Polimorfismo y Reutilización

Cosijopii García

8 de noviembre de 2024

Concepto del Polimorfismo I

- El polimorfismo es la capacidad de un objeto de tomar múltiples formas.
- En programación orientada a objetos, permite que una misma función o método tenga distintos comportamientos según el tipo de datos con los que opere.

```
class Animal {
public void sound() {
    System.out.println("Animal makes a sound");
}
}
class Dog extends Animal {
@Override
    public void sound() {
System.out.println("Dog barks");
}
}
```

Sobrecarga vs. Sobreescritura I

- **Sobrecarga**: Métodos con el mismo nombre pero diferentes parámetros.
- **Sobreescritura**: Una subclase redefine un método de su superclase.

```
Class MathOperations {
    public int sum(int a, int b) {
        return a + b;
    }

    public double sum(double a, double b) {
        return a + b;
    }
}
```

Constructores y Polimorfismo I

- Los constructores no pueden ser sobreescritos pero pueden ser sobrecargados.

```
class Car {
    private String model;
    private int year;
    // Constructor sin par metros
    public Car() {
        this.model = "Unknown";
        this.year = 0;
    }
    // Constructor con par metros
    public Car(String model, int year) {
        this.model = model;
        this.year = year;
    }
}
```

Uso de objetos a través de su clase base (Upcasting) I

- El upcasting permite tratar un objeto de una subclase como si fuera de su superclase.

```
Animal myDog = new Dog(); // Upcasting
myDog.sound(); //Llama al metodo sobrescrito
en Dog: "Dog barks"
```

Clases Abstractas I

- **Concepto**: Una clase abstracta no puede ser instanciada directamente.
- **Importancia**: Proporciona una base común para las clases derivadas.

```
abstract class Animal {
    public abstract void makeSound();
}

class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
```

Clases Abstractas II

```
    @Override
    public void makeSound() {
        System.out.println("Cat meows");
    }
}
```

¿Qué es una interfaz en Java? I

- Una **interfaz** en Java es una colección de métodos abstractos.
- Las interfaces pueden tener métodos abstractos (sin cuerpo) y métodos con implementación predeterminada (`default`).
- Unas clases pueden **implementar** una o más interfaces.
- Sirven para definir comportamientos que diferentes clases pueden compartir.

Herencia vs Interfaces en Java I

- **Herencia**:
  - Una clase puede heredar de una única clase base (*single inheritance*).
  - La herencia es una relación **es un tipo de** (is-a).
  - Las subclases heredan tanto los métodos como los atributos de la superclase.
  - Uso de la palabra clave `extends`.

- **Interfaces**:
  - Una clase puede implementar múltiples interfaces (*multiple inheritance*).
  - Las interfaces definen un conjunto de comportamientos que una clase debe implementar.
  - No hay herencia de atributos ni implementación de métodos (excepto métodos `default`).
  - Uso de la palabra clave `implements`.

Herencia vs Interfaces en Java II

Resumen:

- La herencia se centra en el **reuso de código** y es limitada a una sola superclase.
- Las interfaces permiten **compartir comportamientos** y permiten la implementación múltiple.

Definición de una Interfaz I

Ejemplo de una interfaz simple en Java:

```
interface Animal {
    void makeSound();
}
```

- La interfaz `Animal` declara un método abstracto `makeSound()`.
- Cualquier clase que implemente esta interfaz debe proporcionar su propia implementación de este método.

Implementación de una Interfaz I

Las clases que implementan una interfaz deben proporcionar implementaciones para todos los métodos de esa interfaz:

```
class Dog implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks");
    }
}

class Cat implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Cat meows");
    }
}
```

Uso de Interfaces I

Las interfaces permiten definir comportamientos compartidos entre clases no relacionadas:

```
public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        Animal myCat = new Cat();

        myDog.makeSound(); // Output: Dog barks
        myCat.makeSound(); // Output: Cat meows
    }
}
```

- Se pueden utilizar referencias de tipo `Animal` para objetos de clases que implementan la interfaz `Animal`.
- Esto permite cambiar el comportamiento de manera dinámica.

