias a objetos de otra clase, lo que facilita la creación de programas más complejos.

Ejemplo: Relación entre Clases

```
public class Motor {
    String tipo;
    public Motor(String tipo) {
        this.tipo = tipo;
    }
    public void mostrarTipo() {
        System.out.println("Tipo de motor: " + tipo);
    }
}
public class Coche {
    String marca;
    Motor motor;
    public Coche(String marca, Motor motor) {
        this.marca = marca;
        this.motor = motor;
    }
    public void mostrarInformacion() {
        System.out.println("Marca del coche: " + marca);
        motor.mostrarTipo();
    }
}
public class Main {
    public static void main(String[] args) {
        Motor motor1 = new Motor("V8");
        Coche coche1 = new Coche("Ford", motor1);
        coche1.mostrarInformacion();
    }
}
```

Salida:

```
Marca del coche: Ford
Tipo de motor: V8
```

5. Ejercicio Práctico

Problema

Crea una clase `Estudiante` con:

1. Atributos: `nombre`, `edad` **y** `notaMedia`**.**

2. Métodos:

- `asignarDatos`: Recibe valores para los atributos.
- `mostrarInformacion`: Muestra los datos del estudiante.

3. Crea dos objetos de la clase y muestra su información.

6. Preguntas de Evaluación

1. ¿Qué palabra clave se usa para crear un objeto en Java?

a) `new`
b) `create`
c) `instance`
d) `object`

2. ¿Cómo accedemos a un atributo de un objeto?

a) `objeto:atributo`
b) `objeto.atributo`
c) `atributo(objeto)`
d) Ninguna de las anteriores.

3. ¿Cuál de las siguientes afirmaciones es correcta?

a) Una clase puede tener solo un objeto.
b) Los objetos no pueden compartir métodos.
c) Una clase puede tener múltiples objetos.
d) Los métodos no pueden acceder a los atributos.

4. ¿Qué ocurre si creamos múltiples objetos de una clase?

a) Comparten la misma memoria.
b) Cada objeto tiene su propia copia de atributos de instancia.
c) Los objetos sobrescriben los atributos del otro.
d) No es posible crear múltiples objetos.

Parte 11: Utilización de clases heredadas

La **herencia** es uno de los conceptos fundamentales de la **Programación Orientada a Objetos (POO)**. En Java, la herencia permite que una clase (subclase) herede los atributos y métodos de otra clase (superclase). Esto fomenta la **reutilización del código** y facilita la extensión de funcionalidades sin necesidad de duplicar código.

1. Concepto de Herencia

- La **superclase** (o clase base) es la clase que proporciona atributos y métodos.
- La **subclase** (o clase derivada) es la clase que hereda de la superclase.
- La palabra clave `extends` se utiliza en Java para establecer una relación de herencia.

Sintaxis General

```
class SuperClase {
    // Atributos y métodos de la superclase
}
class SubClase extends SuperClase {
    // Atributos y métodos adicionales de la subclase
}
```

2. Ejemplo Básico de Herencia

Vamos a modelar la relación entre una clase `Persona` (superclase) y una clase `Estudiante` (subclase):

Código

```
// Superclase
public class Persona {
    String nombre;
    int edad;
    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
    public void mostrarInformacion() {
        System.out.println("Nombre: " + nombre + ", Edad: " + edad);
    }
}
// Subclase que hereda de Persona
public class Estudiante extends Persona {
    String curso;
    public Estudiante(String nombre, int edad, String curso) {
        super(nombre, edad); // Llamada al constructor de la superclase
        this.curso = curso;
    }
    // Método adicional en la subclase
    public void mostrarCurso() {
        System.out.println("Curso: " + curso);
    }
}
// Clase Principal
public class Main {
    public static void main(String[] args) {
        Estudiante estudiante1 = new Estudiante("Juan", 20, "Java Básico");
        // Métodos heredados de la clase Persona
        estudiante1.mostrarInformacion();
        // Método específico de la clase Estudiante
        estudiante1.mostrarCurso();
    }
}
```

Salida

```
Nombre: Juan, Edad: 20
Curso: Java Básico
```

3. La Palabra Clave `super`

La palabra clave `super` se utiliza para:

