Day 6
Thu 24 Aug 2017
Java Tutorial
Instructor: Hameed Mahmoud

Day 6

Basics of Java Programming Language

Objectives

1. Practice
2. Syntax
3. Try Catch
4. File I/O

Java Syntax

When we consider a Java program, it can be defined as a collection of objects that communicate via invoking each other's methods. Let us now briefly look into what do class, object, methods, and instance variables mean.

- **Object** – Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behavior such as wagging their tail, barking, eating. An object is an instance of a class.
- **Class** – A class can be defined as a template/blueprint that describes the behavior/state that the object of its type supports.
- **Methods** – A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.
- **Instance Variables** – Each object has its unique set of instance variables. An object's state is created by the values assigned to these instance variables.

First Java Program

Let us look at a simple code that will print the words ***Hello World***.

Example

```
public class MyFirstJavaProgram {

   /* This is my first java program.
    * This will print 'Hello World' as the output
    */

   public static void main(String []args) {
      System.out.println("Hello World"); // prints Hello World
   }
}
```

Let's look at how to save the file, compile, and run the program. Please follow the subsequent steps –

- Open notepad and add the code as above.
- Save the file as: MyFirstJavaProgram.java.
- Open a command prompt window and go to the directory where you saved the class. Assume it's C:\.
- Type 'javac MyFirstJavaProgram.java' and press enter to compile your code. If there are no errors in your code, the command prompt will take you to the next line (Assumption : The path variable is set).
- Now, type ' java MyFirstJavaProgram ' to run your program.
- You will be able to see ' Hello World ' printed on the window.

Output

```
C:\> javac MyFirstJavaProgram.java

C:\> java MyFirstJavaProgram

Hello World
```

Basic Syntax

About Java programs, it is very important to keep in mind the following points.

- **Case Sensitivity** – Java is case sensitive, which means identifier **Hello** and **hello** would have different meaning in Java.
- **Class Names** – For all class names the first letter should be in Upper Case. If several words are used to form a name of the class, each inner word's first letter should be in Upper Case.
- **Example:** *class MyFirstJavaClass*
- **Method Names** – All method names should start with a Lower Case letter. If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.
- **Example:** *public void myMethodName()*
- **Program File Name** – Name of the program file should exactly match the class name.
- When saving the file, you should save it using the class name (Remember Java is case sensitive) and append '.java' to the end of the name (if the file name and the class name do not match, your program will not compile).
- **Example:** Assume 'MyFirstJavaProgram' is the class name. Then the file should be saved as *'MyFirstJavaProgram.java'*
- **public static void main(String args[])** – Java program processing starts from the main() method which is a mandatory part of every Java program.

Java Identifiers

All Java components require names. Names used for classes, variables, and methods are called **identifiers**.

In Java, there are several points to remember about identifiers. They are as follows –

- All identifiers should begin with a letter (A to Z or a to z), currency character ($) or an underscore (_).
- After the first character, identifiers can have any combination of characters.
- A key word cannot be used as an identifier.
- Most importantly, identifiers are case sensitive.
- Examples of legal identifiers: age, $salary, _value, __1_value.
- Examples of illegal identifiers: 123abc, -salary.

Java Modifiers

Like other languages, it is possible to modify classes, methods, etc., by using modifiers. There are two categories of modifiers –

- **Access Modifiers** – default, public , protected, private
- **Non-access Modifiers** – final, abstract, strictfp

We will be looking into more details about modifiers in the next section.

Java Variables

Following are the types of variables in Java –

- Local Variables
- Class Variables (Static Variables)
- Instance Variables (Non-static Variables)

Java Arrays

Arrays are objects that store multiple variables of the same type. However, an array itself is an object on the heap. We will look into how to declare, construct, and initialize in the upcoming chapters.

Java Enums

Enums were introduced in Java 5.0. Enums restrict a variable to have one of only a few predefined values. The values in this enumerated list are called enums.

With the use of enums it is possible to reduce the number of bugs in your code.

For example, if we consider an application for a fresh juice shop, it would be possible to restrict the glass size to small, medium, and large. This would make sure that it would not allow anyone to order any size other than small, medium, or large.

Example

```
class FreshJuice {
   enum FreshJuiceSize{ SMALL, MEDIUM, LARGE }
   FreshJuiceSize size;
}

public class FreshJuiceTest {

   public static void main(String args[]) {
      FreshJuice juice = new FreshJuice();
      juice.size = FreshJuice.FreshJuiceSize.MEDIUM ;
      System.out.println("Size: " + juice.size);
   }
}
```

The above example will produce the following result –

Output

```
Size: MEDIUM
```

Note – Enums can be declared as their own or inside a class. Methods, variables, constructors can be defined inside enums as well.

Java Keywords

The following list shows the reserved words in Java. These reserved words may not be used as constant or variable or any other identifier names.

abstract	assert	boolean	break
byte	case	catch	char
class	const	continue	default
do	double	else	enum
extends	final	finally	float
for	goto	if	implements
import	instanceof	int	interface
long	native	new	package
private	protected	public	return
short	static	strictfp	super

switch	synchronized	this	throw
throws	transient	try	void
volatile	while		

Comments in Java

Java supports single-line and multi-line comments very similar to C and C++. All characters available inside any comment are ignored by Java compiler.

Example

```
public class MyFirstJavaProgram {

   /* This is my first java program.
    * This will print 'Hello World' as the output
    * This is an example of multi-line comments.
    */

   public static void main(String []args) {
      // This is an example of single line comment
      /* This is also an example of single line comment. */
      System.out.println("Hello World");
   }
}
```

Output

```
Hello World
```

Using Blank Lines

A line containing only white space, possibly with a comment, is known as a blank line, and Java totally ignores it.

