Polymorphism with Examples

The word 'polymorphism' means 'having many forms'. In Java, polymorphism refers to the ability of a message to be displayed in more than one form. This concept is a key feature of Object-Oriented Programming and it allows objects to behave differently based on their specific class type.

Types of Polymorphism in Java

There are two types of polymorphism in Java:

1. Compile-time polymorphism
2. Runtime polymorphism.

1. Compile-Time (Static) Polymorphism

Compile-time polymorphism is achieved through method overloading. Method overloading occurs when two or more methods have the same name but differ in either the number or types of their parameters. This allows us to perform different operations with the same method name, without needing to remember different function names like `sum1()`, `sum2()`, etc.

In Java, when a method is called, the compiler determines which version of the method to invoke based on the number or type of arguments passed. This is resolved at compile time, hence the name compile-time polymorphism.

Example of Method Overloading:

```
class Sum {
    public int sum(int x, int y) {
        return (x + y);
```

```
    }

    public int sum(int x, int y, int z) {
        return (x + y + z);
    }

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

    public static void main(String[] args) {
        Sum obj = new Sum();
        System.out.println(obj.sum(10, 20));
        System.out.println(obj.sum(10.5, 20.5));
    }
}
```

Output:

```
30

31.0
```

In this example, the same method name `sum()` is used for different parameter combinations (two integers, three integers, or two doubles). The compiler determines the appropriate method based on the input.

2. Runtime (Dynamic) Polymorphism

Runtime polymorphism is achieved through method overriding, which is used when a method in a subclass has the same name, return type, and parameters as the method in the superclass.

Here, the decision of which method to invoke is made at runtime based on the object that is being referred to.

For instance, if you have a base class `Employee` with a method `printDetails()` and a subclass `SoftwareDeveloper`that also defines its own `printDetails()` method, the JVM will determine which version of the method to call based on the actual object (whether it is an `Employee` or `SoftwareDeveloper`) at runtime.

Key Points About Method Overloading in Java

1. Method Overloading Criteria:
    - Overloading is achieved by changing the number of parameters or the data types of parameters. You can also use a mix of both.
    - Method overloading cannot be done by changing only the return type.
2. Automatic Type Conversion: If a method is called with an argument type that doesn't exactly match, Java may automatically convert the type (if possible) to match the method's signature. For example, if an `int` is passed to a method expecting a `long`, Java will convert it automatically.
3. Ambiguity Avoidance: Java does not allow method overloading based solely on the return type because it could lead to ambiguity. For example, if two methods have the same signature but different return types, the compiler would not be able to decide which method to invoke.
4. Operator Overloading (Not Allowed in Java): While languages like C++ allow operator overloading, Java does not support it. However, Java internally overloads operators like the `+` operator for string concatenation. For example:

```
String s1 = "Hello ";
String s2 = "World!";
System.out.println(s1 + s2);  // Output: Hello World!
```