1. Acceder a atributos y métodos de la superclase que han sido ocultados por la subclase.

2. Llamar al constructor de la superclase.

Ejemplo de Uso de `super`

```
public class Animal {
    String nombre;
    public Animal(String nombre) {
        this.nombre = nombre;
    }
    public void dormir() {
        System.out.println(nombre + " está durmiendo.");
    }
}
public class Perro extends Animal {
    public Perro(String nombre) {
        super(nombre); // Llamada al constructor de la superclase
    }
    public void ladrar() {
        System.out.println(nombre + " está ladrando.");
    }
    public void dormirComoAnimal() {
        super.dormir(); // Llama al método de la superclase
    }
}
public class Main {
    public static void main(String[] args) {
        Perro perro1 = new Perro("Bobby");
        perro1.dormir(); // Heredado
        perro1.ladrar(); // Método propio
        perro1.dormirComoAnimal(); // Llamada explícita a super.dormir()
    }
}
```

Salida

```
Bobby está durmiendo.
Bobby está ladrando.
Bobby está durmiendo.
```

4. Herencia y Sobrescritura de Métodos

La **sobrescritura** de métodos (method overriding) permite que una subclase redefina un método heredado de la superclase. Esto se logra utilizando la anotación `@Override`.

Ejemplo: Sobrescritura de Métodos

```
public class Animal {
    public void hacerSonido() {
          System.out.println("El animal hace un sonido.");
    }
}
public class Perro extends Animal {
    @Override
    public void hacerSonido() {
          System.out.println("El perro ladra.");
    }
}
public class Gato extends Animal {
    @Override
    public void hacerSonido() {
          System.out.println("El gato maúlla.");
    }
}
public class Main {
    public static void main(String[] args) {
          Animal animal1 = new Perro();
          Animal animal2 = new Gato();
          animal1.hacerSonido();
          animal2.hacerSonido();
    }
}
```

Salida

```
El perro ladra.
El gato maúlla.
```

5. Ejercicio Práctico

Problema

Crea una jerarquía de clases:

1. Una superclase llamada `Vehiculo` con atributos:

- `marca`, `modelo` y un método `mostrarInformacion`.

2. Una subclase llamada `Coche` que herede de `Vehiculo` y añada un atributo `velocidadMaxima`.

- Define un método para mostrar la velocidad máxima.

3. Crea una instancia de `Coche` y muestra toda su información.

6. Preguntas de Evaluación

1. ¿Qué palabra clave se utiliza para establecer la herencia en Java?

a) `extends`
b) `inherits`
c) `super`
d) `implements`

2. ¿Qué hace la palabra clave `super`?

a) Accede a métodos y atributos privados de una clase.
b) Llama al constructor de la superclase o accede a métodos/atributos heredados.
c) Declara una nueva subclase.
d) Define un método sobrecargado.

3. ¿Qué es sobrescritura de métodos?

a) Definir un método con el mismo nombre pero diferentes parámetros.
b) Reescribir un método heredado en una subclase.
c) Crear múltiples métodos con el mismo cuerpo.
d) Ocultar métodos de una clase.

4. ¿Cuál de las siguientes afirmaciones es verdadera?

a) Una subclase no puede acceder a métodos de la superclase.

b) La palabra clave `super` se usa para llamar al constructor de la superclase.

c) La herencia no permite reutilización del código.

d) Una subclase puede sobrescribir constructores.

Parte 12: Librerías y paquetes de clases. Utilización y creación

En Java, las **librerías** y **paquetes de clases** son herramientas fundamentales para organizar, reutilizar y modularizar el código. Java proporciona un conjunto extenso de librerías estándar a través del **Java Standard Library**, y además permite a los desarrolladores crear y utilizar sus propios paquetes.

1. Concepto de Librerías y Paquetes

1. **Librería: Una librería es un conjunto de clases y métodos predefinidos que facilitan el desarrollo de software al ofrecer funcionalidades reutilizables.**

   Ejemplo: La **librería estándar de Java** incluye paquetes como `java.util`, `java.io`, y `java.lang`.

2. **Paquete (Package): Es una agrupación lógica de clases e interfaces relacionadas.**

   - Facilita la **organización** del código.
   - Evita **conflictos de nombres** entre clases.
   - Permite **encapsular** funcionalidades específicas.

