Unidad 10

Utilización avanzada de clases

Contenidos

Parte 1: Relaciones entre clases. Composición de clases

1. Introducción a las Relaciones entre Clases

En la **Programación Orientada a Objetos (POO)**, las clases no son entidades aisladas; interactúan entre sí para modelar sistemas reales. Estas interacciones se denominan **relaciones entre clases** y pueden clasificarse en:

1. **Asociación: Una clase utiliza o conoce a otra clase.**
2. **Composición: Una clase está formada por una o más instancias de otras clases.**
3. **Agregación: Similar a la composición, pero las partes pueden existir independientemente.**
4. **Herencia: Una clase hereda atributos y métodos de otra clase.**

1.1. Asociación

La asociación representa una relación entre dos clases donde una clase utiliza a otra.

Ejemplo de Asociación

```
public class Profesor {
    private String nombre;
    public Profesor(String nombre) {
        this.nombre = nombre;
    }
    public String getNombre() {
        return nombre;
    }
}
public class Curso {
    private String nombreCurso;
    private Profesor profesor;
    public Curso(String nombreCurso, Profesor profesor) {
        this.nombreCurso = nombreCurso;
        this.profesor = profesor;
    }
    public void mostrarInformacion() {
        System.out.println("Curso: " + nombreCurso);
        System.out.println("Profesor: " + profesor.getNombre());
    }
}
public class Main {
    public static void main(String[] args) {
        Profesor profesor1 = new Profesor("Juan Pérez");
        Curso curso1 = new Curso("Matemáticas", profesor1);
        curso1.mostrarInformacion();
    }
}
```

Salida:

```
Curso: Matemáticas
Profesor: Juan Pérez
```

1.2. Composición

La composición es una relación **"tiene un"** en la que una clase contiene una o más instancias de otra clase como parte de su estado interno. Si el objeto "contenedor" se destruye, también lo hacen las partes.

Ejemplo de Composición

```
public class Motor {
    private String tipo;
    public Motor(String tipo) {
        this.tipo = tipo;
    }
    public String getTipo() {
        return tipo;
    }
}
public class Coche {
    private String marca;
    private Motor motor;
    public Coche(String marca, Motor motor) {
        this.marca = marca;
        this.motor = motor;
    }
    public void mostrarInformacion() {
        System.out.println("Marca: " + marca);
        System.out.println("Motor: " + motor.getTipo());
    }
}
public class Main {
    public static void main(String[] args) {
        Motor motor1 = new Motor("Eléctrico");
        Coche coche1 = new Coche("Tesla", motor1);
        coche1.mostrarInformacion();
    }
}
```

Salida:

```
Marca: Tesla
Motor: Eléctrico
```

1.3. Agregación

La agregación también es una relación “tiene un”, pero las partes pueden existir de forma independiente al todo.

Ejemplo de Agregación

```
public class Autor {
    private String nombre;
    public Autor(String nombre) {
          this.nombre = nombre;
    }
    public String getNombre() {
          return nombre;
    }
}
public class Libro {
    private String titulo;
    private Autor autor;
    public Libro(String titulo, Autor autor) {
          this.titulo = titulo;
          this.autor = autor;
    }
    public void mostrarInformacion() {
          System.out.println(“Título: “ + titulo);
          System.out.println(“Autor: “ + autor.getNombre());
    }
}
public class Main {
    public static void main(String[] args) {
          Autor autor1 = new Autor(“Gabriel García Márquez”);
          Libro libro1 = new Libro(“Cien años de soledad”, autor1);
          libro1.mostrarInformacion();
    }
}
```

Salida:

```
Título: Cien años de soledad
Autor: Gabriel García Márquez
```

1.4. Diferencias entre Composición y Agregación

Aspecto	Composición	Agregación
Dependencia	Las partes dependen del todo.	Las partes pueden existir independientemente.
Ejemplo	Motor en un coche.	Autor de un libro.
Representación UML	Línea con rombo lleno.	Línea con rombo vacío.

1.5. Ventajas de las Relaciones entre Clases

1. **Reutilización del Código: Permite utilizar componentes existentes en nuevas clases.**
2. **Modularidad: Cada clase tiene una responsabilidad clara y única.**
3. **Flexibilidad: Facilita la implementación de sistemas complejos.**

2. Ejercicio Guiado

Problema

1. **Crea una clase `Direccion` con los atributos `calle`, `ciudad` y `código postal`.**
2. **Crea una clase `Persona` que tenga un atributo `nombre` y una composición con la clase `Direccion`.**
3. **Escribe un método en `Persona` que muestre la información completa de la persona, incluyendo su dirección.**

3. Preguntas de Evaluación

1. ¿Qué tipo de relación representa una composición?

a) “Es un”.
b) “Tiene un”.
c) “Puede ser”.
d) “Usa un”.

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

a) En composición, las partes son independientes del todo.
b) En agregación, las partes dependen del todo.
c) En composición, las partes no existen sin el todo.
d) Ninguna de las anteriores.

3. ¿Qué palabra clave se utiliza para crear una relación entre clases en Java?

a) `class`.
b) `new`.
c) Ninguna, las relaciones se crean mediante atributos.
d) `extend`.

Parte 2: Herencia. Concepto y tipos (simple y múltiple).

La **herencia** es uno de los pilares fundamentales de la **Programación Orientada a Objetos (POO)**. Permite que una clase (subclase o clase hija) herede atributos y métodos de otra clase (superclase o clase padre), promoviendo la **reutilización del código** y la **extensibilidad**.

1. Concepto de Herencia

1. Definición:

La herencia establece una relación **"es un"** entre una clase base (superclase) y una clase derivada (subclase).

- La subclase **hereda** los atributos y métodos de la superclase.
- Puede añadir nuevos métodos y atributos propios.
- Puede **sobrescribir** métodos heredados para cambiar su comportamiento.

2. Palabra Clave:

En Java, se utiliza la palabra clave `extends` para implementar herencia.

Ejemplo de Herencia

```
public class Animal {
    public void comer() {
        System.out.println("Este animal está comiendo.");
    }
}
public class Perro extends Animal {
    public void ladrar() {
        System.out.println("El perro está ladrando.");
    }
}
public class Main {
    public static void main(String[] args) {
        Perro miPerro = new Perro();
        // Métodos heredados
        miPerro.comer();
        // Métodos propios
        miPerro.ladrar();
    }
}
```

Salida:

```
Este animal está comiendo.
El perro está ladrando.
```

2. Tipos de Herencia

2.1. Herencia Simple

En la **herencia simple**, una clase hereda de una sola clase base. Java admite este tipo de herencia de forma nativa.

