Introduction to Programming – Functions

Sujit Chakrabarti

Loops enable us to perform repetitive computations without having to grow the code size in proportion to the number of times the task is to be done.

```
for(int j = 0; j < 10; j++) {
  System.out.println("Hello world!");
}
```

Figure 1: Loop to print "Hello world!" 10 times

The loop shown in fig. 1 will print out `"Hello world!"` ten times. This is good. However, this also has its drawbacks. Suppose that you wish to print another message, say "Hello friend!" twenty times, in addition to the above. Using just loops, there's only one way to do it: write another loop as shown in fig. 2.

```
for(int j = 0; j < 10; j++) {
  System.out.println("Hello world!");
}

for(int j = 0; j < 20; j++) {
  System.out.println("Hello friend!");
}
```

Figure 2: Loop to print "Hello world!" 10 times

We notice that the two loops above are very similar: they print *a message*, and they do it *certain number* of times. Barring the message and the number of times, the two actions are identical. We ask if Java gives us a way by which this identical-ness can be captured effectively, and can be used to our benefit. Turns out that *functions* (or, more appropriately *methods* in Java terminology) give us exactly what we are looking for.

1 Java Methods

Methods are reusable pieces of code which can be used as per needed anywhere and any number of times as required, of course, by not violating basic rules of Java language. When discussing methods, we talk of their creation (called *method definition*), and their use (called *method call* or *method invocation*).

1.1 Method Definition

Fig. 3 shows a method definition to do what loops were doing in fig. 1 and fig. 2.

```
public static void printMessage(String m, int n) {
   for(int j = 0; j < n; j++) {
     System.out.println("Hello " + m + "!");
   }
}
```

Figure 3: Method `printMessage` to print a message n times

1.2 Invoking a Method

The method definition shown in fig. 3 can be used as follows from the `main` method:

```
public static void main(String[] a) {
  printMessage("world", 10);
  printMessage("friend", 20);
}
```

Figure 4: Method `printMessage` called from `main` method twice.

This will give us an identical output as the code in fig. 2. However, as you will notice, we have now avoided the need to write the loop twice to do two sets of message printing. We captured its common part by defining a method as in fig. 3, and done both sets of printing by calling the method twice as shown in fig. 4.

1.3 Details

Let's look a bit more in details, the syntactic elements of a method. Components of a method definition:

- **Name.** A method has name by which it can be referred. In the example, the name of the method is `printMessage`.
- **Body.** Body captures whatever the method is supposed to do. In the example, the following piece of code is its body:

```
{
    for(int j = 0; j < n; j++) {
      System.out.println("Hello " + m + "!");
    }
}
```

- **Formal Parameters.** These are the input variables to the method. In the example, there are two parameters: `m` of type `String`, and `n` of type `int`.
- **Return type.** This signifies the type of the value the function computes. In the given example, the return type is `void`.
- **Access specifier.** The keyword `public` in the method definition is called the access specifier. For the moment, we aren't in a position to tell exactly what it means. So, let's just say that all our methods are going to be `public`. This is likely to change soon.
- **Storage class specifier.** The keyword `storage` in the method definition is called the storage class specifier. For the moment, we aren't in a position to tell exactly what it means. So, let's just say that all our methods are going to be `static`. This is going to change pretty soon.

Components of a method call:

1. Name. The name of the method.
2. Actual parameters. The values which are being sent to the method as inputs from the calling site are called the actual parameter. In the given example, `printMessage("world", 10)` has two actual parameters: `"world"` and `10`. `printMessage("friend", 20)`'s actual parameters are `"friend"` and `20`.

Here's another example of a method definition.

```
public static int add(int a, int b) {
  int sum = a + b;
  return sum;
}
```

Figure 5: Method `add` to add two numbers

Fig. 5 defines a method `add` to add to two integers `a` and `b` supplied to it as parameters. The body of the method is:

```
{
  int sum = a + b;
  return sum;
}
```

Return type is `int`. In this example, we see a new instruction `return sum;` which returns the value of `sum` to where the function was called from. A typical call to this function would look like:

```
public static void main(String[] a) {
  int c = add(10, 20);
  System.out.println(c);
}
```

Figure 6: Method invocation to `add` to add 10 and 20 and initialise the variable `c` with the value returned from it (i.e. 30).

The above function `add` does something trivial – i.e. adding two integers – for which it's not needed to write a function. However, in practice, function would often be implementation of non-trivial computations or algorithms, e.g. computing the factorial of n, the sum of all elements of an array, whether a given string is a palindrome or not, or even the shortest path between two nodes in a graph.

For the compilation to succeed, the following semantic rules must be satisfied:

1. The declared return type of a method and the type of value returned must agree. For example, let's change the method `add` in fig. 5 as follows:

```
  public static int add(int a, int b) {
    String sum = "Sum";
    return sum;
  }
```

The compiler will display the following error message:

```
error: incompatible types
    return sum;
           ^
  required: int
  found:    String
```

As can be seen, the declared return type of the method is `int`, but the type of the value being returned (`sum`) is `String`. This is not acceptable as per the language rules.

2. The number of formal parameters in the method definition should match the number of actual parameters in the method call. If we change the method call in fig. 6 to `int c = add(10);`, the compiler will give us the following error message:

```
error: method add in class Function cannot be applied
 to given types;
    int c = add(10);
            ^
  required: int,int
  found: int
  reason: actual and formal argument lists differ in
   length
```

3. The type of actual parameters in the function call should match the types of the corresponding formal parameters of the function definition. If we change the method call in fig. 6 to `int c = add(10, "20");`, the compiler will give us the following error message:

```
error: method add in class Function cannot be applied
 to given types;
    int c = add(10, "20");
            ^
  required: int,int
  found: int,String
  reason: actual argument String cannot be converted
```

```
   to int by method invocation conversion
```

It's a good idea to discuss the reasons behind the above typing rules being there in Java.