2. Utilización de Paquetes Predefinidos

Los paquetes predefinidos de Java contienen clases y métodos listos para usar. Algunos ejemplos importantes incluyen:

Paquete	Descripción
`java.lang`	Clases fundamentales como `String`, `Math`
`java.util`	Clases de utilidades como `ArrayList`, `HashMap`.
`java.io`	Manejo de archivos y flujos de datos.
`java.sql`	Manejo de bases de datos con JDBC.
`java.time`	Manejo de fechas y horas.

Ejemplo: Uso de `java.util.ArrayList`

```
import java.util.ArrayList; // Importar la clase ArrayList
public class EjemploArrayList {
    public static void main(String[] args) {
          ArrayList<String> lista = new ArrayList<>();
          // Añadir elementos a la lista
          lista.add("Manzana");
          lista.add("Banana");
          lista.add("Cereza");
          // Recorrer e imprimir la lista
          for (String fruta : lista) {
                System.out.println(fruta);
          }
    }
}
```

Salida:

```
Manzana
Banana
Cereza
```

3. Creación de Paquetes Propios

Los desarrolladores pueden crear sus propios paquetes para organizar sus clases.

Pasos para Crear un Paquete

1. Declara el paquete al inicio del archivo utilizando la palabra clave `package`.
2. Guarda el archivo en un directorio que coincida con el nombre del paquete.
3. Importa el paquete en otras clases utilizando `import`.

Ejemplo: Creación de un Paquete Propio

Paso 1: Crear el Paquete

Creamos una clase `Persona` en el paquete `mipaquete`:

```
// Archivo: Persona.java
package mipaquete;
public class Persona {
    private String nombre;
    public Persona(String nombre) {
        this.nombre = nombre;
    }
    public void mostrarNombre() {
        System.out.println("Nombre: " + nombre);
    }
}
```

Paso 2: Utilizar el Paquete en Otra Clase

Creamos una clase `Main` que utiliza el paquete `mipaquete`:

```
// Archivo: Main.java
import mipaquete.Persona; // Importar la clase Persona del paquete mipaquete
public class Main {
    public static void main(String[] args) {
        // Crear un objeto de la clase Persona
        Persona persona1 = new Persona("Juan");
        persona1.mostrarNombre();
    }
}
```

Estructura de Directorios

La estructura de archivos debe coincidir con el nombre del paquete:

```
/mipaquete/
    Persona.java
Main.java
```

Compilar y Ejecutar

1. Compila las clases con:

```
javac mipaquete/Persona.java Main.java
```

2. Ejecuta la clase `Main`:

```
java Main
```

Salida:

```
Nombre: Juan
```

4. Importar Paquetes

Para importar paquetes en Java, existen dos formas:

1. Importar una clase específica:

```
import mipaquete.Persona;
```

2. Importar todas las clases de un paquete:

```
import mipaquete.*;
```

5. Paquetes y Subpaquetes

Java permite crear **subpaquetes** dentro de un paquete principal.

Ejemplo: Creación de un Subpaquete

1. Clase en el subpaquete:

```
package mipaquete.subpaquete;
public class Estudiante {
    public void mostrarInformacion() {
          System.out.println("Clase en un subpaquete.");
    }
}
```

1. Clase Principal:

```
import mipaquete.subpaquete.Estudiante;
public class Main {
    public static void main(String[] args) {
          Estudiante estudiante = new Estudiante();
          estudiante.mostrarInformacion();
    }
}
```

Estructura de Directorios:

```
/mipaquete/subpaquete/
    Estudiante.java
Main.java
```

6. Ejercicio Práctico

Problema

1. Crea un paquete `empresa`.
2. Dentro del paquete, crea una clase `Empleado` con los atributos `nombre` y `salario`.
3. Define métodos para asignar y mostrar la información del empleado.
4. Crea una clase principal `Main` que importe el paquete `empresa` y utilice la clase `Empleado`.

7. Preguntas de Evaluación

1. ¿Qué palabra clave se utiliza para declarar un paquete en Java?

a) `package`
b) `import`
c) `namespace`
d) `class`

2. ¿Cómo se importan todas las clases de un paquete?

a) `import paquete.*;`
b) `include paquete.*;`
c) `import paquete.all;`
d) `package.import *;`

3. ¿Qué estructura de directorios debe seguir un paquete en Java?

a) Debe coincidir con el nombre del paquete.
b) Debe ser una carpeta aleatoria.
c) No necesita una estructura específica.
d) Debe ser una subcarpeta de `java`.

