Polymorphism is the ability of an object to take on many forms. In Java, it's a fundamental concept of object-oriented programming (OOP) that allows you to treat objects of different classes that are related by inheritance through a common superclass. This means a single variable can be used to refer to objects of different subclasses. There are two main types of polymorphism in Java: **compile-time** and **runtime** polymorphism.

**Compile-Time Polymorphism (Method Overloading)**

Compile-time polymorphism is also known as **static polymorphism**. It is achieved through **method overloading**, where multiple methods in the same class have the same name but different parameters. The compiler determines which method to call based on the number, type, and order of the arguments passed at compile time.

**Key characteristics:**

* **Same method name.**
* **Different parameter lists** (number, type, or order of parameters).
* **Return type can be different**, but it's not a condition for overloading.
* The decision of which method to call is made at compile time.

Java

class Calculator {

// Overloaded method to add two integers

public int add(int a, int b) {

return a + b;

}

// Overloaded method to add two doubles

public double add(double a, double b) {

return a + b;

}

}

// Example usage

Calculator calc = new Calculator();

System.out.println(calc.add(5, 10)); // Calls the int method

System.out.println(calc.add(5.5, 10.5)); // Calls the double method

**Runtime Polymorphism (Method Overriding)**

Runtime polymorphism, or **dynamic polymorphism**, is achieved through **method overriding**. This is where a subclass provides a specific implementation for a method that is already defined in its superclass. The decision of which method to call is made at runtime, based on the actual object type. This is the more common and powerful form of polymorphism.

**Key characteristics:**

* **Inheritance is required.**
* The method in the subclass must have the **exact same name, parameters, and return type** as the method in the superclass.
* The **@Override** annotation is often used for clarity and to ensure the compiler checks the method signature.
* The decision of which method to execute is made at runtime.

**The "IS-A" Relationship**

Runtime polymorphism relies on the "is-a" relationship, a core concept in inheritance. For example, a **Cat** is an **Animal**, and a **Dog** is an **Animal**. This allows you to create an **Animal** variable and assign it a **Cat** or **Dog** object.

Java

// Superclass

class Animal {

public void makeSound() {

System.out.println("The animal makes a sound.");

}

}

// Subclass 1

class Dog extends Animal {

@Override

public void makeSound() {

System.out.println("The dog barks.");

}

}

// Subclass 2

class Cat extends Animal {

@Override

public void makeSound() {

System.out.println("The cat meows.");

}

}

// Example usage demonstrating runtime polymorphism

public class Zoo {

public static void main(String[] args) {

Animal myDog = new Dog(); // myDog is a Dog, but is referred to as an Animal

Animal myCat = new Cat(); // myCat is a Cat, but is referred to as an Animal

myDog.makeSound(); // At runtime, the JVM calls the Dog's makeSound() method

myCat.makeSound(); // At runtime, the JVM calls the Cat's makeSound() method

}

}

In the example above, even though myDog and myCat are declared as type **Animal**, the correct makeSound() method is invoked at runtime based on the actual object they refer to (Dog and Cat). This flexibility is the essence of runtime polymorphism.

Sources

Teach me polymorphism java with example please?

Polymorphism is an object-oriented programming (OOP) concept that allows you to treat objects of different classes that are related through inheritance with a common superclass. The term literally means "many forms." In Java, there are two types of polymorphism: **compile-time** and **runtime**.

**Compile-Time Polymorphism (Method Overloading)**

Compile-time polymorphism is also known as static polymorphism. It's achieved when a class has multiple methods with the same name but different parameters. The compiler decides which method to call at compile time based on the arguments provided.

**Example: Method Overloading**

Java

class Calculator {

// Method to add two integers

public int add(int a, int b) {

return a + b;

}

// Overloaded method to add three integers

public int add(int a, int b, int c) {

return a + b + c;

}

// Another overloaded method to add two doubles

public double add(double a, double b) {

return a + b;

}

}

public class Main {

public static void main(String[] args) {

Calculator calc = new Calculator();

System.out.println(calc.add(2, 3)); // Calls the first method

System.out.println(calc.add(2, 3, 4)); // Calls the second method

System.out.println(calc.add(2.5, 3.5)); // Calls the third method

}

}

In this example, the Calculator class has three add methods. The compiler determines which one to use by checking the number and type of arguments passed.

**Runtime Polymorphism (Method Overriding)**

Runtime polymorphism, or dynamic polymorphism, is achieved through **method overriding**. This occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. The decision of which method to call is made at runtime, not compile time. This is only possible with an "is-a" relationship, where a subclass is a type of its superclass (e.g., a **Dog** is an **Animal**).

**Example: Method Overriding**

Java

// Superclass

class Animal {

public void makeSound() {

System.out.println("The animal makes a sound.");

}

}

// Subclass

class Dog extends Animal {

// This method overrides the makeSound() method of the superclass

@Override

public void makeSound() {

System.out.println("The dog barks.");

}

}

// Subclass

class Cat extends Animal {

@Override

public void makeSound() {

System.out.println("The cat meows.");

}

}

public class Zoo {

public static void main(String[] args) {

// Here we create objects of type Dog and Cat, but refer to them as Animal

Animal myDog = new Dog();

Animal myCat = new Cat();

// At runtime, the JVM knows the actual object type and calls the correct method

myDog.makeSound(); // Output: The dog barks.

myCat.makeSound(); // Output: The cat meows.

}

}

Here, myDog and myCat are declared as type **Animal**, but they hold references to **Dog** and **Cat** objects, respectively. At runtime, the Java Virtual Machine (JVM) looks at the actual object and calls the overridden makeSound() method from the Dog or Cat class, showcasing the "many forms" of the makeSound() method. .