Implementación de Múltiples Interfaces I

Una clase puede implementar más de una interfaz:

```
interface Flyer {
    void fly();
}

interface Swimmer {
    void swim();
}

class Duck implements Flyer, Swimmer {
    @Override
    public void fly() {
        System.out.println("Duck flies");
    }

    @Override
```

Implementación de Múltiples Interfaces II

```
    public void swim() {
        System.out.println("Duck swims");
    }
}
```

- `Duck` implementa dos interfaces: `Flyer` y `Swimmer`.
- Debe proporcionar implementaciones para ambos métodos: `fly()` y `swim()`.

Métodos `default` en Interfaces I

A partir de Java 8, las interfaces pueden tener métodos con implementación:

```
interface Animal {
    void makeSound();

    default void sleep() {
        System.out.println("The animal is s
    }
}
```

- El método `sleep()` tiene una implementación predeterminada en la interfaz.
- Las clases que implementan `Animal` no están obligadas a sobrescribir este método, pero pueden hacerlo si lo desean.

Beneficios de Usar Interfaces I

- Facilitan la **polimorfia**: diferentes clases pueden tener una interfaz común.
- Permiten la **separación de la implementación y la especificación**.
- Facilitan la **modularidad y reutilización** de código.
- Se pueden usar para definir **contratos** entre diferentes partes del sistema.

Conclusión I

- Las interfaces son una herramienta clave para lograr la flexibilidad y modularidad en Java.
- Permiten a las clases compartir comportamientos comunes sin requerir una herencia rígida.
- Con el uso adecuado de interfaces, es posible construir sistemas más robustos, modulares y escalables.

¿Qué son las Inner Classes en Java?

- Las Inner Classes son clases definidas dentro de otras clases.
- Permiten organizar y encapsular lógica relacionada de manera conveniente.
- Pueden acceder a los miembros (atributos y métodos) de la clase que las contiene.

Tipos de Inner Classes

Existen cuatro tipos principales de Inner Classes:

- **Inner Class Regular** - Clase dentro de otra clase.
- **Inner Class Estática** - Clase anidada con el modificador `static`.
- **Clase Anónima** - Clase que se declara e instancia en una sola expresión.
- **Método Local** - Clase dentro de un método de la clase externa.

Inner Class Regular I

- Se define dentro de otra clase sin el modificador `static`.
- Puede acceder a todos los atributos y métodos de la clase externa, incluyendo privados.

Ejemplo:

```
public class Externa {
    private String mensaje = "Hola desde Externa!";

    public class Interna {
        public void imprimirMensaje() {
            System.out.println(mensaje);
        }
    }
}
```

Uso de la Inner Class Regular I

- Para instanciar una Inner Class Regular, se necesita una instancia de la clase externa.

Ejemplo de uso:

```
public class Main {
    public static void main(String[] args) {
        Externa externa = new Externa();
        Externa.Interna interna = externa.new Interr
        interna.imprimirMensaje();
    }
}
```

Inner Class Estática I

- Usa el modificador `static`.
- No puede acceder a los atributos y métodos de instancia de la clase externa.

Ejemplo:

```
public class Externa {
    public static class InternaEstatica {
        public void mensajeEstatico() {
            System.out.println("Mensaje desde Inner
        }
    }
}
```

Clase Anónima I

- Clase sin nombre que se crea e instancia en una sola expresión.
- Útil para implementar interfaces o clases abstractas de manera rápida.

Ejemplo:

```
public class Main {
    public static void main(String[] args) {
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
System.out.println("Ejecutando en clase anima");
            }
        };
        new Thread(runnable).start();
    }
}
```

Clase Método Local I

- Definida dentro de un método de la clase externa.
- Solo es visible dentro de ese método.

