# Lecture Notes

# Functions

Welcome to the session on Functions.

In the last session, you learnt about loops. Loops enable us to perform repetitive computations without having to grow the code length in proportion to the number of times the task repeated.
Consider this code:

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

The loop shown will print out " Hello world !" ten times, which 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 follows —

```
for(int j = 0; j < 10; j++) {
    System .out . println (" Hello world !");
}
for(int j = 0; j < 20; j++) {
    System .out . println (" Hello friend !");
}
```

Notice that the two loops above are very similar: they print a message, and they do it a certain number of times. Apart from the message and the number of times the loops are executed, the two loops are identical. But, you can ask if Java gives us a way to effectively capture this identicalness, which we can use to our benefit. Turns out that functions (or more precisely, methods, in Java terminology) give us exactly what we are looking for.

**Functions** are reusable pieces of code, which we can use as per our needs anywhere and any number of times, of course, by not violating the basic rules of the Java language.

When discussing methods, we talk about their creation (called method definition), and their use (called method call or method invocation).

You learnt two major types of functions, ones that returned a value and the others that didn't. So, the first method you looked at was printHelloWorld. Here is the code for this function:

```
public static void printHelloWorld() {
    System.out.println("Hello World!");
}
```

This function, when called inside the main function, printed "Hello world!". This function did not return any value, which could be stored in a variable or further acted upon. This was a standalone function. When called, it would execute the statement inside it but nothing more.

Then you looked at another function that returned a variable:

```
public static int test() {
    int variable = 9;
    return variable;

}
```

## Defining Functions

Let's look at function definitions a bit more in detail. Here are the important components of a function definition:

- **Name**: A method has a name, which is used to refer to it. In the example, the name of the methods were printHelloWorld() and test(). **A method can only be called by the proper name that is assigned to it**.
- **Body**: The body of a method captures whatever the method is supposed to do. It refers to the statements inside the "{ } " brackets.
- **Return type**: This signifies the type of the value the function computes. In the printHelloWorld example, it didn't return any data, so the return type was void. In the other example, the return type was integer, and it was used to return an integer value.
- **Access specifier**: The public keyword in the method definition is the access specifier. You will learn more about this in the next lectures. For now, all our methods in this session are public.
- **Storage class specifier**: The keyword storage in the method definition is called the storage class specifier. You will learn more about this in the next lectures. In this lecture, all the methods are static.

But, these functions you just went through were independent from the rest of the code inside the main body. Moreover, such functions have very limited usage. You need functions to interact with the code. The function should be able to take values from some variables and perform some operations on them. Then, it will return something or display some result based on these values. Ideally, this is how it will help you even more.

This could be done by using parameters, i.e., by passing input variables into the method.

```
public class HelloFriend{
    public static void main(String[] args) {
        printGreeting("Danny");
    }

    public static void printGreeting(String name) {
        System.out.println("Hello " + name);
    }
}
```

Here, you passed the value “Danny” as a parameter into the function. The function definition indicated that it expects a parameter of type String to be passed into it. Here, the function simply took the name and printed a greeting using this name.

After this, you went through another example where you used a function to compare two values passed into it.

```
public class CompareFunction {

    public static void main(String[] args) {
        int num1 = 20;
        int num2 = 27;
        String compareResult = compare(num1, num2);
        System.out.print(compareResult);

    }

    public static String compare(int var1, int var2) {

        String result;
        if (var1 > var2) {
            result = var1 + " is larger than " + var2;
        } else if (var1 == var2) {
            result = "Both numbers are equal";
        } else {
            result = var2 + " is larger than " + var1;
        }
        return result;
    }
}
```

Here are some important considerations for the compilation to succeed:

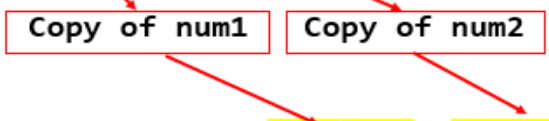
1. The declared return type of a method and the type of the value returned must be the same. However, if a method doesn’t return any value, it must be declared with a void return type
2. The number of formal parameters in the method definition should match the number of actual parameters in the method call.
3. The types of actual parameters in the method call should match the types of the corresponding formal parameters of the method definition.

Then you looked at how a swap function would not change the values passed into the function because Java uses pass-by-value.

```
public class SwapNum {
    public static void main(String[] args) {
        int num1 = 10;
        int num2 = 16;

        swap(num1, num2);
        System.out.println("num1 = "+num1+" num2 = "+num2);
    }

    public static void swap(int var1, int var2)
    {
        int temp = var2;
        var2 = var1;
        var1 = temp;
    }
}
```




## Inbuilt Functions in Java

After this, you learnt about some inbuilt functions in Java, which are methods written for you by the creators of Java. You used some of the functions from MATH and STRING libraries. These libraries are crucial in Java they can considerably reduce your efforts when programming. Some of the functions you saw were Math.sqrt() and Math.abs() from the MATH library, and String.length(), String.concat(), and String.equalsIgnoreCase() from the String library.

Then, you saw how to read the content of your files using code. Here, we used the text “Alice in Wonderland” to perform various operations. The operations included reading a single line, reading all lines, and reading all words from a file.

