Lecture 05: Functions

SE115: Introduction to Programming I

Previously on SE 115…

- Loops
- for loop
- while loop
- do while loop
- Nested Loops

Functions

- We come back to the “flow” of the program once again.
- With conditionals, we were able to branch into different paths of execution.
- With loops, we were able to repeat a set of statements while some conditions held.
- Now, we will learn how to use functions, and learn about a new way to change the flow of the program!

Functions

- We come back to the “flow” of the program once again.
- With conditionals, we were able to branch into different paths of execution.
- With loops, we were able to repeat a set of statements while some conditions held.
- Now, we will learn how to use functions, and learn about a new way to change the flow of the program!
- Aren’t you excited? :)

Functions

- Functions vs methods
  - Functions are called “methods” in object oriented programming. They are the same thing, just the context is different.
- You have already used functions:
  - main(String[] args): the starting point of all our programs,
  - System.out.println(): prints something on the screen,
  - nextInt() method of Scanner class: gets the next integer from the keyboard
- You are already familiar with some of their properties:
  - They have a name,
  - They can have parameters (println),
  - They return values (nextInt).

Functions

- Remember how we have selected parts of the code for looping?
- Similar to that one, some parts of the code might be required to run not on a loop but on different occasions.
- For example, we might want to print something on the screen.
- This “procedure” requires a set of steps, and instead of repeating these steps, they are grouped into a block of code, and this block of code is given a name so that it can be called when needed.

Functions

- The main reason we have functions is that we can divide the problem into smaller pieces that are easier to maintain: “divide and conquer”.
- One other reason we use the functions is the concept of “reusability”.
- We don't need to reinvent the wheel again. Once having been defined, a function can be called and used and then re-called and re-used anytime required.

Functions

- Functions are one of the building blocks of more complex programs.
- Consider **println** and **nextInt**: We have used them to create programs for different purposes.
- It is also a means of abstraction.
    - For example, think about a light switch. You flip it, and it either turns on or off the lights. You don’t know anything about its electrical setup, such as the cables, switches and electrical power. All you do is to flip the switch.
    - Same goes for printf, we don’t know how it prints on the screen, but it does.

Functions

- Let’s learn about how we can “define” a function and how it operates.
- To do so, I’ll define the “main” function one more time:

```
public static void main(String[] args) {
  // function body
}
```

- The keywords “public” and “static” are required because of the object oriented nature of the Java programming language.
- They enable the function to be accessible by all (public), and does not require an object to be instantiated (static).

Functions

- The keyword “void” is the type of the function.
- We know about the primitives, such as int, double and float, as types.
- The “void” type means “nothing” - so this means:
  - the function does not return anything, or
  - the function does not produce anything for its caller.
- The name of the function is important; this is how it is called.

Functions

- The function “main” is special, it is the entry point to the program.
- The parameters are given in parentheses, in this case an array of String objects.
- If the type was not “void”, then we would have had a “return” line at the end of the function body.
    - We can still have a single
      return;
      at the end of the function body, but it is optional.

Functions

- Let’s write our first function besides main.
- The function **myFunction** has to be static, similar to main, so that we can call it without a `Functions` object.
- **myFunction** has a return type of integer; it also has two parameters, both integers.
- The function simply doubles the “value” of variable “a”, and adds the value of “b”, and returns the result.
- Let me show you the flow…

```
public class Functions {
    public static int myFunction(int a, int b) {
        return 2 * a + b;
    }

    public static void main(String[] args) {
        int x = 10;
        int y = 20;
        int z = myFunction(x, y);
        System.out.println(z);
        System.out.println(myFunction(x, y));
        System.out.println(myFunction(10, 20));
    }
}
```

