Theory: Overloading

15 minutes 5 / 15 problems solved

Overloading allows you to change the method's signature: the number of parameters, their type or both. If methods have the same name, but a different number or type of parameters, they are **overloaded**. It means you can invoke different methods by the same name by passing different arguments.

§1. How to overload methods

As an example, let's consider some overloaded method from the standard class `Math`:

```
public static int abs(int a) { return (a < 0) ? -a : a; }

public static float abs(float a) { return (a <= 0.0F) ? 0.0F - a : a; }
```

These methods have the same name but different type of the argument. They are overloaded.

> **Important** that it's impossible to declare more than one method with the same name and parameters (number and types), even with different return types. The return type is not considered for overloading because it's not a part of the signature.

Here are four methods `print` for printing different values.

```
public static void print(String stringToPrint) {
    System.out.println(stringToPrint);
}

public static void print(String stringToPrint, int times) {
    for (int i = 0; i < times; i++) {
        System.out.println(stringToPrint);
    }
}



public static void print(int times, String stringToPrint) {

    for (int i = 0; i < times; i++) {

        System.out.println(stringToPrint);

    }

}



public static void print(int val) {

    System.out.println(val);

}
```

The first method prints an input string, the second and the third ones print an input string a given number of times, and the last one prints an integer value. These methods are overloaded.

Let's invoke these methods:

```
print("some string");
print("another string", 2);
print(2, "another string again");
print(5);
```

Current topic:

Overloading

Topic depends on:

Declaring a method Stage 3

Topic is required for:

Multiple constructors

Table of contents:

↑ Overloading

§1. How to overload methods

§2. Overloading and casting

§3. Conclusion

Feedback & Comments

As you can see, it's possible to call any of these methods by the same name passing suitable arguments. The code outputs:

```
some string
another string
another string
another string again
another string again
5
```

Note, that in the case when parameters have different types, changing the order of these parameters is a valid case of overloading, as in the second and third methods from the example above.

The overloading mechanism allows us not to write different names for methods that perform similar operations.

Looking ahead, we'll assume that overloading is a form of the static (compile-time) polymorphism.

§2. Overloading and casting

To understand how overloading deals with type casting, let's consider an example of overloaded methods that only differ in the type on the single argument and see when each of them will be invoked and why.

```
public class OverloadingExample {

    public static void print(short a) {
        System.out.println("short arg: " + a);
    }

    public static void print(int a) {
        System.out.println("int arg: " + a);
    }



    public static void print(long a) {

        System.out.println("long arg: " + a);

    }



    public static void print(double a) {

        System.out.println("double arg: " + a);

    }



    public static void main(String[] args) {

        print(100);

    }

}
```

Now if we call `print(100)`, the program outputs:

```
int arg: 100
```

What we see here is that 100 is treated as `int` and the corresponding method is invoked.

In the case where the type of a method parameter is not exactly the same as the type of the passed argument, the compiler chooses the method that has the closest type of the argument in order of the implicit casting.

Since all integer literals are treated as `int` by default, `int` will be the starting point. The closest one will then be `long`.

Let's remove or comment the method `public static void print(int a)`, then recompile and run the program again. The result is as expected:

```
long arg: 100
```

Ok, now, let's remove the method `public static void print(long a)` too. Since we have no method with `float` argument, the next type in the order of implicit type casting will be `double`. After recompiling the program outputs:

```
double arg: 100.0
```

If we remove the method `public static void print(double a)` the only method we have left is the one with `short` type of argument. The program won't compile if we just call `print(100)` as we did before.

Let's explain why. When we pass some value to the method, the compiler does not evaluate it. All that is known is that it is integer literal and hence has integer type.

In our case, since 100 is treated as an `int` by default and JVM doesn't know if the passed value can be cast to `short` safely, the only way to pass `short` argument is by casting the value explicitly:

```
public class OverloadingExample {

    public static void print(short a) {
        System.out.println("short arg: " + a);
    }

    public static void main(String[] args) {
        print((short) 100);  // explicit casting
    }

}
```

§3. Conclusion

Method overloading allows you to implement two or more methods with the same name, but different arguments. The arguments of such methods may differ in their number or type. This helps to avoid having various method references for similar tasks. When invoked, the proper method is chosen based on the provided arguments. If the argument has a different type from what is expected, the closest type of the argument in order of the implicit casting is used.

Report a typo

525 users liked this theory. **9** didn't like it. **What about you?**

Start practicing

Comments (14) Hints (2) Useful links (0) Show discussion