polymorphism is a fundamental concept that allows objects of different classes to be treated as objects of a common superclass. There are two main types of polymorphism in Java: compile-time (static) polymorphism and runtime (dynamic) polymorphism. Let's look at examples of each type:

1. **Compile-Time Polymorphism (Method Overloading)**:

Compile-time polymorphism occurs when the decision about which method to call is made at compile time based on the method signature (the method name and the parameters). Method overloading is a common form of compile-time polymorphism, where a class has multiple methods with the same name but different parameters.

```java
class Calculator {
    int add(int a, int b) {
        return a + b;
    }

    double add(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calculator = new Calculator();
        System.out.println(calculator.add(5, 7));        // Calls the int version
        System.out.println(calculator.add(3.2, 4.8));    // Calls the double version
    }
}
```

In this example, the `Calculator` class has two `add` methods, one that takes two integers and another that takes two doubles. The appropriate method is called based on the argument types at compile time.

2. **Runtime Polymorphism (Method Overriding)**:

Runtime polymorphism occurs when the decision about which method to call is made at runtime based on the actual type of the object. Method overriding is a form of runtime polymorphism, where a subclass provides a specific implementation of a method that is already defined in the superclass.

```java
```

```
class Animal {
    void makeSound() {
        System.out.println("Some sound");
    }
}

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

class Cat extends Animal {
    @Override
    void makeSound() {
        System.out.println("Meow");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Dog();
        myAnimal.makeSound();  // Calls Dog's makeSound method at runtime

        myAnimal = new Cat();
        myAnimal.makeSound();  // Calls Cat's makeSound method at runtime
    }
}
```

In this example, we have a hierarchy of classes (`Animal`, `Dog`, and `Cat`). The `makeSound` method is overridden in the `Dog` and `Cat` subclasses. At runtime, the actual type of the object determines which version of `makeSound` is called when you invoke it.

These examples illustrate both compile-time and runtime polymorphism in Java. Compile-time polymorphism is achieved through method overloading, while runtime polymorphism is achieved through method overriding. Both forms of polymorphism contribute to the flexibility and extensibility of object-oriented programs.