Índice

Herencia

Concepto

La herencia es un mecanismo importante en la programación orientada a objetos que permite definir nuevas clases a partir de clases existentes. En Java, una clase puede heredar características (métodos y atributos) de otra clase a través de la herencia.

La clase que se está heredando se llama clase base o superclase, mientras que la clase que hereda se llama subclase o clase derivada.

La herencia en Java se define utilizando la palabra clave extends. Para heredar una clase, simplemente escribimos la palabra clave extends seguida del nombre de la clase base:

```
public class Subclase extends ClaseBase {
    // código de la subclase
}
```

Al utilizar la herencia en Java, la subclase hereda todos los miembros públicos y protegidos de la superclase. Los miembros privados de la superclase no son heredados, ya que no son accesibles fuera de la clase.

La subclase también puede agregar nuevos miembros y/o modificar los miembros heredados de la superclase. Si la subclase tiene un método con el mismo nombre, parámetros y tipo de retorno que un método en su clase padre, la subclase "anula" la implementación del método padre y proporciona su propia implementación.

La herencia en Java permite la creación de jerarquías de clases, en las cuales las clases más específicas heredan características de las clases más generales. Por ejemplo, en una jerarquía de clases Vehículo -> Automóvil -> Coche, la clase Coche hereda los métodos y atributos de Automóvil, que a su vez hereda los de Vehículo. De esta forma, se puede reutilizar el código de las clases más generales para crear nuevas clases más específicas.

Ejemplos

Superclase - Vehículo

```java
public class Vehiculo {
    private String marca;
    private String modelo;

    public Vehiculo(String marca, String modelo) {
        this.marca = marca;
        this.modelo = modelo;
    }

    public String getMarca() {
        return marca;
    }

    public String getModelo() {
        return modelo;
    }

    public void acelerar() {
        System.out.println("El vehículo está acelerando");
    }

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

Subclases- Coche/Moto

```java
public class Coche extends Vehiculo {
    private int numPuertas;

    public Coche(String marca, String modelo, int numPuertas) {
        super(marca, modelo);
        this.numPuertas = numPuertas;
    }

    public int getNumPuertas() {
        return numPuertas;
    }

    public void abrirMaletero() {
        System.out.println("Abriendo el maletero del coche");
    }
}
```

```java
public class Moto extends Vehiculo {
    private boolean llevaPasajero;

    public Moto(String marca, String modelo, boolean llevaPasajero) {
        super(marca, modelo);
        this.llevaPasajero = llevaPasajero;
    }

    public boolean isLlevaPasajero() {
        return llevaPasajero;
    }

    public void hacerCaballito() {
        System.out.println("Haciendo un caballito con la moto");
    }
}
```

Interfaz (Interface)

Concepto

En Java, una interfaz es una colección de métodos y constantes abstractos que se definen sin implementación. Las interfaces permiten definir un conjunto de métodos que una clase debe implementar, pero no especifican cómo se implementan. En lugar de proporcionar una implementación, las interfaces definen un contrato que una clase debe cumplir.

Las interfaces se definen utilizando la palabra clave "interface" en Java, seguida del nombre de la interfaz y una lista de métodos y constantes abstractos que define la interfaz.

Ejemplo

Este es un ejemplo de interfaz en Java que define métodos para operar con figuras geométricas:

Interfaz

```java
public interface FiguraGeometrica {
    public double calcularArea();
    public double calcularPerimetro();
}
```

Clases que implementan dicha interface:

```java
public class Cuadrado implements FiguraGeometrica {
    private double lado;

    public Cuadrado(double lado) {
        this.lado = lado;
    }

    public double calcularArea() {
        return lado * lado;
    }

    public double calcularPerimetro() {
        return 4 * lado;
    }
}
```

```java
public class Circulo implements FiguraGeometrica {
    private double radio;

    public Circulo(double radio) {
        this.radio = radio;
    }

    public double calcularArea() {
        return Math.PI * radio * radio;
    }

    public double calcularPerimetro() {
        return 2 * Math.PI * radio;
    }
}
```

En este caso, la clase "Cuadrado" y la clase "Circulo" implementan la interfaz "FiguraGeometrica" y proporcionan una implementación para los métodos "calcularArea" y "calcularPerimetro" para cada figura geométrica.

De esta manera, podrías usar estas clases en cualquier parte de tu aplicación que requiera el cálculo del área y el perímetro de figuras geométricas, sin importar qué tipo de figura geométrica sea. Esto hace que el código sea más flexible y fácil de mantener.

Clases Abstractas

Concepto

En Java, una clase abstracta es una clase que no puede ser instanciada directamente, sino que se utiliza como una clase base para otras clases. Las clases abstractas se utilizan para definir la estructura y el comportamiento común a un conjunto de clases relacionadas.

Para definir una clase abstracta en Java, se utiliza la palabra clave "abstract" en la definición de la clase. Por ejemplo:

Ejemplo

Clase Abstracta

