Overloading

JUNE 2, 2015

Reusing the same method name in the same class or subclass but with different arguments (and optionally, a different return type) is called **Method Overloading in Java**.

There are certain **rules for overloading**, the below code points out all of the rules:

```java
public class Foo {
    public void doStuff(int y, String s) { }

    public void doStuff(int x) { }  // Valid overload as DIFFERENT ARGUMENT LIST
                                    // (and methods can be overloaded
                                    // in the same class or in subclass)
}

class Bar extends Foo {

    private void doStuff(long a) {  // Valid overload as DIFFERENT ARGUMENT LIST
                                    // (access modifier CAN be same or different)
    }

    public String doStuff(int x, int y) { // Valid overload as DIFFERENT ARGUMENT LIST
        return "s";                       // (return type CAN be same or different)
    }

    public void doStuff(int y, long s) throws IOException { } // Valid method overload as DI
                                                              // (overloaded methods CAN dec
                                                              // checked exceptions)

    public String doStuff(int y, String s) { // Invalid overload, MUST change method's
        return "s";                          // argument list (compiler error)
    }

    public static void doStuff(int x) { }  // Invalid overload, MUST change method's
                                           // argument list (compiler error)
}
```

In short, the only rule you **MUST** obey is to **change the argument list** of the overloaded method, the rest are all optional.

Invoking Overloaded Methods

```java
public class Animal {
}

class Horse extends Animal {
}

class UseAnimals {
    public void doStuff(Animal a) {
        System.out.println("In the Animal version");
    }

    public void doStuff(Horse h) {
        System.out.println("In the Horse version");
    }

    public static void main(String[] args) {
        UseAnimals ua = new UseAnimals();
        Animal animalObj = new Animal();
        Horse horseObj = new Horse();
        Animal animalRefToHorse = new Horse();

        ua.doStuff(animalObj);
        ua.doStuff(horseObj);
        ua.doStuff(animalRefToHorse);
    }
}
```

The output of the above program is:

```
In the Animal version
In the Horse version
In the Animal version
```

Notice the call `doStuff(animalRefToHorse)`, here the `Animal` version of `doStuff()` is called despite the actual object being passed is of a `Horse`. The **reference type** (not the object type) determines which overloaded method is invoked.

To summarize, **which overridden version** of the method to call (in other words, from which class in the inheritance tree) is decided **at runtime based on object type**, but **which overloaded version** of the method to call is based on the **reference type of the argument passed at compile time**.

Therefore, **polymorphism doesn’t determine which overloaded version is called, polymorphism does come into play when the decision is about which overridden version of a method is called**.

The confounding part in method overloading

I will show you different scenarios after which you can answer any question related to method matching (which method will be invoked).

The 3 factors that can make overloading a little tricky *(written in order of preference)*:

- Widening
- Autoboxing
- Var-args

```
public class MethodOverloading {

    static void go(float x) {
        System.out.print("float ");
    }

    static void go(Long x) {
        System.out.print("Long ");
    }

    static void go(double x) {
        System.out.print("double ");
    }

    static void go(Double x) {
        System.out.print("Double ");
    }

    static void go(int x, int y) {
        System.out.print("int,int ");
    }

    static void go(byte... x) {
        System.out.print("byte... ");
    }

    static void go(Long x, Long y) {
        System.out.print("Long,Long ");
    }

    static void go(long... x) {
        System.out.print("long... ");
    }

    public static void main(String[] args) {
        byte b = 5;
        short s = 5;
        long l = 5;
        float f = 5.0f;
        // widening beats autoboxing
        go(b);
        go(s);
        go(l);
        go(f);
        // widening beats var-args
        go(b, b);
        // auto-boxing beats var-args
        go(l, l);
    }
}
```

The output of the above program is `float float float float int,int Long,Long`.

From the output it is clear that the JVM prefers **widening over autoboxing and var-args**. In every case, when an exact match isn't found, the JVM uses the method with the smallest argument that is wider than the parameter.

Now look at the call `go(l, l)` at last, this invokes `go(Long x, Long y)` rather than `go(long... x)` which clearly shows JVM prefers **boxing over var-args**.