Ejemplo:

```
public class Externa {
    public void metodoExterno() {
        class MetodoLocal {
            void mostrar() {
System.out.println("Clase dentro de un m todo");
            }
        }
        MetodoLocal local = new MetodoLocal();
        local.mostrar();
    }
}
```

Ventajas y Desventajas de Inner Classes I

Ventajas:

- Mejora la encapsulación y organiza mejor el código.
- Permite clases auxiliares que solo serán usadas en la clase externa.

Desventajas:

- Puede complicar el código, reduciendo la legibilidad.
- Uso excesivo puede hacer el código más difícil de mantener.

Conclusión

- Las Inner Classes son una herramienta poderosa en Java que permite modularidad y encapsulación.
- Facilitan la implementación de clases auxiliares y lógica relacionada.
- Su uso debe ser mesurado para evitar sobrecomplicaciones en el código.

¿Qué es RTTI?

- RTTI (Runtime Type Identification) permite conocer el tipo exacto de un objeto en tiempo de ejecución.
- Útil en programación orientada a objetos para manejar referencias de superclase que apuntan a instancias de diversas subclases.
- Permite verificar el tipo de un objeto y aplicar comportamientos específicos de acuerdo con su clase.

¿Por qué es útil RTTI?

- Facilita el manejo de múltiples tipos de objetos en estructuras de datos o colecciones.
- Se usa cuando el tipo de un objeto debe conocerse o manipularse para ejecutar una operación particular.
- Permite aprovechar el polimorfismo y aplicar comportamientos específicos.

Operadores de RTTI en Java

- Java proporciona dos mecanismos principales para RTTI:
  - **Operador** `instanceof`: Verifica si un objeto es de un tipo específico o una de sus subclases.
  - **Método** `getClass()`: Retorna la clase exacta del objeto en tiempo de ejecución.

Ejemplo de RTTI con `instanceof` I

Supongamos un sistema con una clase `Animal` y subclases `Perro` y `Gato`.

```
public class Animal { }

public class Perro extends Animal {
    public void ladrar() {
        System.out.println("Guau!");
    }
}

public class Gato extends Animal {
    public void maullar() {
        System.out.println("Miau!");
    }
}
```

Ejemplo de RTTI con `instanceof` II

```
public class Main {
    public static void main(String[] args) {
        Animal animal = new Perro();

        if (animal instanceof Perro) {
            ((Perro) animal).ladrar();
        } else if (animal instanceof Gato) {
            ((Gato) animal).maullar();
        }
    }
}
```

Explicación del Ejemplo con `instanceof`

- Usamos `instanceof` para verificar el tipo exacto de `animal`.
- Si es de tipo `Perro`, llamamos al método `ladrar()`, que solo existe en `Perro`.
- Si es de tipo `Gato`, llamamos al método `maullar()`, que solo existe en `Gato`.
- Esto permite manejar tipos específicos en tiempo de ejecución.

Ejemplo de RTTI con `getClass()` I

```
public class Main {
    public static void main(String[] args) {
        Animal animal = new Gato();

        if (animal.getClass() == Perro.class) {
            System.out.println("Es un Perro");
        } else if (animal.getClass() == Gato.class)
            System.out.println("Es un Gato");
        }
    }
}
```

- `getClass()` devuelve la clase exacta del objeto.
- Comparamos con == para identificar el tipo.

Comparación: `instanceof` vs `getClass()`

- `instanceof` permite verificar si un objeto es de un tipo específico o de una subclase.
- `getClass()` devuelve la clase exacta del objeto, sin considerar subclases.
- `instanceof` es generalmente más flexible cuando se trabaja con jerarquías de herencia.

Ventajas y Desventajas de RTTI

Ventajas:

- Facilita la manipulación de tipos específicos en tiempo de ejecución.
- Permite lógica condicional basada en el tipo de objeto.

Desventajas:

- Su uso excesivo puede romper el principio de diseño Abierto/Cerrado.
- Puede hacer el código menos mantenible y reducir el polimorfismo.

Conclusión

- RTTI es una herramienta útil cuando es necesario identificar tipos en tiempo de ejecución.
- Puede mejorar la flexibilidad del código, pero su uso debe ser moderado para evitar una lógica excesivamente compleja.
- Usar RTTI en combinación con buenas prácticas de diseño orientado a objetos garantiza un código robusto y escalable.