```java
public abstract class Animal {
    private String nombre;

    public Animal(String nombre) {
        this.nombre = nombre;
    }

    public String getNombre() {
        return nombre;
    }

    public abstract void hacerSonido();
}
```

En este ejemplo, la clase "Animal" es una clase abstracta que define un atributo "nombre" y un método abstracto "hacerSonido()". El método "hacerSonido()" no tiene implementación, lo que significa que cualquier clase que extienda la clase "Animal" debe proporcionar su propia implementación para este método.

Luego, podrías crear varias clases que extiendan la clase "Animal" y proporcionen una implementación para el método "hacerSonido()". Por ejemplo:

Clases que extienden de la clase abstracta

```
public class Perro extends Animal {
    public Perro(String nombre) {
        super(nombre);
    }

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

public class Gato extends Animal {
    public Gato(String nombre) {
        super(nombre);
    }

    public void hacerSonido() {
        System.out.println("Miau");
    }
}
```

Recursividad

Concepto

La recursividad en Java es un concepto que se refiere a la capacidad de una función de llamarse a sí misma, lo que puede resultar muy útil en ciertas situaciones, especialmente en casos donde una solución se puede dividir en varios subproblemas idénticos más pequeños.

Ejemplos

Factorial

El factorial de un número se define como el producto de todos los números enteros positivos desde 1 hasta ese número. Por ejemplo, el factorial de 5 (escrito como 5!) es 5 x 4 x 3 x 2 x 1 = 120.

```java
public static int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Fibonacci

La secuencia de Fibonacci es una serie de números en la que cada número es la suma de los dos números anteriores. Por ejemplo, los primeros 10 números de la serie de Fibonacci son: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34.

```java
public static int fibonacci(int n) {
    if (n <= 1) {
        return n;
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

Sobrecarga (Overload)

Concepto

La sobrecarga de métodos en Java se refiere a la capacidad de definir múltiples métodos en una clase con el mismo nombre pero diferentes parámetros. Cada método sobrecargado realiza una operación similar, pero con diferentes tipos y/o cantidades de parámetros.

En la sobrecarga de métodos, los métodos deben tener el mismo nombre, pero deben tener diferentes tipos y/o cantidades de parámetros. Los métodos también pueden tener diferentes modificadores de acceso (como public, private, protected) y valores de retorno, pero el nombre y la firma de los parámetros deben ser diferentes.

Ejemplos

Suma

Aquí se muestra una clase de ejemplo con métodos con el mismo nombre pero diferente firma en sus parámetros.

```java
public class Ejemplo {
    public static int suma(int x, int y) {
        return x + y;
    }

    public static double suma(double x, double y) {
        return x + y;
    }

    public static String suma(String x, String y) {
        return x + y;
    }
}
```

Constructores de una clase

La sobrecarga también se produce cuando en una clase objeto realizamos varias implementaciones del constructor de la clase, recibiendo en este caso diferentes parámetros.

```java
public class Persona {
    private String nombre;
    private int edad;
    private String ocupacion;

    // Constructor por defecto
    public Persona() {
        this.nombre = "Desconocido";
        this.edad = 0;
        this.ocupacion = "Desempleado";
    }

    // Constructor con nombre y edad
    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
        this.ocupacion = "Desempleado";
    }

    // Constructor con nombre, edad y ocupación
    public Persona(String nombre, int edad, String ocupacion) {
        this.nombre = nombre;
        this.edad = edad;
        this.ocupacion = ocupacion;
    }
```

Sobreescritura (Override)

Concepto

La sobreescritura de métodos en Java es un mecanismo mediante el cual una clase hija proporciona su propia implementación de un método que ya se ha definido en la clase padre. Es decir, cuando una subclase tiene un método con el mismo nombre, parámetros y tipo de retorno que un método en su clase padre, la subclase "anula" la implementación del método padre y proporciona su propia implementación.

La sobreescritura de métodos es importante en la programación orientada a objetos porque nos permite definir comportamientos específicos de la subclase para métodos heredados de la superclase. La subclase puede cambiar el comportamiento del método padre para adaptarse a sus propias necesidades.

La sobreescritura de métodos se realiza en la subclase utilizando la misma firma de método que el método que se está sobrescribiendo en la superclase. La firma de método incluye el nombre del método, el número y tipos de parámetros y el tipo de retorno. La anotación @Override se puede utilizar para indicar que se está sobrescribiendo un método.

Ejemplo

En este ejemplo, la clase Perro hereda de la clase Animal y sobrescribe el método hacerSonido(). Cuando llamamos al método hacerSonido() en un objeto de la clase Perro, se ejecutará la implementación del método de la clase Perro, que imprime "Ladrando" en la consola en lugar de la implementación de la clase Animal que imprime "Haciendo un sonido genérico". La anotación @Override se utiliza para indicar que se está sobrescribiendo el método hacerSonido() de la clase Animal.