Widening reference variables

Reference widening depends on inheritance, in other words if it passes the *IS-A* test then no harm. Consider the below code:

```java
public class Animal {
    static void eat() {
    }
}

class Dog extends Animal {

    void go(Animal a) { }

    public static void main(String[] args) {
        Dog d = new Dog();
        d.go(d);
    }
}
```

The above code compiles fine as `Dog` can widen into an `Animal` because it passes the *IS-A* test. If in case, `Dog` didn't have extended `Animal` then widening wouldn't be possible and the code wouldn't compile.

Similarly, you **cannot** widen `Integer` to `Long` but you **can** widen `int` to `long`.

Combine Widening with Boxing

Let's see what happens when the compiler has to widen and then autobox the parameter for a match to be made.

```java
public class WidenAndBox {
    static void go(Long x) {
        System.out.println("Long");
    }

    public static void main(String[] args) {
        byte b = 5;
        go(b); // compiler has to widen to `long` and
               // then box to `Long`
    }
}
```

The above program fails to compile, the JVM **does not widen and then box**. It may be because widening existed in the earlier versions of Java and it wanted a method that is invoked via widening shouldn't lose out to a newly created method that relies on boxing. In other words, Java designers wanted the preexisting code should function the way it used to.

Now imagine if JVM tried to box first, the `byte` would have been converted to a `Byte`. Now we're back to trying to widen a `Byte` to a `Long`, and of course, the *IS-A* test fails.

So both of the ways didn't work.

Now let's see another program when the compiler has to autobox and then widen the parameter for a proper match.

```java
public class BoxAndWiden {
    static void go(Object obj) {
        Byte b2 = (Byte) obj;    // ok - obj refers to a Byte object
        System.out.println(b2);
    }

    public static void main(String[] args) {
        byte b = 5;
        go(b);
    }
}
```

The above code compiles and produces the output 5. Firstly, the `byte b` was boxed to a `Byte`. And then the `Byte` reference was widened to an `Object` (since `Byte` extends `Object`). So, the `go()` method got an `Object` reference that actually refers to a `Byte` object.

From the above 2 examples its certain that the JVM **can never widen and then box** but **can box and then widen**.

Combine both Widening and Boxing with Var-args

```java
public class Vararg {

    static void wide_vararg(long... x) {
        System.out.println("long...");
    }

    static void box_vararg(Integer... x) {
        System.out.println("Integer...");
    }

    static void box_widen_vararg(Object... x) {
        System.out.println("Object...");
    }

    public static void main(String[] args) {
        int i = 5;
        wide_vararg(i, i);  // needs to widen and use var-args
        box_vararg(i, i);   // needs to box and use var-args
        box_widen_vararg(i, i); // needs to box and then widen and finally use var-args
    }
}
```

The above code compiles fine and produces the output:

```
long...
Integer...
Object...
```

From the result, its clear that we can **successfully combine var-args with either widening or boxing**.

Q&A

Q1. Consider the below program in which a method is both overridden and overloaded.

```java
public class Animal {
    public void eat() {
        System.out.println("Animal eating");
    }
}

class Horse extends Animal {
    public void eat() {
        System.out.println("Horse eating hay ");
    }

    public void eat(String s) {
        System.out.println("Horse eating " + s);
    }
}
```

Figure out which version of `eat()` will run on each of the invocation made?

```
A. Animal ah = new Horse();
   ah.eat();
B. Horse he = new Horse();
   he.eat("Apples");
C. Animal a2 = new Animal();
   a2.eat("treats");
D. Animal ah2 = new Horse();
   ah2.eat("Carrots");
```

Overriding

« Previous

Constructors

Next »

Anagh Hegde • a year ago

```
public class MyTest {

public static void main(String[] args) {
MyTest test = new MyTest();
int i = 9;
test.TestOverLoad(i);
}

void TestOverLoad(String a){
System.out.println(8);
}

void TestOverLoad(Object a){
```

```
System.out.println(10);
}

}
```

o/p: 10

see more

Reply • Share ›