Conceptos Generales de Excepciones

- Las **excepciones** son eventos que interrumpen el flujo normal de un programa.
- Se producen por errores en tiempo de ejecución, como división por cero o acceso fuera de los límites de un arreglo.

Definición de Excepción

- Una **excepción** es un objeto que representa un error o una condición inesperada.
- Permite gestionar errores sin finalizar abruptamente el programa.

Jerarquía de Excepciones en Java

- Java organiza sus excepciones en una jerarquía con `Throwable` como clase raíz.
- Principales subclases:
  - `Exception` (excepciones verificadas)
  - `RuntimeException` (excepciones no verificadas)
  - `Error` (problemas graves que normalmente no se pueden manejar)

Jerarquía de Excepciones - Ejemplo en Java

```
public class EjemploJerarquia {
    public static void main(String[] args) {
        try {
            int resultado = 10 / 0; // Provoca Arit
        } catch (ArithmeticException e) {
            System.out.println("Error: Division por
        }
    }
}
```

Propagación de Excepciones

- Las excepciones se propagan hacia arriba en la pila de llamadas hasta ser manejadas o hasta finalizar el programa.
- Si un método no maneja una excepción, se pasa al método que lo llamó.

Propagación - Ejemplo en Java

```
public class Propagacion {
    public static void metodoA() throws ArithmeticE
        metodoB();
    }

    public static void metodoB() throws ArithmeticE
        int resultado = 10 / 0;
    }

    public static void main(String[] args) {
        try {
            metodoA();
        } catch (ArithmeticException e) {
            System.out.println("Manejo de excepcion
        }
    }
}
```

Gestión de Excepciones

- **Manejo de Excepciones:** El uso de bloques `try-catch-finally` para capturar y manejar excepciones.
- **Lanzamiento de Excepciones:** Uso de la palabra clave `throw` para lanzar una excepción manualmente.

Manejo de Excepciones - Ejemplo en Java

```
public class ManejoExcepciones {
    public static void main(String[] args) {
        try {
            int[] numeros = {1, 2, 3};
            System.out.println(numeros[3]);
        } catch (ArrayIndexOutOfBoundsException e)
            System.out.println("Error: Indice fuera
        } finally {
            System.out.println("Bloque finally ejecu
        }
    }
}
```

Excepciones Definidas por el Usuario

- Los desarrolladores pueden crear sus propias excepciones derivadas de `Exception`.
- Útil para personalizar el manejo de errores en aplicaciones específicas.

Clase Base de Excepciones

```
public class MiExcepcion extends Exception {
    public MiExcepcion(String mensaje) {
        super(mensaje);
    }
}
```

Manejo de una Excepción Definida por el Usuario

```
public class Main {
    public static void main(String[] args) {
        try {
            lanzarMiExcepcion();
        } catch (MiExcepcion e) {
            System.out.println("Capturada MiExcepci
        }
    }

    public static void lanzarMiExcepcion() throws M
        throw new MiExcepcion("Esto es una excepcio
    }
}
```

Aserciones

- Las aserciones verifican condiciones en tiempo de ejecución y ayudan en la depuración.
- Se activan con la palabra clave `assert` para comprobar supuestos.
- Deben usarse solo para verificar errores de programación y no para manejar la lógica de la aplicación.

Aserciones - Ejemplo en Java

```
public class Aserciones {
    public static void main(String[] args) {
        int valor = -1;
        assert valor >= 0 : "Valor debe ser no nega
        System.out.println("Valor: " + valor);
    }
}
```

Conclusión

- Las excepciones permiten manejar errores en tiempo de ejecución sin detener el programa.
- La gestión adecuada mejora la robustez y la legibilidad del código.
- Las aserciones son útiles para la depuración, pero no deben usarse para la lógica del programa.