Ejemplo de Herencia Simple

```
public class Vehiculo {
    public void mover() {
          System.out.println("El vehículo se está moviendo.");
    }
}
public class Coche extends Vehiculo {
    public void encenderLuces() {
          System.out.println("Las luces del coche están encendidas.");
    }
}
public class Main {
    public static void main(String[] args) {
          Coche miCoche = new Coche();
          miCoche.mover();        // Método heredado
          miCoche.encenderLuces();  // Método propio
    }
}
```

Salida:

```
El vehículo se está moviendo.
Las luces del coche están encendidas.
```

2.2. Herencia Múltiple

Java **no admite herencia múltiple directa** para evitar ambigüedades (problema del diamante). Sin embargo, es posible implementarla de manera indirecta mediante el uso de **interfaces**.

Ejemplo de Herencia Múltiple con Interfaces

```
interface Volador {
    void volar();
}
interface Nadador {
    void nadar();
}
public class Pato implements Volador, Nadador {
    public void volar() {
          System.out.println("El pato está volando.");
    }
    public void nadar() {
          System.out.println("El pato está nadando.");
    }
}
public class Main {
    public static void main(String[] args) {
          Pato miPato = new Pato();
          miPato.volar();
          miPato.nadar();
    }
}
```

Salida:

```
El pato está volando.
El pato está nadando.
```

3. Ventajas y Desventajas de la Herencia

Ventajas	Desventajas
Reutilización del Código: Las subclases pueden aprovechar el código de la superclase.	**Acoplamiento**: Cambios en la superclase pueden afectar a las subclases.
Extensibilidad: Se pueden añadir funcionalidades a través de nuevas subclases.	**Rigidez**: Puede ser difícil modificar jerarquías complejas.
Facilidad de Mantenimiento: El código común se centraliza en la superclase.	**Sobrecarga**: Una jerarquía muy profunda puede complicar la comprensión.

4. Consideraciones en la Herencia

1. Acceso a Miembros de la Superclase:

- Los miembros con acceso `public` y `protected` son accesibles en la subclase.
- Los miembros `private` no son accesibles directamente, pero pueden ser accedidos mediante métodos de la superclase.

2. Jerarquías de Herencia:

- Se recomienda mantener jerarquías simples y claras.
- Una jerarquía profunda puede dificultar la comprensión y mantenimiento del código.

5. Ejercicio Guiado

Problema

1. Crea una superclase `Empleado` con atributos:

- `nombre` (String), `edad` (int) y `salario` (double).
- Método `mostrarInformacion()` que imprima todos los atributos.

2. Crea una subclase `Gerente` que herede de `Empleado`:

- Añade el atributo `departamento` (String).
- Sobrescribe el método `mostrarInformacion()` para incluir el departamento.

3. En la clase `Main`, crea objetos de ambas clases y prueba sus métodos.

6. Preguntas de Evaluación

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

a) `extends`
b) `implements`
c) `inherits`
d) Ninguna de las anteriores

2. ¿Qué diferencia existe entre herencia simple y múltiple?

a) La herencia simple permite heredar de varias clases.
b) La herencia múltiple es posible en Java directamente.
c) La herencia múltiple usa interfaces para evitar ambigüedades.
d) La herencia simple no existe en Java.

3. ¿Por qué Java no admite herencia múltiple directa?

a) Para mejorar la legibilidad del código.
b) Para evitar el problema del diamante.
c) Porque es innecesaria en Java.
d) Porque no se permite heredar de interfaces.

4. ¿Qué tipo de miembros de una superclase no son accesibles directamente en una subclase?

a) `public`
b) `protected`
c) `private`
d) Todos son accesibles.

Parte 3: Superclases y subclases.

En el contexto de la **Programación Orientada a Objetos (POO)**, las **superclases** y **subclases** son conceptos fundamentales que reflejan las relaciones jerárquicas entre clases.

1. Concepto de Superclase

La **superclase** (también conocida como **clase base** o **clase padre**) es una clase que provee atributos y métodos que pueden ser heredados por otras clases.

1. Definición:

- Una superclase contiene la funcionalidad común que será compartida con sus subclases.
- Se define como cualquier clase que es heredada por otra clase.

2. Características:

- Puede tener métodos y atributos comunes.
- Puede definir métodos que las subclases deben sobrescribir.
- Se utiliza para evitar duplicación de código.

Ejemplo de Superclase

```
java
Copiar código
public class Animal {
    protected String nombre;
    public Animal(String nombre) {
          this.nombre = nombre;
    }
    public void comer() {
          System.out.println(nombre + " está comiendo.");
    }
}
```

2. Concepto de Subclase

La **subclase** (también conocida como **clase derivada** o **clase hija**) es una clase que hereda atributos y métodos de la superclase.

1. Definición:

- Una subclase puede extender la funcionalidad de la superclase.

- Puede sobrescribir métodos de la superclase para cambiar su comportamiento.

2. Características:

- Puede añadir atributos y métodos propios.
- Utiliza la palabra clave `extends` para establecer la relación de herencia.

Ejemplo de Subclase

```
public class Perro extends Animal {
    public Perro(String nombre) {
        super(nombre); // Llama al constructor de la superclase
    }
    public void ladrar() {
        System.out.println(nombre + " está ladrando.");
    }
}
```

3. Relación entre Superclases y Subclases

La relación entre una superclase y una subclase se describe como una relación **"es un"** (is-a relationship).

1. Subclase es un tipo especializado de la superclase.

Ejemplo: Un **Perro** es un **Animal**.

2. La subclase hereda las propiedades y métodos de la superclase, pero también puede:

- Definir nuevos métodos y atributos.
- Sobrescribir métodos de la superclase.

4. Uso de `super` en Subclases

La palabra clave `super` se utiliza en subclases para:

1. Llamar al constructor de la superclase.

2. Acceder a métodos o atributos de la superclase.

Ejemplo: Uso de `super`

```
public class Animal {
    protected String nombre;
    public Animal(String nombre) {
        this.nombre = nombre;
    }
    public void dormir() {
        System.out.println(nombre + " está durmiendo.");
    }
}
public class Gato extends Animal {
    public Gato(String nombre) {
        super(nombre); // Llama al constructor de Animal
    }
    public void dormir() {
        super.dormir(); // Llama al método dormir de la superclase
        System.out.println(nombre + " está soñando.");
    }
}
public class Main {
    public static void main(String[] args) {
        Gato miGato = new Gato("Luna");
        miGato.dormir();
    }
}
```

Salida:

```
Luna está durmiendo.
Luna está soñando.
```

5. Sobrecarga y Sobrescritura

1. Sobrecarga de Métodos:

- Múltiples métodos con el mismo nombre pero diferentes parámetros en la misma clase o subclase.