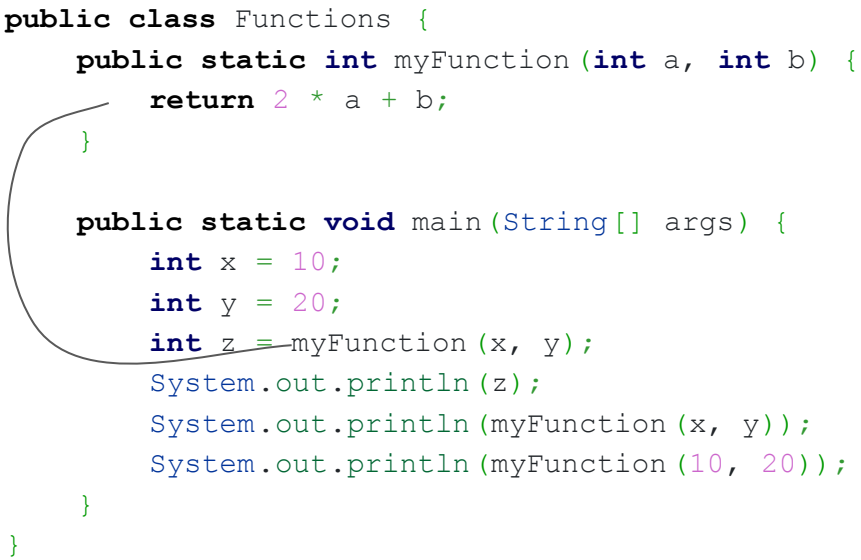
Functions

- The program starts at “main”.
- Then, on its third line, myFunction is called. This changes the flow of the program (see the arrow?).
- Now, some magic happens:
  - The values of arguments x and y, 10 and 20, are assigned to local variables of myFunction, a and b.
    - Now, a = 10 and b = 20.
    - This is called “**pass by value**,” we are passing the values, not the variables!

```
public class Functions {
    public static int myFunction(int a, int b) {
        return 2 * a + b;
    }

    public static void main(String[] args) {
        int x = 10;
        int y = 20;
        int z = myFunction(x, y);
        System.out.println(z);
        System.out.println(myFunction(x, y));
        System.out.println(myFunction(10, 20));
    }
}
```

Functions

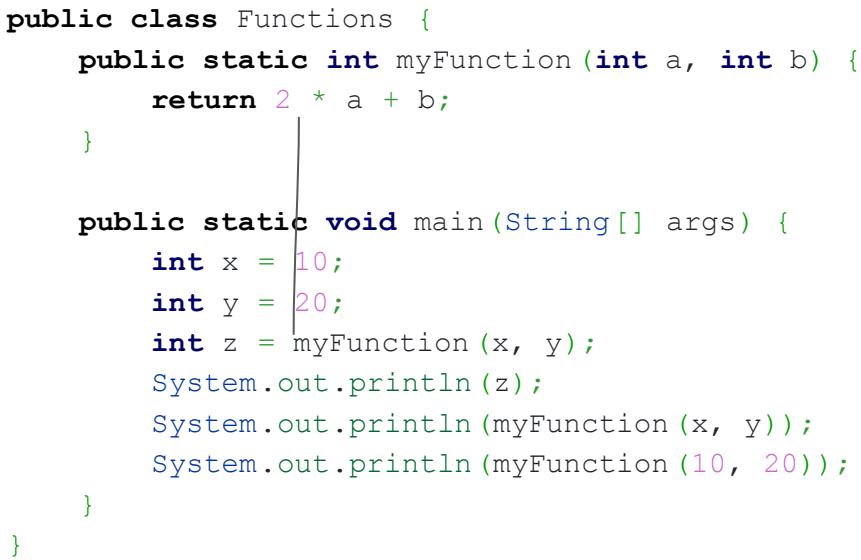
- Now we are in myFunction and a = 10, b = 20.
- The expression 2 * 10 + 20 in the “return” line evaluates to 40, and this value is “returned”.
- Where is it returned to?



```
public class Functions {
    public static int myFunction(int a, int b) {
        return 2 * a + b;
    }

    public static void main(String[] args) {
        int x = 10;
        int y = 20;
        int z = myFunction(x, y);
        System.out.println(z);
        System.out.println(myFunction(x, y));
        System.out.println(myFunction(10, 20));
    }
}
```

Functions

- Now we are in myFunction and a = 10, b = 20.
- The expression 2 * 10 + 20 in the “return” line evaluates to 40, and this value is “returned”.
- Where is it returned to?
  - There (see the arrow).
- Now the right hand side of the assignment operator has the **value** of 40, and this value is assigned to the variable z.



Functions

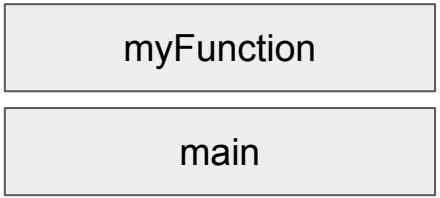
- The “returning” has two important aspects:
  - The **value** that has been returned (if not void),
  - and **where** it is returned.
- See, when you “call” a function, the program must know where to get back; so it uses a “stack”, which is like a stack of plates, and places the called function on top of the current function.

```
public class Functions {
    public static int myFunction(int a, int b) {
        return 2 * a + b;
    }

    public static void main(String[] args) {
        int x = 10;
        int y = 20;
        int z = myFunction(x, y);
        System.out.println(z);
        System.out.println(myFunction(x, y));
        System.out.println(myFunction(10, 20));
    }
}
```

Functions

- The “returning” has two important aspects:
  - The **value** that has been returned (if not void),
  - and **where** it is returned.
- See, when you “call” a function, the program must know where to get back; so it uses a “stack”, which is like a stack of plates, and places the called function on top of the current function.

