6-methods-and-functions

Contents

- Methods
  - Overloading
  - Invocation
  - Returning
  - Arguments

Methods

Overloading

Method overloading is also called `compile time polymorphism`, this is so because we define the methods and their behavior at compile time (for the most part that is mostly true). These methods must have the same name, but they may not share the same type or/and number of arguments, the return type can be the same. The idea is that at runtime the compiler will choose the correct implementation of the overloaded method, and invoke that one.

```
class OverloadDemo {
    void test() {
        System.out.println("No parameters");
    }
    void test(int a) {
        System.out.println("a: " + a);
    }
    void test(int a, int b) {
        System.out.println("a and b: " + a + " " + b);
    }
    double test(double a) {
        System.out.println("double a: " + a);
        return a*a;
    }
}
```

Below is an example of how we may call the overloaded methods, which at runtime will be routed to the correct implementation of the method `test`

```
OverloadDemo ob = new OverloadDemo();
ob.test();
ob.test(10);
ob.test(10, 20);
ob.test(123.25);
```

Note that there are `caveats`, from the implementation above we can see that, we are correctly calling the test method, without parameters, with one int or two int parameters or one double. However what would happen if we instead called the test method with two double paramters, or with a float one.

```
OverloadDemo ob = new OverloadDemo();
ob.test(10.0, 20.0); // compile time error, up-cast not possible
ob.test(123.25f); // that is fine, automatic promotion
```

In the example above, the first call will result in a compile time error, because there is no overloaded method which takes two doubles as arguments, however the call below will be just fine, even though we do not have a float overload, the type system and the java compiler would perform automatic promotion of the type casting it from float to double, and it will invoke `test(double)`

Constructors can also be overloaded, there are no special requirements for them, the rules which apply for normal method overloading also apply for constructors, the name is kept the same and the permutation of number and type of arguments must be unique, otherwise the compiler would complain. Type promotion also may occur for constructor overloaded methods.

Invocation

Briefly, the way java works is that all arguments are passed by value, meaning that a copy of the argument is created and pushed on the stack before the method is invoked, however, there are some caveats, since objects are a kind of reference, when we pass object to a function as parameter which the `pointer` or `reference` itself is copied, the actual value of the `reference` still points to the original object, so we end up with an parameter that points to the same object, even though the actual pass is by-value. For other primitive types such as numbers, boolean and char there are no special cases, they are simply copied by-value as well.

Returning

Returning from methods is also done by-value, meaning that the returned type is copied back onto the call stack, the same caveats for object references is valid here, even though the reference itself is copied, the value remains the same meaning that the resulting object points to the same object on the heap. We are also allowed to return `new Type()` from methods, and do not have to worry about going out of scope, and the garbage collector cleaning up the created object, as long as at least one reference to that object exists in our program

```
Box createBox() {
    return new Box();
}
Box b = createBox(); // b will still point to the new box, therefore the
   collector can not forcefully collect/sweep it
createBox(); // the result is not used, meaning that no reference point to
    the created box, eligible for collection
```

Arguments

One particularly important feature in java is variable arguments list, which can be passed to a method, to signify that the method takes an unkown, possibly 0 number of arguments to operate on. The variable number

of arguments are defined with the `...` notation, and it is important to know that they have to be of the same type.

```
    void vamethod(int ... v)

    vamethod(10); // 1 arg
    vamethod(1, 2, 3); // 3 args
    vamethod(); // no args
```

The way this works internally is that the compiler would implicitly define `v` the variable argument passed to the function as an array, and thus inside the function we can iterate over these arguments, we have access to the `.length` property of the array, but in effect the method with a variable argument becomes something like `vamethod(int[] v)`

`A method can have "normal" parameters along with a variable-length parameter. However, the variable-length parameter must be the last parameter declared by the method. Attempt to declare a second varargs type parameter is not allowed`

Similarly to other normal methods, varargs can be overloaded, remember they are just a substitute for an array of a given type `type[]`, meaning that we can define and overload the function with multiple types of varargs such as the following

```
    void vamethod([other optional paramters], int ... v)
    void vamethod([other optional paramters], boolean ... v)
    void vamethod([other optional paramters], double ... v)
    void vamethod([other optional paramters], float ... v)
```

The second way to overload a varargs method is to add one or more normal parameters. This is what was done with vamethod(String, int …). In this case, Java uses both the number of arguments and their types. But that is no different in esense from the regular method overloading

```
    void vamethod(String n)
    void vamethod(String n, int ... v)
    // these two pairs are essentially equivalent, varargs is more or less
       syntactic sugar for an argument of type[] first and fore most
    void vamethod(String n)
    void vamethod(String n, int[] v)
    // this is why we can actually call varargs methods like that, even
       though it is pointless, it demonstrates how the compiler treats them
    vamethod("", new int[]{1, 2, 3});
```

Now what would happen if we had these two methods defined as illustrated below, a compile time error will arise precisely because the way the compiler implicitly treats the variable arguments paramters as arrays of a given type, we can not have the following

```
    void vamethod(String n, int[] v)
    void vamethod(String n, int ... v)
    // that is a compile time error, remember that varargs are essentially
       the same as type[]
```

There might be some ambiguity when we define and call varargs methods, which will cause compile time error, consider the following examples

```
    void vamethod(int ...v)
    void vamethod(boolean ...v)
```

```
    vamethod() // compile time error, since both are equally valid

    void vamethod(int ...v)
    void vamethod(int, int ...v)
    vamethod(1) // compile time error, since both are equally valid
```