2. Sobrescritura de Métodos:

- La subclase redefine un método de la superclase para cambiar su comportamiento.
- Usa la anotación `@Override`.

Ejemplo de Sobrescritura

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

Salida:

```
El perro ladra.
```

6. Ventajas de Superclases y Subclases

Ventajas	Descripción
Reutilización del Código	La subclase aprovecha el código de la superclase.
Extensibilidad	Es fácil añadir nuevas funcionalidades en subclases.
Centralización de Funcionalidades Comunes	Reduce duplicación y facilita el mantenimiento.

7. Ejercicio Guiado

Problema

1. Define una superclase `Vehiculo` con atributos:

- `marca` y `velocidadMaxima`.
- Método `mostrarInformacion()` para imprimir los atributos.

2. Define una subclase `Coche` que:

- Herede de `Vehiculo`.
- Añada un atributo `tipoCombustible`.
- Sobrescriba el método `mostrarInformacion()` para incluir el tipo de combustible.

3. En la clase principal, crea un objeto de la clase `Coche` e imprime su información.

8. Preguntas de Evaluación

1. ¿Qué relación describe la herencia entre una superclase y una subclase?

a) “Tiene un”.
b) “Es un”.
c) “Usa un”.
d) Ninguna de las anteriores.

2. ¿Para qué se utiliza la palabra clave `super`?

a) Llamar al constructor de la subclase.
b) Sobrescribir métodos.
c) Acceder al constructor o métodos de la superclase.
d) Ninguna de las anteriores.

3. ¿Qué permite la sobrescritura de métodos?

a) Reutilizar código sin cambiarlo.
b) Cambiar el comportamiento de un método heredado.
c) Crear múltiples versiones de un método.
d) Definir un método privado.

4. ¿Qué ocurre si una subclase no llama explícitamente al constructor de la superclase?

a) Lanza un error.
b) Se llama al constructor por defecto de la superclase.
c) No se inicializa el objeto.
d) Se ejecuta el método `toString()`.

Parte 4: Constructores y herencia.

Los **constructores** desempeñan un papel crucial en la inicialización de objetos en Java. Cuando se combina con **herencia**, los constructores permiten inicializar tanto los atributos de la **superclase** como los de la **subclase**, asegurando que los objetos derivados estén completamente preparados para su uso.

1. ¿Qué es un Constructor?

Un **constructor** es un método especial que:

1. Se ejecuta automáticamente cuando se crea un objeto.

2. Tiene el mismo nombre que la clase.

3. No tiene un valor de retorno, ni siquiera `void`.

4. Se utiliza para inicializar los atributos de una clase.

Tipos de Constructores

1. Constructor por Defecto:

- Generado automáticamente si no se declara un constructor en la clase.
- No recibe parámetros.

```
public class Persona {
    public Persona() {
          System.out.println(“Constructor por defecto invocado”);
    }
}
```

2. Constructor Parametrizado:

- Recibe parámetros para inicializar los atributos al momento de crear el objeto.

```
public class Persona {
    private String nombre;
    public Persona(String nombre) {
          this.nombre = nombre;
    }
}
```

2. Constructores en Herencia

Cuando una subclase hereda de una superclase:

1. El constructor de la superclase se invoca automáticamente antes de que se ejecute el constructor de la subclase.

2. La palabra clave `super` se utiliza para invocar explícitamente un constructor específico de la superclase.

Invocación Implícita del Constructor

Si no se llama explícitamente al constructor de la superclase, Java invoca el constructor **por defecto** de la superclase.

Ejemplo

```
public class Animal {
    public Animal() {
          System.out.println("Constructor de Animal");
    }
}
public class Perro extends Animal {
    public Perro() {
          System.out.println("Constructor de Perro");
    }
}
public class Main {
    public static void main(String[] args) {
          Perro miPerro = new Perro();
    }
}
```

Salida:

```
Constructor de Animal
Constructor de Perro
```

Invocación Explícita del Constructor con `super`

Si la superclase no tiene un constructor por defecto o deseas invocar un constructor específico, debes usar `super`.

Ejemplo

```
public class Animal {
    private String especie;
    public Animal(String especie) {
          this.especie = especie;
          System.out.println("Especie: " + especie);
    }
}
public class Perro extends Animal {
    public Perro(String especie) {
          super(especie); // Llama al constructor de Animal
          System.out.println("Es un perro.");
    }
}
public class Main {
    public static void main(String[] args) {
          Perro miPerro = new Perro("Canino");
    }
}
```

Salida:

```
Especie: Canino
Es un perro.
```

Constructor Parametrizado en la Superclase y Subclase

Cuando tanto la superclase como la subclase tienen constructores parametrizados, se pueden combinar para inicializar atributos en ambos niveles.

Ejemplo

```
public class Empleado {
    protected String nombre;
    public Empleado(String nombre) {
        this.nombre = nombre;
    }
}
public class Gerente extends Empleado {
    private String departamento;
    public Gerente(String nombre, String departamento) {
        super(nombre); // Llama al constructor de Empleado
        this.departamento = departamento;
    }
    public void mostrarInformacion() {
        System.out.println("Nombre: " + nombre);
        System.out.println("Departamento: " + departamento);
    }
}
public class Main {
    public static void main(String[] args) {
        Gerente gerente = new Gerente("Laura", "Recursos Humanos");
        gerente.mostrarInformacion();
    }
}
```

Salida:

```
Nombre: Laura
Departamento: Recursos Humanos
```

3. Reglas Importantes sobre Constructores y Herencia

1. La superclase debe tener un constructor accesible:

- Si no tiene un constructor por defecto, debe llamarse explícitamente uno de sus constructores parametrizados.

2. La palabra clave `super`:

- Debe ser la primera línea del constructor de la subclase si se utiliza.
- No puede coexistir con `this` en el mismo constructor.

3. Constructores no se heredan:

- Aunque los métodos se heredan, los constructores no lo hacen. Las subclases deben definir sus propios constructores.

4. Beneficios de Constructores en Herencia

1. Inicialización Completa:

Garantiza que tanto los atributos de la superclase como los de la subclase estén correctamente inicializados.

2. Evita Duplicación de Código:

Centraliza la inicialización de atributos comunes en la superclase.

3. Flexibilidad:

Permite personalizar la inicialización en diferentes niveles de la jerarquía de herencia.