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;

public class ScanText {
    public static void main(String args[]) throws FileNotFoundException {

        //creating File instance to reference text file in Java
        File inputfile = new File("C:\\Users\\path");
        Scanner scan = new Scanner(inputfile);
```

```
        //Reading each line of file using Scanner class
        int lineNumber = 1;  // keep track of line number

        while (scan.hasNextLine() == true) {
            String line = scan.nextLine();
            System.out.println("line " + lineNumber + " :" + line);
            lineNumber++;
        }
    }
}
```

You also learnt that a sentence different from a line. The 'Enter', or the hidden new line expression '**\n**', after each line, is recognised by the scanner in the code, and used to separate one line from another.

When instead of lines, you had to read all the words from the file, you used the hasNext() function instead of hasNextLine().

# Exceptions

Let's recall what happens when you run a program:

1. The program is compiled.
2. The program is run/executed.

However, when the program is running, it may run into problems or errors and stop its execution.

So, let's look at an example where your compiler returns an exception error when running the code:

```
public class Exceptions {
    public static void main(String args[]) {
        int data = 67/0;
        System.out.println("rest of the code goes here");
    }
}
```

The exception was 'main' java.lang.ArithmeticException: / by zero. It states that there is a mathematical or arithmetic error, and it is due to an attempted division by zero. This was because, division by 0 is not defined.

The rest of the code, after this exception, is not executed. The code executes up to this point and stops executing after this exception is encountered. Here, you understood what the source of the error could be. However, the code following the source of the exception will not always be dependent on the source, i.e. suppose that this division statement was independent of the rest of the code and was only a small part of

it. In that case, it's not fair to stop the execution of the rest of the code just because there is an error in a small independent part.

To wrap your head around this, you need to understand the concept of 'exceptions'.

An exception, by definition, occurs when the flow of a program is disrupted and ends abnormally or unexpectedly. You don't want the program to end abnormally; instead, you need it to tell you what error occurred, if any. For example, exceptions may occur when the user enters invalid data, or when a file cannot be found. Exceptions may also be caused by a user input or a flaw in the program logic.

You handled the exception in the program above by using a 'try and catch' block, as shown below:

```
public class Exceptions {
    public static void main(String args[]) {

        try {
            int data = 50 / 0;
        } catch (Exception e) {
            System.out.println(e);
        }

        System.out.println("Rest of the code is here");
    }
}
```

So, to catch the error and continue the code execution after the error, we'll use the following try and catch block:

```
try {
  // code that can throw exceptions
} catch (Exception e) {
  System.out.println(e);
}
```

A 'try' block is always followed by a 'catch' block. A try block contains the code where exceptions may arise. If the exception occurs, it is caught by the catch block. However, the program doesn't stop altogether. Instead, the rest of the code in the catch block is executed. In other words, the catch block protects your program from crashing completely and can help your program recover or terminate gracefully from the unexpected exception.

There are two common types of exceptions:

1. **Checked exceptions:** These exceptions have to be taken care of while writing the program, since they are checked at the time of compilation. If a statement in your code throws a checked exception, then there must be a statement to either handle the exception, or you must specify the exception using the 'throws' keyword. Otherwise, the program will give you a compilation error.
2. **Unchecked exceptions**: These exceptions are not checked during the program compilation. Hence, there will be no compilation error even if they are not checked. But will cause your program to crash and terminate unexpectedly.

You can handle checked exceptions using either the try and catch block or the Throws keyword, which helps you specify the exception. When reading content from a file, you would recall that we wrote the following highlighted statements:

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;

public class ReadFile {
    public static void main(String args[]) throws FileNotFoundException {

        File inputfile = new File("C:\\Users\\XYZ ");
        Scanner scan = new Scanner(inputfile);
        // scan lines from the file
        String Line =scan.nextLine();
        System.out.println(Line);
    }
}
```

This is because when you use the scan(file) statement, it throws an exception, *FileNotFoundException*. This exception is checked during the program's compilation. Hence, if there is no provision in the code to handle it, or it isn't specified, then there is a compilation error.

You can also write the above code using try and catch:

```
import java.util.Scanner;
import java.io.File;

public class TryNCatch {
    public static void main(String args[]) {

        try {
            //creating File instance to reference text file in Java
            File inputfile = new File("C:\\Users\\path");
            Scanner scan = new Scanner(inputfile);
            while (scan.hasNextLine() == true) {
                String line = scan.nextLine();
                System.out.println(line);
            }

            scan.close();
        } catch (Exception e) {
            System.out.print("File not found");
        }
    }
}
```

Now, you've written the part of the program that can throw an exception into the try block, so the program will try and execute it. But if an exception is thrown, the exception will be intercepted or caught by the catch block, and the instruction in the catch block will then be executed.

A 'try and catch' block is especially useful in cases where you need to read from a file. This checks if the exception can be handled at the time of compilation. At other times, it is good to have this in place, especially if the code is long and contains several independent parts. It helps you to identify which part threw the exception.

Then, you used the try and catch block to create a program that writes its output to a file:

```
import java.util.Scanner;
import java.io.FileWriter;

public class WriteFile {
    public static void main(String args[]) {
        Scanner scan = new Scanner(System.in);
        try {
            FileWriter writer = new FileWriter("C:\\Users\\..\\test.txt");
            Boolean write = true;

            System.out.print("Enter what you wish to write ");
            String line = scan.nextLine();
            // write this to the file.
            writer.write(line);
            writer.close();
        } catch (Exception e) {
            System.out.println(e);
        }
        System.out.println("Done");
    }
}
```

Thus, handling exceptions is a necessary requirement in programming and a good habit to establish

When you are building a large program, there may be many independent parts in the program that can work independently from one another. Therefore, you don't want an exception in one part of the program to cause the entire program to crash completely, especially when other independent parts of the program can continue to execute without issues.

Therefore, it is a good practice to wrap your code around try catch blocks, especially around pieces of code that you suspect may cause exceptions. This will help you quarantine code that causes unexpected exceptions, and prevent it from affecting other parts of the program.

Specifically, by wrapping code and exceptions within try catch blocks, you will be able to:

- allow other parts of the program to continue operation without interruption or
- allow other parts of the program to finish whatever they are working on and let the program reach a state to terminate gracefully