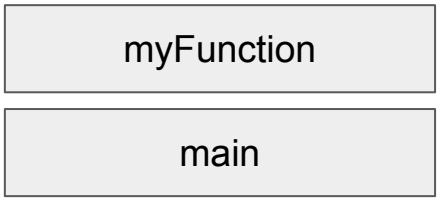
```
public class Functions {
    public static int myFunction(int a, int b) {
        return 2 * a + b;
    }

    public static void main(String[] args) {
        int x = 10;
        int y = 20;
        int z = myFunction(x, y);
        System.out.println(z);
        System.out.println(myFunction(x, y));
        System.out.println(myFunction(10, 20));
    }
}
```



Functions

- So, when myFunction is done, it is taken from that stack, and we know where we will get back; to the function just below it: main.
- Let's continue!
- The value of z is printed.
- Now, we get to the first aspect, the value.

```
public class Functions {
    public static int myFunction(int a, int b) {
        return 2 * a + b;
    }

    public static void main(String[] args) {
        int x = 10;
        int y = 20;
        int z = myFunction(x, y);
        System.out.println(z);
        System.out.println(myFunction(x, y));
        System.out.println(myFunction(10, 20));
    }
}
```



Functions

- Instead of assigning the returned value to a variable, we immediately use it as a “value” to another function: println.
- So, the line println(myFunction(x, y)) will print 40, too.
- What about the last one?
- This time we are explicitly sending values rather than sending the values of variables. Same result.

```
public class Functions {
    public static int myFunction(int a, int b) {
        return 2 * a + b;
    }

    public static void main(String[] args) {
        int x = 10;
        int y = 20;
        int z = myFunction(x, y);
        System.out.println(z);
        System.out.println(myFunction(x, y));
        System.out.println(myFunction(10, 20));
    }
}
```

Functions

- Values!
  - This is called “pass by value” (I know, I’m repeating this, but it is important).
  - This will become important when we are working with the “references” of objects - after the midterm.
  - For now, while we are using primitives, we will be dealing with values!
- Returning to previous function.
  - A function can call another function, as we always call println in main, and that called function can also call another function.
  - Call stack helps the computer to keep track of these calls.

```
public class Functions {
    public static int myFunction(int a, int b) {
        return 2 * a + b;
    }

    public static void main(String[] args) {
        int x = 10;
        int y = 20;
        int z = myFunction(x, y);
        System.out.println(z);
        System.out.println(myFunction(x, y));
        System.out.println(myFunction(10, 20));
    }
}
```

Function Locality and Scope

- Let’s remember what scope is.
- Scope is where the variables live; it is where they are bound.
- The curly braces define a “block scope”

```
public class Example {
  public static void main(String[] args) {
    int a = 12;
    for(int i=0;i<10;i++) {
      System.out.println(i);
    }
  }
}
```

Function Locality and Scope

- Function main lives in the class scope.
- Function parameter **args** lives in the function main scope.
- We also define a in the function main scope.

class scope

scope for function main

```
public class Example {
  public static void main(String[] args) {
    int a = 12;
    for(int i=0;i<10;i++) {
      System.out.println(i);
    }
  }
}
```

Function Locality and Scope

- We define a **for** loop using curly braces, and we define the variable i in that.
- As soon as the **for** block ends the variable i is out of its scope and does not exist anymore.

```
public class Example {                                  class scope
  public static void main(String[] args) {              scope for function main
    int a = 12;
    for(int i=0;i<10;i++) {                             scope for for loop
      System.out.println(i);
    }
  }
}
```

Function Locality and Scope

- The variable i is **local** to the **for** loop.
- The variables created in a block can go into other blocks, but if the variable is created in a block, it cannot get out of it.

```
public class Example {
  public static void main(String[] args) {
    int a = 12;
    for(int i=0;i<10;i++) {
      System.out.println(i);
    }
  }
}
```

class scope

scope for function main

scope for **for** loop

Function Locality and Scope

- We can access variable **a** inside the **for** loop.
- But we cannot access variable **i** outside the for loop.
- Same goes for functions and classes.

```
public class Example {                                  class scope
  public static void main(String[] args) {              scope for function main
    int a = 12;
    for(int i=0;i<10;i++) {                             scope for for loop
      System.out.println(i);
    }
  }
}
```

Function Locality and Scope

- The function keeps track of its local variables and any variable that comes from an outer scope.
- This can get very complicated, so be careful with the names of variables.
    - **Just because they have the same name it does not mean that they are the same variable. The scope matters.**

A Problem

- Let’s try to write a function that makes sense.
- Assume that we need an “absolute” function which returns the absolute value of an integer.
- The input will be an integer value.
- In the body, we will check if the value is smaller than zero.
- If so, we multiply it with -1.
- Then we return the positive value.