5. Ejercicio Guiado

Problema

1. Crea una superclase `Vehiculo` con:

- Atributos: `marca` (String) y `velocidadMaxima` (int).
- Constructor parametrizado para inicializar estos atributos.

2. Crea una subclase `Coche` que:

- Añada el atributo `numeroPuertas` (int).
- Use `super` para inicializar los atributos heredados.

3. Implementa un método `mostrarInformacion()` en ambas clases para mostrar los detalles completos del objeto.

6. Preguntas de Evaluación

1. ¿Qué palabra clave se utiliza para invocar un constructor de la superclase?

a) `this`
b) `super`
c) `extends`
d) Ninguna de las anteriores

2. ¿Qué ocurre si una subclase no llama explícitamente al constructor de la superclase?

a) Se llama automáticamente al constructor por defecto de la superclase.
b) No se inicializan los atributos de la superclase.
c) Se genera un error en tiempo de compilación.
d) Se inicializan automáticamente con valores predeterminados.

3. ¿Es posible heredar un constructor de una superclase?

a) Sí, si es público.
b) No, los constructores no se heredan.
c) Sí, si la subclase no tiene constructores propios.
d) No, excepto si se usa `super`.

4. ¿Dónde debe ubicarse la llamada a `super` en un constructor?

a) En cualquier parte del constructor.
b) Antes de la declaración de los atributos de la subclase.
c) Debe ser la primera línea del constructor.
d) No es obligatorio llamarla.

Parte 5: Modificadores en clases, atributos y métodos.

Los **modificadores de acceso** en Java son herramientas esenciales para definir la **visibilidad** y **control de acceso** a las clases, atributos y métodos. Con ellos, podemos establecer reglas sobre quién puede acceder o modificar determinadas partes del código.

1. Modificadores de Acceso

En Java, los modificadores de acceso determinan la visibilidad de las clases, atributos y métodos. Existen cuatro niveles principales:

Modificador	Dentro de la Clase	Dentro del Paquete	En Subclases	Desde Fuera
`public`	√	√	√	√
`protected`	√	√	√	×
(sin modificador)	√	√	×	×
`private`	√	×	×	×

1.1. `public`

- Permite que la clase, atributo o método sea accesible desde **cualquier lugar**.
- Es el modificador más abierto.

Ejemplo: Uso de `public`

```
public class Persona {
    public String nombre;
    public void mostrarNombre() {
        System.out.println("Nombre: " + nombre);
    }
}
public class Main {
    public static void main(String[] args) {
        Persona persona = new Persona();
        persona.nombre = "Juan";
        persona.mostrarNombre(); // Acceso público
    }
}
```

1.2. `private`

- Limita el acceso únicamente a la **clase donde se declara**.

- Es el modificador más restrictivo y se usa para la encapsulación.

Ejemplo: Uso de `private`

```
public class Persona {
    private String nombre;
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public String getNombre() {
        return nombre;
    }
}
public class Main {
    public static void main(String[] args) {
        Persona persona = new Persona();
        persona.setNombre("Ana"); // Acceso a través de un método público
        System.out.println(persona.getNombre());
    }
}
```

Salida:

```
Ana
```

1.3. `protected`

- Permite acceso desde:
  - La misma clase.
  - Clases dentro del mismo paquete.
  - Subclases (incluso si están en un paquete diferente).

Ejemplo: Uso de `protected`

```
  public class Animal {
      protected String especie;
      public void mostrarEspecie() {
            System.out.println("Especie: " + especie);
      }
  }
  public class Perro extends Animal {
      public void ladrar() {
            System.out.println("El perro de especie " + especie + " está ladran-
do.");
      }
  }
  public class Main {
      public static void main(String[] args) {
            Perro miPerro = new Perro();
            miPerro.especie = "Canino";
            miPerro.mostrarEspecie();
            miPerro.ladrar();
      }
  }
```

1.4. Sin Modificador (Default o Package-Private)

- Permite acceso solo desde clases dentro del mismo paquete.
- Es el modificador predeterminado si no se especifica otro.

Ejemplo: Sin Modificador

```
class Vehiculo {
    String marca; // Default
    void mostrarMarca() {
        System.out.println("Marca: " + marca);
    }
}
public class Main {
    public static void main(String[] args) {
        Vehiculo vehiculo = new Vehiculo();
        vehiculo.marca = "Toyota"; // Acceso permitido dentro del paquete
        vehiculo.mostrarMarca();
    }
}
```

2. Modificadores en Clases

En Java, solo se pueden usar los modificadores `public` y `default` (sin modificador) para las clases.

1. Clases `public`:

- Accesibles desde cualquier lugar.

2. Clases con Default:

- Solo accesibles desde el mismo paquete.

Ejemplo

```
public class ClasePublica {
    // Clase accesible desde cualquier lugar
}
class ClaseDefault {
    // Clase accesible solo dentro del paquete
}
```

3. Modificadores en Métodos

Los métodos pueden usar todos los modificadores de acceso (`public`, `protected`, `private`, default). Además, pueden tener otros modificadores:

1. `static`:
   - Pertenece a la clase en lugar de a una instancia específica.
   - Se puede llamar sin crear un objeto.

```
public static void saludar() {
    System.out.println("Hola");
}
```

2. `final`:
   - Evita que el método sea sobrescrito por subclases.

```
public final void despedir() {
    System.out.println("Adiós");
}
```

3. `abstract`:
   - Declara un método sin implementación (solo en clases abstractas).

```
public abstract void metodoAbstracto();
```

4. Modificadores en Atributos

1. `final`:
   - Impide que el valor del atributo sea modificado después de ser inicializado.

```
public final int MAX_EDAD = 100;
```

2. `static`:
   - El atributo pertenece a la clase, no a las instancias.

```
public static String idioma = "Español";
```

5. Ejercicio Guiado

Problema

1. Define una clase `Persona` con los atributos:

- `nombre` (private), `edad` (protected), y `pais` (public).
- Crea métodos getter y setter para `nombre`.

2. Define una subclase `Estudiante` que:

- Herede de `Persona`.
- Implemente un método para mostrar toda la información.

3. En la clase principal:

- Crea un objeto de `Estudiante` y prueba los modificadores.

6. Preguntas de Evaluación

1. ¿Qué modificador permite acceso desde cualquier lugar?

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

2. ¿Qué modificador solo permite acceso dentro de la clase?

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

3. ¿Qué hace un atributo `final`?

a) Evita que sea heredado.
b) Evita que sea sobrescrito.
c) Evita que sea modificado después de inicializarse.
d) Ninguna de las anteriores.