Introducción a la Manipulación del Sistema de Archivos

- Java proporciona clases en el paquete `java.io` y `java.nio.file` para trabajar con archivos y directorios.
- Estas clases permiten crear, eliminar, y modificar archivos y carpetas.

Ejemplo: Creación de un Archivo

```
import java.io.File;
import java.io.IOException;

public class FileExample {
    public static void main(String[] args) {
        File file = new File("example.txt");
        try {
            if (file.createNewFile()) {
                System.out.println("Archivo creado:
            } else {
                System.out.println("El archivo ya e
            }
        } catch (IOException e) {
            System.out.println("Error al crear el a
        }
    }
}
```

Ejemplo: Lectura de un Directorio

```
import java.io.File;

public class DirectoryExample {
    public static void main(String[] args) {
        File dir = new File("miDirectorio");
        if (dir.isDirectory()) {
            String[] archivos = dir.list();
            for (String archivo : archivos) {
                System.out.println(archivo);
            }
        } else {
            System.out.println("No es un directorio
        }
    }
}
```

Concepto de Flujos en Java

- Un **flujo** es una secuencia de datos que puede provenir o dirigirse a diferentes fuentes como archivos, red, entrada estándar, entre otros.
- Java utiliza el paquete `java.io` para el manejo de flujos.

Tipos de Flujos

- **Flujos de entrada (InputStream)**: Permiten leer datos de una fuente.
- **Flujos de salida (OutputStream)**: Permiten escribir datos a un destino.
- **Flujos de bytes**: Trabajan con datos en formato binario (e.g., `FileInputStream`).
- **Flujos de caracteres**: Trabajan con datos en formato de texto (e.g., `FileReader`).

Ejemplo: Lectura de un Archivo de Texto

```
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;

public class FileReaderExample {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader
            String linea;
            while ((linea = br.readLine()) != null)
                System.out.println(linea);
            }
        } catch (IOException e) {
            System.out.println("Error al leer el ar
        }
    }
}
```

Características de los Flujos

- Los flujos de datos en Java pueden ser:
  - **Secuenciales**: Se leen o escriben datos en el orden en que llegan.
  - **Bufferizados**: Permiten almacenar datos temporalmente para mejorar la eficiencia.
  - **Cadenas de Flujos**: Permiten encadenar múltiples flujos para realizar tareas de manera eficiente.

Ejemplo: Uso de Buffer para Lectura Eficiente

```
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;

public class BufferedStreamExample {
    public static void main(String[] args) {
        try (BufferedInputStream bis = new Buffered
            int byteData;
            while ((byteData = bis.read()) != -1) {
                System.out.print((char) byteData);
            }
        } catch (IOException e) {
            System.out.println("Error al leer el ar
        }
    }
}
```

Usos de los Flujos

- Lectura y escritura de archivos de texto y binarios.
- Comunicación en redes mediante flujos de entrada y salida.
- Transferencia de datos entre programas o procesos.

Archivos de Texto

- Un **archivo de texto** almacena datos en un formato legible, utilizando caracteres.
- Se utiliza principalmente para almacenar datos de configuración o documentos.

Ejemplo: Escritura en un Archivo de Texto

```
import java.io.FileWriter;
import java.io.IOException;

public class TextFileWriter {
    public static void main(String[] args) {
        try (FileWriter writer = new FileWriter("ou
            writer.write("Este es un ejemplo de arc
            writer.write("Java es muy vers til.");
        } catch (IOException e) {
            System.out.println("Error al escribir e
        }
    }
}
```

Archivos Binarios

- Un **archivo binario** almacena datos en formato binario, no legible directamente por humanos.
- Es útil para almacenar datos complejos como imágenes, videos, o archivos comprimidos.

Ejemplo: Escritura en un Archivo Binario

```
import java.io.FileOutputStream;
import java.io.IOException;

public class BinaryFileWriter {
    public static void main(String[] args) {
        byte[] data = {65, 66, 67, 68}; // Represen
        try (FileOutputStream fos = new FileOutputSt
            fos.write(data);
        } catch (IOException e) {
            System.out.println("Error al escribir e
        }
    }
}
```