A Problem

- Here are a few “test cases”
  - Input: 2, must return 2
  - Input: 0, must return 0
  - Input: -1, must return 1
  - Input: -2, must return 2
- The borders, in this case values around zero are important.
- We must make sure that these cases are handled correctly.

A Problem

- Let's call the function **absolute**, it will accept an **int**eger value, and will **return** an **int**eger value. Here is the signature:

```
public static int absolute(int v) { }
```

- The body should check if the value of variable **v** is positive or not.

```
if (v < 0) { v *= -1; }
```

- We should return the value:

```
return v;
```

A Problem

```
public class AbsoluteF {
    public static int absolute(int v) {
        if (v < 0) {
            v *= -1; // return v * (-1);
        }
        return v;
    }

    public static void main(String[] args) {
        System.out.println("Absolute -1: " + absolute(-1));
        System.out.println("Absolute 0: " + absolute(0));
        System.out.println("Absolute 2: " + absolute(2));
        for(int i=-10;i<2;i++) {
            System.out.println("Absolute " + i + ": " + absolute(i));
        }
    }
}
```

we can immediately return here, too.

Use a loop to test a lot of values!

Another Problem

- How about the **isPrime** function?

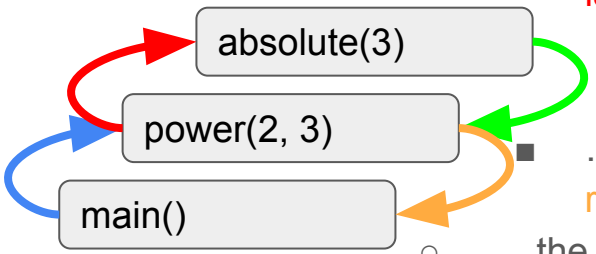
Another Problem

- How about the **isPrime** function?
  - We input a value.
  - We check if it is prime.
    - How can we check it?
  - We return true if it is prime, false otherwise.
- You already did this in a lab, now try to refactor it as a function at home!

Mathematical Functions

- Let’s create a class that includes some mathematical functions.
- Java already includes one, Math, and that is the one you should be using, but let’s try this for practice and for demonstration.
- Let’s call the class MyMath;
  - We already have an **absolute** function,
  - Let’s add a **power** function,
  - Also a **min** function which returns the smaller of two values,
  - And a **max** function which returns the greater of two values.

```java
public class MyMath {
    public static int absolute(int v) {
        if (v < 0) {
            v *= -1;
        }
        return v;
    }

    public static double power(int base, int pwr) {
        if(pwr == 0) return 1;
        boolean reverse = false;
        if(pwr < 0) reverse = true;
        int b = base;
        for(int i=1;i<absolute(pwr); i++) {
            b *= base;
        }
        if (reverse) return 1.0/(double)b;
        return b;
    }

    public static int min(int a, int b) {
        if(a < b) return a;
        else return b;
    }

    public static int max(int a, int b) {
        if(a > b) return a;
        else return b;
    }

    public static void main(String[] args) {
        System.out.println("Testing functions!");
        System.out.println("2^3=" + power(2, 3));
        System.out.println("13^0=" + power(13, 0));
        System.out.println("2^(-3)=" + power(2, -3));
        System.out.println("Min(2, 3)=" + min(2, 3));
        System.out.println("2^16=" + power(2, 16));
    }
}
```



- What happens when the **power** function is called?
  - The program is in the **main** function; and before we can call the **println** we have to calculate the value of **power(2, 3)**. So we call it.
    - When in **power** we have to take the **absolute** of the pwr variable. So **absolute** is called.
      - In **absolute**, we calculate the absolute value and return to…
    - … the **power** method. When it is done, we return to…
  - … the **main** method.
- This is the call stack! Each function returns to the function it was called from…

Function calls and scopes

- Once again: the function returns to the function it has been called from.
- Each function has its own local scope.
- Next week we will continue with more interesting functions!
- Make sure you try all the exercises.