4. ¿Qué modificador permite acceso a subclases incluso en diferentes paquetes?

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

Parte 6: Sobreescritura de métodos.

La **sobrescritura de métodos** (method overriding) es un mecanismo clave en la **Programación Orientada a Objetos (POO)** que permite a una subclase proporcionar una implementación específica para un método que ya está definido en su **superclase**.

1. ¿Qué es la Sobrescritura de Métodos?

1. Definición:

- La sobrescritura ocurre cuando una subclase redefine un método de la superclase usando el mismo nombre, tipo de retorno y parámetros.
- Cambia o extiende el comportamiento del método heredado.

2. Características Clave:

- El método en la subclase debe tener la **misma firma** (nombre, parámetros y tipo de retorno) que el de la superclase.
- No se puede reducir la visibilidad del método (por ejemplo, de `public` a `protected`).
- Debe usar la anotación `@Override` para indicar explícitamente que está sobrescribiendo un método.

1.1. Uso de la Anotación `@Override`

La anotación `@Override` asegura que el compilador verifique si el método realmente está sobrescribiendo un método de la superclase.

Ejemplo: Uso de `@Override`

```
public class Animal {
    public void sonido() {
          System.out.println("El animal hace un sonido.");
    }
}
public class Perro extends Animal {
    @Override
    public void sonido() {
          System.out.println("El perro ladra.");
    }
}
public class Main {
    public static void main(String[] args) {
          Animal miAnimal = new Perro();
          miAnimal.sonido(); // Llama al método sobrescrito en Perro
    }
}
```

Salida:

```
El perro ladra.
```

2. Reglas de Sobrescritura

1. Firma del Método:

- La subclase debe mantener el mismo nombre, parámetros y tipo de retorno que el método de la superclase.

2. Visibilidad:

- No se puede reducir la visibilidad del método. Por ejemplo, un método `public` en la superclase no puede ser `protected` o `private` en la subclase.

3. Modificadores `final` y `static`:

- Los métodos marcados como `final` no se pueden sobrescribir.
- Los métodos `static` no se pueden sobrescribir; en su lugar, se ocultan (hiding).

4. Excepciones:

- La subclase no puede lanzar excepciones más generales que las declaradas en la superclase.

3. Diferencia entre Sobrecarga y Sobrescritura

Característica	Sobrecarga de Métodos	Sobrescritura de Métodos
Contexto	Mismo método en la misma clase.	Método en la subclase con el mismo nombre y firma.
Parámetros	Deben ser diferentes en tipo o número.	Deben ser exactamente iguales.
Tipo de Retorno	Puede cambiar.	Debe ser el mismo o compatible covariantemente.
Modificadores	Puede tener cualquier nivel de visibilidad.	No puede reducir la visibilidad.

4. Uso de la Palabra Clave `super`

En la sobrescritura, se puede usar la palabra clave `super` para llamar al método de la superclase desde la subclase.

Ejemplo: Uso de `super`

```
public class Animal {
    public void sonido() {
        System.out.println("El animal hace un sonido.");
    }
}
public class Gato extends Animal {
    @Override
    public void sonido() {
        super.sonido(); // Llama al método de la superclase
        System.out.println("El gato maúlla.");
    }
}
public class Main {
    public static void main(String[] args) {
        Gato miGato = new Gato();
        miGato.sonido();
    }
}
```

Salida:

```
El animal hace un sonido.
El gato maúlla.
```

5. Ejemplo Completo

Ejemplo: Clases de Empleados

1. Crea una superclase `Empleado` con:

- Atributo `nombre`.
- Método `calcularSalario()` que devuelve un valor base.

2. Crea dos subclases:

- `EmpleadoTiempoCompleto`: Sobrescribe `calcularSalario()` para devolver un salario fijo.
- `EmpleadoPorHoras`: Sobrescribe `calcularSalario()` para devolver un salario basado en horas trabajadas.

```
public class Empleado {
    protected String nombre;
    public Empleado(String nombre) {
        this.nombre = nombre;
    }
    public double calcularSalario() {
        return 1000; // Salario base
    }
    public void mostrarInformacion() {
        System.out.println("Empleado: " + nombre);
        System.out.println("Salario: " + calcularSalario());
    }
}
public class EmpleadoTiempoCompleto extends Empleado {
    public EmpleadoTiempoCompleto(String nombre) {
        super(nombre);
    }
    @Override
    public double calcularSalario() {
        return 2000; // Salario fijo para empleados a tiempo completo
    }
}
public class EmpleadoPorHoras extends Empleado {
    private int horasTrabajadas;
    private double tarifaPorHora;
    public EmpleadoPorHoras(String nombre, int horasTrabajadas, double tari-
faPorHora) {
        super(nombre);
        this.horasTrabajadas = horasTrabajadas;
        this.tarifaPorHora = tarifaPorHora;
    }
    @Override
    public double calcularSalario() {
        return horasTrabajadas * tarifaPorHora;
    }
}
public class Main {
    public static void main(String[] args) {
        Empleado emp1 = new EmpleadoTiempoCompleto("Ana");
        Empleado emp2 = new EmpleadoPorHoras("Carlos", 40, 15);
        emp1.mostrarInformacion();
        emp2.mostrarInformacion();
    }
```

```
}
```

Salida:

```
Empleado: Ana
Salario: 2000.0
Empleado: Carlos
Salario: 600.0
```

6. Ejercicio Guiado

Problema

1. Crea una clase `Figura` con un método `calcularArea()`.

2. Crea dos subclases:

- `Rectangulo`: Calcula el área de un rectángulo.
- `Circulo`: Calcula el área de un círculo.

3. Usa la sobrescritura para personalizar el método `calcularArea()` en cada subclase.

7. Preguntas de Evaluación

1. ¿Qué palabra clave se usa para indicar que un método está sobrescribiendo otro?

a) `@Override`
b) `@Overload`
c) `@Super`
d) Ninguna de las anteriores

2. ¿Qué ocurre si un método sobrescrito intenta reducir la visibilidad del método original?

a) Produce un error de compilación.
b) Se permite si no se usa la anotación `@Override`.
c) Se ejecuta normalmente.
d) Se ignora la sobrescritura.

3. ¿Cuál de los siguientes métodos puede ser sobrescrito?

a) `final`
b) `private`
c) `static`
d) Ninguno de los anteriores

4. ¿Qué se requiere para sobrescribir correctamente un método?

a) Cambiar el tipo de retorno.
b) Mantener la misma firma del método.
c) Reducir el nivel de acceso.
d) Usar `static` en ambos métodos.