4. ¿Cuál de las siguientes es una ventaja del uso de paquetes?

a) Evitar conflictos de nombres.
b) No se puede reutilizar el código.
c) Los paquetes no son organizados.
d) Complica el desarrollo de software.

Parte 13: Documentación sobre librerías y paquetes de clases

La **documentación** es un componente clave en el desarrollo de software. Java proporciona herramientas y convenciones para documentar librerías, paquetes y clases, facilitando la comprensión, el uso y la reutilización del código. La herramienta principal en Java para generar documentación es **Javadoc**.

1. ¿Qué es Javadoc?

Javadoc es una herramienta integrada en el JDK que permite generar automáticamente **documentación HTML** a partir de los comentarios especiales escritos en el código fuente de Java.

- Javadoc facilita la creación de manuales técnicos y de usuario.
- La documentación generada describe:
  - **Clases**
  - **Métodos**
  - **Atributos**
  - **Paquetes**
  - **Constructores**

2. Comentarios de Documentación en Java

Para que Javadoc pueda generar documentación, es necesario usar **comentarios de documentación** especiales:

Sintaxis

```
/**
 * Descripción de la clase, método o atributo.
 * @param nombreParámetro Descripción del parámetro.
 * @return Descripción del valor de retorno.
 * @author Nombre del autor.
 * @version Número de la versión.
 */
```

3. Etiquetas Comunes de Javadoc

Etiqueta	Descripción
`@author`	Nombre del autor del código.
`@version`	Número de versión del código.
`@param`	Describe los parámetros de un método.
`@return`	Describe el valor que devuelve el método.
`@see`	Proporciona una referencia a otra clase o método relacionado.
`@deprecated`	Marca un método o clase como obsoleta.
`@throws` o `@exception`	Describe las excepciones que puede lanzar un método.

4. Ejemplo de Comentarios de Documentación

```
java
Copiar código
/**
 * La clase Calculadora realiza operaciones matemáticas básicas.
 * @author Juan
 * @version 1.0
 */
public class Calculadora {
    /**
     * Suma dos números enteros.
     * @param a El primer número a sumar.
     * @param b El segundo número a sumar.
     * @return La suma de los dos números.
     */
    public int sumar(int a, int b) {
        return a + b;
    }
    /**
     * Resta dos números enteros.
     * @param a El número del cual se resta.
     * @param b El número que se resta.
     * @return El resultado de la resta.
     */
    public int restar(int a, int b) {
        return a - b;
    }
```

```
    /**
     * Multiplica dos números enteros.
     * @param a El primer número.
     * @param b El segundo número.
     * @return El producto de los dos números.
     */
    public int multiplicar(int a, int b) {
        return a * b;
    }
    /**
     * Divide dos números enteros.
     * @param a El numerador.
     * @param b El denominador. No puede ser cero.
     * @return El resultado de la división.
     * @throws ArithmeticException Si el denominador es cero.
     */
    public int dividir(int a, int b) throws ArithmeticException {
        if (b == 0) {
            throw new ArithmeticException("No se puede dividir por
cero.");
        }
        return a / b;
    }
}
```

5. Generar la Documentación con Javadoc

Para generar la documentación HTML a partir del código fuente:

1. Escribe comentarios Javadoc en tus clases, métodos y atributos.
2. Guarda el archivo como `.java`.
3. Compila y ejecuta el comando `javadoc` en la terminal.

Ejemplo: Generar Documentación

Supón que tenemos el archivo `Calculadora.java`:

1. Ejecuta el siguiente comando desde la terminal:

```
javadoc -d docs Calculadora.java
```

- `d docs`: Indica la carpeta donde se guardará la documentación generada.
- `Calculadora.java`: Archivo fuente que contiene los comentarios Javadoc.

1. Abre el archivo `index.html` en la carpeta `docs` para ver la documentación en el navegador.

6. Documentación de Paquetes

Para documentar paquetes, puedes incluir un archivo especial llamado `package-info.java`. Este archivo proporciona información general sobre el paquete.

