
Taller Práctico y Guiado sobre Herencia e Interfaces en Java

Este taller tiene como objetivo guiarte a través de los conceptos de herencia, interfaces, y otros aspectos clave de la Programación Orientada a Objetos en Java. A lo largo de este taller, desarrollaremos ejemplos prácticos y ejercicios para reforzar tu comprensión.

Objetivos del Taller

1. Comprender los conceptos de herencia y polimorfismo en Java.
2. Aprender a utilizar `abstract`, `interface`, `extends`, `implements`, `super`, `@Override`, `final`, `protected`.
3. Ejercitar la sobreescritura de métodos y el uso del polimorfismo.

Conceptos y Ejercicios

1. Clases Abstractas y Métodos Abstractos

- **Concepto**: Una clase abstracta no se puede instanciar y puede contener métodos abstractos (sin implementación) y métodos concretos (con implementación).
- **Uso**: Se utilizan para definir una plantilla común para un grupo de subclases.

Ejemplo y Ejercicio:

java
Copiar código

```java
// Clase abstracta Vehiculo
abstract class Vehiculo {
    protected String marca;

    // Método abstracto (sin implementación)
    public abstract void encender();

    // Método concreto
    public void mostrarMarca() {
        System.out.println("Marca: " + marca);
    }
}

// Clase Carro que extiende de Vehiculo
class Carro extends Vehiculo {
    public Carro(String marca) {
        this.marca = marca;
    }

    // Implementación del método abstracto
    @Override
    public void encender() {
        System.out.println("El carro está encendido.");
    }
}
```

Ejercicio:

1. Crea una clase abstracta llamada `Animal` con un método abstracto `hacerSonido()`.
2. Crea una clase `Perro` que extienda `Animal` y sobreescriba `hacerSonido()` para imprimir "Guau".

2. Interfaces y `implements`

- **Concepto**: Las interfaces definen un contrato que las clases pueden implementar. Todos los métodos en una interfaz son abstractos por defecto.
- **Uso**: Se usan para definir comportamientos que pueden ser implementados por cualquier clase, independientemente de su lugar en la jerarquía de herencia.

Ejemplo y Ejercicio:

java
Copiar código

```java
// Interfaz Volable
interface Volable {
    void volar();
}

// Clase Pajaro que implementa Volable
class Pajaro implements Volable {
    @Override
    public void volar() {
        System.out.println("El pájaro está volando.");
    }
}
```

Ejercicio:

1. Crea una interfaz `Nadador` con un método `nadar()`.
2. Implementa `Nadador` en una clase `Pez` que imprime "El pez está nadando.".

3. Uso de `extends` y `implements`

- **Concepto**: `extends` se usa para heredar de una clase base y `implements` se usa para implementar una o más interfaces.
- **Uso**: `extends` permite la reutilización del código de la clase padre, y `implements` permite que una clase implemente los métodos de una interfaz.

Ejemplo y Ejercicio:

java
Copiar código

```java
// Clase abstracta SerVivo
abstract class SerVivo {
    public abstract void respirar();
}

// Interfaz Caminante
interface Caminante {
    void caminar();
}

// Clase Humano que extiende SerVivo e implementa Caminante
class Humano extends SerVivo implements Caminante {
    @Override
    public void respirar() {
        System.out.println("El humano respira.");
    }

    @Override
    public void caminar() {
        System.out.println("El humano camina.");
    }
}
```

Ejercicio:

1. Crea una clase abstracta `Instrumento` con un método abstracto `tocar()`.
2. Define una interfaz `Afinable` con un método `afinar()`.

---

3. Crea una clase `Guitarra` que extienda `Instrumento` e implemente `Afinable`, sobreescribiendo ambos métodos.

4. Uso de `super` y `@Override`

- **Concepto**: `super` se usa para llamar al constructor de la clase padre o a sus métodos. `@Override` indica que un método está siendo sobrescrito de la clase padre.
- **Uso**: Se utiliza para personalizar o extender la funcionalidad de la clase base.

Ejemplo y Ejercicio:

java
Copiar código

```java
// Clase Animal con un método protegido
class Animal {
    protected String sonido = "Sonido de animal";

    public void hacerSonido() {
        System.out.println(sonido);
    }
}

// Clase Perro que extiende de Animal
class Perro extends Animal {
    public Perro() {
        super.sonido = "Guau";  // Uso de super para modificar el atributo de la clase padre
    }

    @Override
    public void hacerSonido() {
        super.hacerSonido(); // Llamada al método de la clase padre
        System.out.println("El perro está ladrando.");
    }
}
```

Ejercicio:

1. Crea una clase `Gato` que extienda `Animal` y sobreescriba `hacerSonido()`. Usa `super` para llamar al método de la clase base y luego agrega "El gato maúlla.".

5. `final` y `protected`

- **Concepto**: `final` se usa para evitar que una clase se herede, que un método se sobrescriba, o que una variable cambie su valor. `protected` permite que el acceso a un miembro de la clase sea posible dentro del paquete y por las subclases.
- **Uso**: Protege la integridad de la clase y sus métodos, y controla el acceso a los miembros de la clase.

Ejemplo y Ejercicio:

java
Copiar código

```java
// Clase final que no puede ser heredada
final class Constante {
    public final int valor = 42; // Variable final que no puede ser modificada

    // Método final que no puede ser sobrescrito
    public final void mostrarValor() {
        System.out.println("Valor: " + valor);
    }
}
```

Ejercicio:

1. Intenta crear una clase `SubConstante` que herede de `Constante` y observa qué error se produce.
2. Crea una clase `ClasePadre` con un método `protegido()` y verifica que solo puedes llamarlo desde dentro del mismo paquete o desde una subclase.

6. Polimorfismo y Sobreescritura de Métodos

- **Concepto**: El polimorfismo permite usar una referencia de la clase padre para manejar objetos de la clase hija. La sobreescritura permite a una subclase proporcionar una implementación específica de un método que ya está definido en la clase padre.
- **Uso**: Hace que el código sea más flexible y permite el uso de métodos de manera dinámica en tiempo de ejecución.

Ejemplo y Ejercicio:

java
Copiar código

```java
// Clase Vehiculo
class Vehiculo {
    public void mover() {
        System.out.println("El vehículo se mueve.");
    }
}

// Clase Moto que sobrescribe mover
class Moto extends Vehiculo {
    @Override
    public void mover() {
        System.out.println("La moto se mueve rápidamente.");
    }
}

// Clase principal
public class Main {
    public static void main(String[] args) {
        Vehiculo v = new Moto(); // Polimorfismo
        v.mover(); // Llama al método mover de la clase Moto
    }
}
```

Ejercicio:

1. Crea una clase `Avion` que sobrescriba el método `mover()` de la clase `Vehiculo` para que imprima "El avión vuela.".
2. Usa polimorfismo para crear una instancia de `Vehiculo` que apunte a un `Avion` y llama al método `mover()`.