Parte 7: Clases y métodos abstractos y finales

Las **clases abstractas** y los **métodos abstractos** son fundamentales en Java para definir comportamientos genéricos que se implementarán de manera específica en las subclases. Por otro lado, las clases y métodos **finales** se utilizan para restringir la herencia y la sobrescritura.

1. Clases Abstractas

1. Definición:

Una **clase abstracta** es una clase que no se puede instanciar directamente.

- Se utiliza para definir una estructura base para otras clases.
- Puede contener métodos concretos (con implementación) y métodos abstractos (sin implementación).

2. Uso Principal:

- Modelar **conceptos genéricos** que comparten características comunes.
- Forzar a las subclases a proporcionar implementaciones específicas de los métodos abstractos.

3. Palabra Clave:

Se utiliza `abstract` para declarar una clase como abstracta.

Ejemplo de Clase Abstracta

```
public abstract class Figura {
    protected String color;
    public Figura(String color) {
        this.color = color;
    }
    // Método abstracto: sin implementación
    public abstract double calcularArea();
    // Método concreto: con implementación
    public void mostrarColor() {
        System.out.println("Color: " + color);
    }
}
public class Rectangulo extends Figura {
    private double largo;
    private double ancho;
    public Rectangulo(String color, double largo, double ancho) {
        super(color);
        this.largo = largo;
        this.ancho = ancho;
    }
    @Override
    public double calcularArea() {
        return largo * ancho;
    }
}
public class Main {
    public static void main(String[] args) {
        Rectangulo rectangulo = new Rectangulo("Rojo", 5, 3);
        rectangulo.mostrarColor();
        System.out.println("Área: " + rectangulo.calcularArea());
    }
}
```

Salida:

```
Color: Rojo
Área: 15.0
```

Características de las Clases Abstractas

1. No Instanciables:

No se pueden crear objetos directamente de una clase abstracta.

```
Figura figura = new Figura("Azul"); // Error de compilación
```

2. Métodos Abstractos:

- Son métodos sin cuerpo (solo declaración).
- Las subclases deben implementar todos los métodos abstractos de la superclase.

```
public abstract double calcularArea();
```

3. Métodos Concretos:

- Las clases abstractas pueden contener métodos con cuerpo que pueden ser utilizados por las subclases.

2. Métodos Abstractos

1. Definición:

Un método abstracto es un método declarado sin cuerpo.

- Solo puede estar en clases abstractas.
- Obliga a las subclases a proporcionar su implementación.

2. Ejemplo:

```
public abstract class Animal {
    public abstract void sonido();
}
public class Perro extends Animal {
    @Override
    public void sonido() {
          System.out.println("El perro ladra.");
    }
}
```

3. Clases Finales

1. Definición:

Una **clase final** es una clase que no se puede heredar.

- Se utiliza para evitar que otras clases extiendan una clase específica.

2. Palabra Clave:

Se utiliza `final` para declarar una clase como final.

3. Ejemplo:

```
public final class Utilidades {
    public static void imprimirMensaje(String mensaje) {
          System.out.println(mensaje);
    }
}
// Esto generará un error:
public class SubClase extends Utilidades {
    // No se puede heredar de una clase final
}
```

4. Métodos Finales

1. Definición:

Un método final es un método que no puede ser sobrescrito por las subclases.

- Se utiliza para preservar el comportamiento de un método.

2. Ejemplo:

```
public class Persona {
    public final void mostrarIdentidad() {
        System.out.println("Soy una persona.");
    }
}
public class Estudiante extends Persona {
    // Esto generará un error:
    @Override
    public void mostrarIdentidad() {
        System.out.println("Soy un estudiante.");
    }
}
```

5. Diferencias entre Clases Abstractas y Finales

Característica	Clases Abstractas	Clases Finales
Instanciación	No se pueden instanciar.	Se pueden instanciar.
Herencia	Pueden ser heredadas.	No pueden ser heredadas.
Propósito	Definir una estructura base para subclases.	Evitar que otras clases hereden.

6. Ejemplo Completo

Problema: Figuras Geométricas con Restricciones

1. Define una clase abstracta `Figura` con un método abstracto `calcularArea()`.
2. Crea una subclase `Triangulo` que implemente el cálculo del área.
3. Define una clase `Constantes` como final para almacenar valores constantes (como PI).
4. En la clase principal, crea objetos de las subclases y muestra sus áreas.

```
public abstract class Figura {
    protected String nombre;
    public Figura(String nombre) {
        this.nombre = nombre;
    }
    public abstract double calcularArea();
    public void mostrarNombre() {
        System.out.println("Figura: " + nombre);
    }
}
public class Triangulo extends Figura {
    private double base;
    private double altura;
    public Triangulo(String nombre, double base, double altura) {
        super(nombre);
        this.base = base;
        this.altura = altura;
    }
    @Override
    public double calcularArea() {
        return (base * altura) / 2;
    }
}
public final class Constantes {
    public static final double PI = 3.14159;
}
public class Main {
    public static void main(String[] args) {
        Triangulo triangulo = new Triangulo("Triángulo", 5, 3);
        triangulo.mostrarNombre();
        System.out.println("Área: " + triangulo.calcularArea());
        System.out.println("Valor de PI: " + Constantes.PI);
    }
}
```

Salida:

```
Figura: Triángulo
Área: 7.5
Valor de PI: 3.14159
```

7. Ejercicio Guiado

Problema

1. Crea una clase abstracta `Empleado` con el método abstracto `calcularSalario()`.
2. Define una subclase `EmpleadoPorHoras` que implemente el cálculo del salario basado en horas trabajadas.
3. Define una clase final `ConstantesEmpresa` para almacenar valores como el salario mínimo.
4. En la clase principal, crea un objeto de `EmpleadoPorHoras` e imprime su salario.

8. Preguntas de Evaluación

1. ¿Qué ocurre si una clase contiene un método abstracto y no se declara como abstracta?

a) Se genera un error de compilación.
b) Se permite, pero el método no tiene cuerpo.
c) La clase no puede ser heredada.
d) Ninguna de las anteriores.

2. ¿Qué ocurre si una subclase no implementa todos los métodos abstractos de su superclase abstracta?

a) Se permite, pero no se puede instanciar.
b) La subclase debe declararse como abstracta.
c) Ambas respuestas anteriores son correctas.
d) Ninguna de las anteriores.