Ejemplo: Documentar un Paquete

1. Crear un archivo `package-info.java`

```java
/**
 * Este paquete contiene clases relacionadas con operaciones matemáticas.
 * Proporciona una calculadora básica para realizar sumas, restas,
 * multiplicaciones y divisiones.
 */
package mipaquete;
```

2. Estructura del Paquete

```
/mipaquete/
    Calculadora.java
    package-info.java
```

3. Generar Documentación del Paquete

Ejecuta el comando:

```
javadoc -d docs mipaquete/*.java
```

- Esto genera documentación para todas las clases dentro del paquete `mipaquete`.

7. Ejercicio Práctico

Problema

1. Crea un paquete llamado `biblioteca`.

2. Dentro del paquete, crea una clase `Libro` con atributos:

- `titulo`, `autor`, `numPaginas`.

3. Documenta la clase con Javadoc:

- Añade descripciones generales y etiquetas `@param` y `@return`.

4. Genera la documentación utilizando el comando `javadoc`.

8. Preguntas de Evaluación

1. ¿Qué herramienta se utiliza en Java para generar documentación?

a) `Javadoc`
b) `DocGen`
c) `JavaDocs`
d) `JavaComment`

2. ¿Qué etiqueta se utiliza para documentar los parámetros de un método?

a) `@param`
b) `@args`
c) `@method`
d) `@return`

3. ¿Qué comando se usa para generar la documentación Javadoc?

a) `java -doc`
b) `javadoc`
c) `javac -docs`
d) `docgen`

4. ¿Dónde se almacena la documentación generada por Javadoc?

a) En el archivo fuente `.java`.
b) En la carpeta especificada con `d`.
c) En una base de datos.
d) Dentro de la JVM.

5. ¿Qué archivo se utiliza para documentar un paquete?

a) `package.xml`
b) `package-info.java`
c) `README.md`
d) `doc-package.txt`

Conclusión del Bloque: Desarrollo de clases

En este bloque hemos profundizado en el **desarrollo de clases**, que constituye el corazón de la **Programación Orientada a Objetos (POO)** en Java. Comprender cómo se crean y utilizan las clases es esencial para construir aplicaciones modulares, reutilizables y eficientes. A lo largo de este bloque, hemos cubierto conceptos fundamentales, con ejemplos detallados y ejercicios prácticos, facilitando la asimilación de estos principios.

1. Resumen de los Puntos Clave

1. Concepto de Clase

Una **clase** es una plantilla que define los **atributos** (estado) y **métodos** (comportamiento) que tendrán los objetos.

- Ejemplo: La clase `Persona` con atributos `nombre` y `edad` y métodos como `saludar()`.

2. Estructura y Miembros de una Clase

Una clase en Java puede incluir:

- **Atributos**: Variables que representan el estado.
- **Métodos**: Bloques de código que representan comportamientos.
- **Constructores**: Métodos especiales para inicializar objetos.
- **Clases anidadas**: Clases dentro de otras clases.

3. Creación de Atributos

Los atributos definen el estado de los objetos y pueden ser:

- **De instancia**: Pertenecen a cada objeto.
- **Estáticos**: Compartidos entre todos los objetos de la clase.

4. Creación de Métodos

Los métodos permiten definir acciones. Los conceptos clave incluyen:

- Métodos **con retorno** y **sin retorno** (`void`).
- Métodos con **parámetros** y su sobrecarga.
- Uso de la palabra clave `this` para diferenciar atributos y parámetros.

5. Creación de Constructores

Los **constructores** son esenciales para inicializar objetos:

- **Constructores por defecto**: Sin parámetros.
- **Constructores parametrizados**: Permiten inicializar atributos al crear el objeto.
- **Sobrecarga de constructores**: Múltiples constructores con diferentes parámetros.

6. Ámbito de Atributos y Variables

El **ámbito** determina dónde y cómo se puede acceder a una variable:

- **Variables locales**: Declaradas dentro de un método o bloque.
- **Variables de instancia**: Atributos de un objeto.
- **Variables estáticas**: Compartidas por todos los objetos.

7. Sobrecarga de Métodos

La **sobrecarga** permite definir múltiples métodos con el mismo nombre, pero con diferentes parámetros, lo que facilita la flexibilidad y adaptabilidad del código.

8. Visibilidad y Encapsulación

Mediante **modificadores de acceso** (`public`, `private`, `protected`), podemos:

- Proteger los datos de la clase.
- Controlar el acceso a métodos y atributos.
- Implementar **encapsulación**, accediendo a atributos privados a través de métodos setters y getters.

