Polymorphism in Java: Compile-Time vs Runtime

Quick Recap:

Method Overloading: Same method name, different parameters, in the same class.

Method Overriding: Same method name, same parameters, but in subclass.

Why Overloading is Compile-Time Polymorphism:

In method overloading, the method to be called is determined by the compiler based on the number and types of arguments.

Example:

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

At compile time:

```
Calculator c = new Calculator();
c.add(2, 3);       // Compiler chooses int add(int, int)
c.add(2.0, 3.0);   // Compiler chooses double add(double, double)
```

So, it's called compile-time polymorphism.

Why Overriding is Runtime Polymorphism:

In method overriding, the method to be executed is decided at runtime based on the object type, not the reference type.

Example:

```
class Animal {
    void sound() { System.out.println("Animal sound"); }
}
class Dog extends Animal {
    void sound() { System.out.println("Dog barks"); }
}
```

At runtime:

Animal a = new Dog();

a.sound(); // Output: Dog barks

The JVM resolves the method call based on the actual object type at runtime.

Summary Table:

Feature	Method Overloading	Method Overriding
Decided at	Compile-time	Runtime
Method Signature	Must differ	Must be same
Class Location	Same class	Parent-child relationship
Polymorphism Type	Compile-time polymorphism	Runtime polymorphism
Example	sum(int, int) vs sum(int, int, int)	Animal.sound() vs Dog.sound()