3. ¿Qué hace un método `final`?

a) Permite ser sobrescrito.
b) Impide ser sobrescrito.
c) Declara que no tiene cuerpo.
d) Ninguna de las anteriores.

4. ¿Qué característica define una clase abstracta?

a) No se puede heredar.
b) No se puede instanciar directamente.
c) Todos sus métodos deben ser abstractos.
d) Ninguna de las anteriores.

Parte 8: Interfaces. Clases abstractas vs. Interfaces.

1. ¿Qué es una Interfaz?

Una **interfaz** es una estructura que define un conjunto de métodos que una clase debe implementar. A diferencia de una clase abstracta, una interfaz no contiene atributos ni métodos concretos (hasta Java 8, con algunas excepciones).

1. Definición:

- Una interfaz actúa como un contrato que las clases que la implementan deben cumplir.
- Los métodos en una interfaz son implícitamente **abstractos** y **públicos** (no requieren las palabras clave `abstract` o `public`).

2. Palabra Clave:

- Se utiliza la palabra clave `interface` para definir una interfaz.
- Una clase utiliza la palabra clave `implements` para implementar una interfaz.**1.1. Ejemplo Básico**

```
// Definición de una interfaz
public interface Vehiculo {
    void encender();
    void apagar();
}
// Implementación de la interfaz
public class Coche implements Vehiculo {
    @Override
    public void encender() {
          System.out.println("El coche está encendido.");
    }
    @Override
    public void apagar() {
          System.out.println("El coche está apagado.");
    }
}
public class Main {
    public static void main(String[] args) {
          Coche miCoche = new Coche();
          miCoche.encender();
          miCoche.apagar();
    }
}
```

Salida:

```
El coche está encendido.
El coche está apagado.
```

1.2. Características de las Interfaces

1. Múltiples Implementaciones:

- Una clase puede implementar múltiples interfaces, superando la limitación de herencia simple.

```
public interface Volador {
    void volar();
}
public interface Nadador {
    void nadar();
}
public class Pato implements Volador, Nadador {
    @Override
    public void volar() {
          System.out.println("El pato está volando.");
    }
    @Override
    public void nadar() {
          System.out.println("El pato está nadando.");
    }
}
```

2. Métodos por Defecto (Desde Java 8):

- Las interfaces pueden contener métodos concretos con la palabra clave `default`.

```
public interface Vehiculo {
    default void frenar() {
          System.out.println("El vehículo está frenando.");
    }
}
```

3. Métodos Estáticos (Desde Java 8):

- Las interfaces pueden contener métodos estáticos.

```
public interface Utilidades {
    static void mostrarMensaje(String mensaje) {
          System.out.println(mensaje);
    }
}
```

2. Diferencias entre Clases Abstractas e Interfaces

Aspecto	Clases Abstractas	Interfaces
Instanciación	No se pueden instanciar.	No se pueden instanciar.
Herencia	Permite heredar de una sola clase.	Una clase puede implementar múltiples interfaces.
Atributos	Puede tener atributos con cualquier modificador de acceso.	Solo atributos estáticos y finales.
Métodos	Métodos concretos y abstractos.	Métodos abstractos, por defecto y estáticos.
Propósito	Modelar jerarquías de clases.	Definir un contrato de comportamiento.

2.1. Ejemplo ComparativoClase Abstracta

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

Interfaz

```
public interface Volador {
    void volar();
}
```

Uso Combinado

```
public class Pajaro extends Animal implements Volador {
    public Pajaro(String nombre) {
        super(nombre);
    }
    @Override
    public void sonido() {
        System.out.println(nombre + " canta.");
    }
    @Override
    public void volar() {
        System.out.println(nombre + " está volando.");
    }
}
```

3. Ventajas y Limitaciones de las Interfaces

Ventajas

1. Herencia Múltiple:

- Una clase puede implementar múltiples interfaces, permitiendo combinar comportamientos.

2. Desacoplamiento:

- Facilita el diseño de sistemas modulares al definir contratos claros.

3. Compatibilidad:

- Las interfaces pueden ser implementadas por cualquier clase, sin importar su jerarquía.

Limitaciones

1. Falta de Estado Compartido:

- No se pueden definir atributos no estáticos.

2. Comportamiento General:

- Las interfaces no pueden contener lógica común para ser reutilizada directamente (salvo métodos por defecto).

4. Ejemplo Completo

Problema: Vehículos con Diferentes Capacidades

1. Define una interfaz `Vehiculo` con los métodos `arrancar()` y `detener()`.

2. Define dos interfaces adicionales:

- `Volador`: Método `volar()`.
- `Nadador`: Método `nadar()`.

3. Implementa dos clases:

- `Avion`, que implemente `Vehiculo` y `Volador`.
- `Barco`, que implemente `Vehiculo` y `Nadador`.

```
public interface Vehiculo {
    void arrancar();
    void detener();
}
public interface Volador {
    void volar();
}
public interface Nadador {
    void nadar();
}
public class Avion implements Vehiculo, Volador {
    @Override
    public void arrancar() {
          System.out.println("El avión está arrancando.");
    }
    @Override
    public void detener() {
          System.out.println("El avión está deteniéndose.");
    }
    @Override
    public void volar() {
          System.out.println("El avión está volando.");
    }
}
public class Barco implements Vehiculo, Nadador {
    @Override
    public void arrancar() {
          System.out.println("El barco está arrancando.");
    }
    @Override
    public void detener() {
          System.out.println("El barco está deteniéndose.");
    }
    @Override
    public void nadar() {
          System.out.println("El barco está navegando.");
    }
}
public class Main {
    public static void main(String[] args) {
          Avion miAvion = new Avion();
          Barco miBarco = new Barco();
          miAvion.arrancar();
```

```
            miAvion.volar();
            miAvion.detener();
            miBarco.arrancar();
            miBarco.nadar();
            miBarco.detener();
        }
    }
```

Salida:

```
El avión está arrancando.
El avión está volando.
El avión está deteniéndose.
El barco está arrancando.
El barco está navegando.
El barco está deteniéndose.
```

5. Ejercicio Guiado

Problema

1. **Define una interfaz `Empleado` con los métodos `trabajar()` y `descansar()`.**
2. **Crea una interfaz adicional `Freelancer` con el método `cotizarProyecto()`.**
3. **Implementa una clase `Desarrollador` que combine ambas interfaces.**

6. Preguntas de Evaluación

1. ¿Qué palabra clave se usa para implementar una interfaz?

a) `extends`
b) `implements`
c) `interface`
d) Ninguna de las anteriores

2. ¿Qué tipo de métodos se pueden definir en una interfaz?

a) Métodos concretos únicamente.
b) Métodos abstractos, por defecto y estáticos.
c) Solo métodos abstractos.
d) Métodos finales.