9. Paso de Parámetros

- **Tipos primitivos**: Se pasan por valor, enviando una copia.
- **Objetos**: Se pasa una copia de la referencia, permitiendo modificar el estado del objeto.

10. Utilización de Clases y Objetos

La creación de objetos a partir de clases permite modelar entidades del mundo real:

- Uso del operador `new` para instanciar objetos.
- Acceso a atributos y métodos mediante el **operador punto**.

11. Utilización de Clases Heredadas

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

- Uso de la palabra clave `extends`.
- Sobrescritura de métodos con `@Override`.
- Uso de `super` para acceder a constructores o métodos de la superclase.

12. Librerías y Paquetes de Clases

La organización del código mediante **paquetes** permite modularizar y reutilizar clases. Además:

- Uso de **paquetes predefinidos** como `java.util`.
- Creación de **paquetes personalizados**.
- Generación de documentación con **Javadoc**.

2. Importancia del Desarrollo de Clases

1. **Modularidad:** El uso de clases permite dividir el código en componentes más pequeños y manejables.
2. **Reutilización:** Clases bien definidas pueden reutilizarse en diferentes partes de un programa o incluso en otros proyectos.
3. **Encapsulación:** Protege los datos y permite un acceso controlado mediante métodos específicos.
4. **Extensibilidad:** La herencia permite extender el comportamiento de las clases existentes sin duplicar código.
5. **Organización:** Los paquetes facilitan la organización lógica del proyecto, especialmente en aplicaciones grandes.

3. Ejercicio Final del Bloque

Problema: Sistema de Gestión de Empleados

1. **Crea una superclase llamada `Persona` con los siguientes atributos:**
   - `nombre` (String)
   - `edad` (int)
   - Método `mostrarInformacion` para imprimir los atributos.
2. **Crea una subclase llamada `Empleado` que herede de `Persona` y tenga:**
   - Atributo `salario` (double).
   - Método `mostrarInformacion` (sobrescribe el de la superclase e incluye el salario).
3. **Crea una clase `Main` que:**
   - Cree varios objetos de `Empleado` y `Persona`.
   - Pruebe los métodos para mostrar información.

Solución

```
// Superclase Persona
public class Persona {
    protected String nombre;
    protected int edad;
    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
    public void mostrarInformacion() {
        System.out.println("Nombre: " + nombre + ", Edad: " + edad);
    }
}
// Subclase Empleado
public class Empleado extends Persona {
    private double salario;
    public Empleado(String nombre, int edad, double salario) {
        super(nombre, edad); // Llama al constructor de Persona
        this.salario = salario;
    }
    @Override
    public void mostrarInformacion() {
        super.mostrarInformacion();
        System.out.println("Salario: " + salario);
    }
}
// Clase Principal
public class Main {
    public static void main(String[] args) {
        // Crear objetos de Persona
        Persona persona1 = new Persona("Ana", 30);
        // Crear objetos de Empleado
        Empleado empleado1 = new Empleado("Juan", 25, 2500.50);
        Empleado empleado2 = new Empleado("Laura", 28, 3000.75);
        // Mostrar información
        System.out.println("Información de Persona:");
        persona1.mostrarInformacion();
        System.out.println("\nInformación de Empleados:");
        empleado1.mostrarInformacion();
        empleado2.mostrarInformacion();
    }
}
```

Salida Esperada

```
Información de Persona:
Nombre: Ana, Edad: 30
Información de Empleados:
Nombre: Juan, Edad: 25
Salario: 2500.5
Nombre: Laura, Edad: 28
Salario: 3000.75
```

4. Preguntas de Evaluación

1. ¿Qué es una clase en Java?

a) Una plantilla para objetos.
b) Un método.
c) Un valor primitivo.
d) Una librería.

2. ¿Cuál es la palabra clave para heredar de una clase?

a) `super`
b) `extends`
c) `implements`
d) `inherit`

3. ¿Qué permite la sobrecarga de métodos?

a) Ocultar métodos.
b) Definir métodos con el mismo nombre pero parámetros diferentes.
c) Reescribir métodos en una subclase.
d) Ninguna de las anteriores.

4. ¿Qué herramienta se utiliza para generar documentación en Java?

a) Javadoc
b) JavaDocs
c) DocGenerator
d) CompilerDoc