```java
public class Animal {
    public void hacerSonido() {
        System.out.println("Haciendo un sonido genérico");
    }
}

public class Perro extends Animal {
    @Override
    public void hacerSonido() {
        System.out.println("Ladrando");
    }
}
```

Instanciación

Concepto

En Java, una instancia es un objeto creado a partir de una clase. Cada vez que se crea una instancia de una clase, se crea un objeto separado con su propio conjunto de atributos y métodos.

Ejemplo

Por ejemplo, si tenemos la siguiente clase en Java:

```java
public class Persona {
    private String nombre;
    private int edad;

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    public String getNombre() {
        return nombre;
    }

    public int getEdad() {
        return edad;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public void setEdad(int edad) {
        this.edad = edad;
    }
}
```

Podemos generar las siguientes instancias:

```java
Persona persona1 = new Persona("Juan", 25);
Persona persona2 = new Persona("Ana", 30);
```

En este caso, hemos creado dos instancias de la clase "Persona" llamadas "persona1" y "persona2". Cada una de estas instancias es un objeto separado con su propio conjunto de atributos y métodos. Podemos acceder a los atributos y métodos de cada instancia.

Excepciones

Concepto

En Java, una excepción es un evento que se produce durante la ejecución de un programa que interrumpe el flujo normal de ejecución y se propaga hacia arriba a través de la pila de llamadas de métodos hasta que se maneja o termina el programa.

Tipos

Existen dos tipos de excepciones en Java: excepciones comprobadas y excepciones no comprobadas.

- **Excepciones comprobadas**: son excepciones que se deben manejar explícitamente en el código. Esto significa que si un método puede lanzar una excepción comprobada, el código que llama a ese método debe manejar esa excepción o lanzarla hacia arriba para que sea manejada por un código que lo llame. Las excepciones comprobadas se extienden de la clase "Exception".

- **Excepciones no comprobadas**: son excepciones que no se requieren manejar explícitamente en el código. Las excepciones no comprobadas se extienden de la clase "RuntimeException". Estas excepciones pueden ser lanzadas en cualquier momento durante la ejecución de un programa, y no es necesario que se manejen en el código que llama al método que las lanza.

Ejemplos

A continuación se presentan algunos ejemplos de excepciones más comunes en Java:

- **NullPointerException**: Esta excepción se lanza cuando se intenta acceder a un objeto nulo. Por ejemplo, si intentamos llamar a un método de un objeto que es nulo, se lanzará una excepción "NullPointerException".
- **ArrayIndexOutOfBoundsException**: Esta excepción se lanza cuando se intenta acceder a un índice fuera de los límites de un array. Por ejemplo, si intentamos acceder al elemento de un array con un índice mayor que el número de elementos en el array, se lanzará una excepción "ArrayIndexOutOfBoundsException".
- **ArithmeticException**: Esta excepción se lanza cuando se produce una operación aritmética inválida, como la división por cero.
- **IOException**: Esta excepción se lanza cuando ocurre algún problema de entrada/salida, como cuando no se puede acceder a un archivo o cuando se produce un error durante la lectura o escritura de datos.
- **ClassNotFoundException**: Esta excepción se lanza cuando una clase no se puede encontrar durante la ejecución del programa.
- **SQLException**: Esta excepción se lanza cuando se produce un error en la ejecución de una sentencia SQL.

Control de Excepciones

En Java, las excepciones pueden ser controladas mediante el uso de bloques "try-catch". Un bloque "try" se utiliza para envolver el código que puede lanzar una excepción, mientras que un bloque "catch" se utiliza para manejar la excepción lanzada por el bloque "try".

La sintaxis básica de un bloque "try-catch" es la siguiente:

```
try {
    // Código que puede lanzar una excepción
} catch (TipoDeExcepcion nombreDeLaExcepcion) {
    // Código para manejar la excepción
}
```

En el bloque "try", se coloca el código que puede lanzar una excepción. Si se lanza una excepción, se interrumpe la ejecución del código en el bloque "try" y se pasa al bloque "catch". En el bloque "catch", se especifica el tipo de excepción que se quiere manejar, seguido del nombre de la variable que se utilizará para hacer referencia a la excepción lanzada.

Por ejemplo, si queremos manejar una excepción "ArithmeticException" que se puede lanzar al realizar una operación de división, el código se vería así:

```
try {
    int resultado = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("Se ha producido una excepción: " + e.getMessage());
}
```

En este ejemplo, el bloque "try" intenta dividir 10 entre 0, lo que lanzaría una excepción "ArithmeticException". En el bloque "catch", se maneja la excepción imprimiendo un mensaje en la consola que indica que se ha producido una excepción y mostrando el mensaje de la excepción lanzada.

Es importante manejar adecuadamente las excepciones en el código para garantizar que el programa se ejecute de manera segura y eficiente.