3. ¿Puede una clase implementar más de una interfaz?

a) Sí.
b) No.
c) Solo si no tiene métodos abstractos.
d) Ninguna de las anteriores.

Conclusión del Bloque: Utilización avanzada de clases

En este bloque, hemos explorado conceptos avanzados que son fundamentales para el diseño y la implementación de sistemas orientados a objetos en Java. Estos conceptos proporcionan herramientas y estrategias para modelar relaciones, comportamientos y restricciones en las clases y sus interacciones.

1. Resumen de los Puntos Clave

1. Relaciones entre Clases y Composición

- Las relaciones entre clases permiten conectar entidades en un diseño lógico.
- **Composición** y **agregación** modelan relaciones “tiene un”, mientras que la **herencia** representa relaciones “es un”.

2. Herencia y Jerarquías

- **Herencia** permite reutilizar y extender el comportamiento de clases existentes.
- **Superclases y subclases** establecen una jerarquía clara para organizar el código y fomentar la reutilización.
- El uso de **constructores en herencia** asegura una inicialización adecuada en todos los niveles.

3. Modificadores en Clases, Atributos y Métodos

- Los **modificadores de acceso** (`public`, `private`, `protected`, default) controlan la visibilidad y el acceso al código.
- Los modificadores como `final` y `static` agregan restricciones y propiedades especiales a los métodos y atributos.

4. Sobrescritura y Sobrecarga

- **Sobrescritura** permite personalizar el comportamiento de métodos heredados mediante `@Override`.
- **Sobrecarga** permite crear múltiples versiones de un método con diferentes firmas.

5. Clases Abstractas y Métodos Finales

- Las **clases abstractas** proporcionan una estructura base para las subclases, definiendo métodos abstractos y concretos.
- Los **métodos finales** y las **clases finales** restringen la modificación y la herencia, respectivamente.

6. Interfaces

- Las **interfaces** definen un contrato de comportamiento que las clases deben implementar.
- Java permite combinar múltiples interfaces, facilitando la herencia múltiple.
- Desde Java 8, las interfaces pueden incluir métodos por defecto y estáticos.

2. Ventajas del Uso Avanzado de Clases

1. Modularidad:

- Facilita la separación de responsabilidades, mejorando la legibilidad y mantenibilidad del código.

2. Flexibilidad y Reutilización:

- Las interfaces y la herencia permiten adaptar y reutilizar código en diferentes contextos.

3. Encapsulación:

- Los modificadores de acceso controlan cómo y dónde se puede interactuar con las clases y sus miembros.

4. Extensibilidad:

- Las jerarquías de clases bien diseñadas permiten extender funcionalidades de manera eficiente.

5. Consistencia:

- Los contratos definidos por interfaces garantizan que las implementaciones cumplan con las especificaciones establecidas.

3. Ejercicio Final del Bloque

Problema

Diseña un sistema para gestionar un zoológico utilizando los conceptos del bloque:

1. Define una clase abstracta `Animal` con:

- Atributos: `nombre` y `edad`.
- Método abstracto `hacerSonido()`.
- Método concreto `mostrarInformacion()` para imprimir los atributos.

2. Crea subclases que hereden de `Animal`:

- `Perro`: Implementa el método `hacerSonido()` para imprimir “El perro ladra”.
- `Gato`: Implementa el método `hacerSonido()` para imprimir “El gato maúlla”.

3. Define una interfaz `Domestico` con el método `jugar()`.

- Haz que `Perro` y `Gato` implementen esta interfaz.

4. En la clase principal:

- Crea una lista de animales.
- Muestra su información y sus sonidos.
- Llama al método `jugar()` para los animales domésticos.

Código Propuesto

```
import java.util.ArrayList;
public abstract class Animal {
    protected String nombre;
    protected int edad;
    public Animal(String nombre, int edad) {
          this.nombre = nombre;
          this.edad = edad;
    }
    public abstract void hacerSonido();
    public void mostrarInformacion() {
          System.out.println("Nombre: " + nombre + ", Edad: " + edad);
    }
}
public interface Domestico {
    void jugar();
}
public class Perro extends Animal implements Domestico {
    public Perro(String nombre, int edad) {
          super(nombre, edad);
    }
    @Override
    public void hacerSonido() {
          System.out.println("El perro ladra.");
    }
    @Override
    public void jugar() {
          System.out.println(nombre + " está jugando con una pelota.");
    }
}
public class Gato extends Animal implements Domestico {
    public Gato(String nombre, int edad) {
          super(nombre, edad);
    }
    @Override
    public void hacerSonido() {
          System.out.println("El gato maúlla.");
    }
    @Override
    public void jugar() {
          System.out.println(nombre + " está jugando con un ovillo de lana.");
    }
```

```
}
public class Main {
    public static void main(String[] args) {
          ArrayList<Animal> animales = new ArrayList<>();
          animales.add(new Perro("Rex", 5));
          animales.add(new Gato("Luna", 3));
          for (Animal animal : animales) {
                animal.mostrarInformacion();
                animal.hacerSonido();
                if (animal instanceof Domestico) {
                      ((Domestico) animal).jugar();
                }
                System.out.println("---");
          }
    }
}
```

Salida:

```
Nombre: Rex, Edad: 5
El perro ladra.
Rex está jugando con una pelota.
---
Nombre: Luna, Edad: 3
El gato maúlla.
Luna está jugando con un ovillo de lana.
---
```

4. Preguntas de Evaluación

1. ¿Qué palabra clave se utiliza para implementar una interfaz en Java?

a) `extends`
b) `implements`
c) `interface`
d) Ninguna de las anteriores

2. ¿Qué tipo de métodos puede contener una clase abstracta?

a) Métodos concretos únicamente.
b) Métodos abstractos únicamente.
c) Métodos concretos y abstractos.
d) Solo métodos estáticos.

3. ¿Cuál es la diferencia clave entre una clase abstracta y una interfaz?

a) Una interfaz no puede contener métodos concretos.
b) Una clase abstracta permite heredar de múltiples clases.
c) Una interfaz define un contrato de comportamiento, mientras que una clase abstracta puede contener lógica común.
d) No hay diferencia.

4. ¿Qué ocurre si una subclase no implementa un método abstracto de su superclase?

a) Produce un error de compilación.
b) Se permite, pero el método no puede ser llamado.
c) La subclase debe declararse como abstracta.
d) Ambas respuestas a) y c) son correctas.