Inheritance

In Java, classes can be derived from classes. Basically, if you need to create a new class and here is already a class that has some of the code you require, then it is possible to derive your new class from the already existing code.

This concept allows you to reuse the fields and methods of the existing class without having to rewrite the code in a new class. In this scenario, the existing class is called the **superclass** and the derived class is called the **subclass**.

Interfaces

In Java language, an interface can be defined as a contract between objects on how to communicate with each other. Interfaces play a vital role when it comes to the concept of inheritance.

An interface defines the methods, a deriving class (subclass) should use. But the implementation of the methods is totally up to the subclass.

Catching Exceptions

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following –

Syntax

```
try {
   // Protected code
}catch(ExceptionName e1) {
```

```
   // Catch block
}
```

The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception occurred is handled by catch block associated with it. Every try block should be immediately followed either by a catch block or finally block.

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

Example

The following is an array declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

```
// File Name : ExcepTest.java
import java.io.*;

public class ExcepTest {

   public static void main(String args[]) {
      try {
         int a[] = new int[2];
         System.out.println("Access element three :" + a[3]);
      }catch(ArrayIndexOutOfBoundsException e) {
         System.out.println("Exception thrown  :" + e);
      }
      System.out.println("Out of the block");
   }
}
```

This will produce the following result –

Output

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
Out of the block
```

Multiple Catch Blocks

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following –

Syntax

```
try {
   // Protected code
}catch(ExceptionType1 e1) {
   // Catch block
}catch(ExceptionType2 e2) {
   // Catch block
}catch(ExceptionType3 e3) {
   // Catch block
}
```

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

Example

Here is code segment showing how to use multiple try/catch statements.

```
try {
   file = new FileInputStream(fileName);
   x = (byte) file.read();
}catch(IOException i) {
   i.printStackTrace();
   return -1;
}catch(FileNotFoundException f) // Not valid! {
   f.printStackTrace();
   return -1;
}
```

Catching Multiple Type of Exceptions

Since Java 7, you can handle more than one exception using a single catch block, this feature simplifies the code. Here is how you would do it –

```
catch (IOException|FileNotFoundException ex) {
   logger.log(ex);
   throw ex;
```

The Throws/Throw Keywords

If a method does not handle a checked exception, the method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword.

Try to understand the difference between throws and throw keywords, *throws*is used to postpone the handling of a checked exception and *throw* is used to invoke an exception explicitly.

The following method declares that it throws a RemoteException –

Example

```
import java.io.*;
public class className {

   public void deposit(double amount) throws RemoteException {
      // Method implementation
      throw new RemoteException();
   }
   // Remainder of class definition
}
```

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an InsufficientFundsException –

Example

```
import java.io.*;
public class className {

   public void withdraw(double amount) throws RemoteException,
      InsufficientFundsException {
      // Method implementation
   }
   // Remainder of class definition
}
```

The Finally Block

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax –

Syntax

```
try {
   // Protected code
}catch(ExceptionType1 e1) {
   // Catch block
}catch(ExceptionType2 e2) {
   // Catch block
}catch(ExceptionType3 e3) {
   // Catch block
}finally {
   // The finally block always executes.
}
```

Example

```
public class ExcepTest {

   public static void main(String args[]) {
      int a[] = new int[2];
      try {
         System.out.println("Access element three :" + a[3]);
      }catch(ArrayIndexOutOfBoundsException e) {
         System.out.println("Exception thrown  :" + e);
      }finally {
         a[0] = 6;
         System.out.println("First element value: " + a[0]);
         System.out.println("The finally statement is executed");
      }
   }
}
```

This will produce the following result –

Output

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
First element value: 6
The finally statement is executed
```

Java FileOutputStream Class

Java FileOutputStream is an output stream used for writing data to a file.

If you have to write primitive values into a file, use FileOutputStream class. You can write byte-oriented as well as character-oriented data through FileOutputStream class. But, for character-oriented data, it is preferred to use FileWriter than FileOutStream.

Java FileOutputStream Example 1: write byte

```
import java.io.FileOutputStream;

public class FileOutputStreamExample {

    public static void main(String args[]){

          try{

            FileOutputStream fout=new FileOutputStream("D:\\testout.txt");

            fout.write(65);

            fout.close();

            System.out.println("success...");

           }catch(Exception e){System.out.println(e);}

     }

}
```

1.

Output:

```
Success...
```

The content of a text file **testout.txt** is set with the data **A**.

testout.txt

A

Java FileInputStream Class

Java FileInputStream class obtains input bytes from a file. It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc. You can also read character-stream data. But, for reading streams of characters, it is recommended to use FileReader class.

Java FileInputStream example 1: read single character

```
import java.io.FileInputStream;

public class DataStreamExample {

    public static void main(String args[]){

        try{

            FileInputStream fin=new FileInputStream("D:\\testout.txt");

            int i=fin.read();

            System.out.print((char)i);



            fin.close();

        }catch(Exception e){System.out.println(e);}

    }
```

```
    }
```

Note: Before running the code, a text file named as **"testout.txt"** is required to be created. In this file, we are having following content:

Welcome to javatpoint.

After executing the above program, you will get a single character from the file which is 87 (in byte form). To see the text, you need to convert it into character.

Output:

W

Java FileInputStream example 2: read all characters

```
package com.javatpoint;


import java.io.FileInputStream;
public class DataStreamExample {
   public static void main(String args[]){
       try{
         FileInputStream fin=new FileInputStream("D:\\testout.txt");
         int i=0;
         while((i=fin.read())!=-1){
          System.out.print((char)i);
         }
```

```
        fin.close();

      }catch(Exception e){System.out.println(e);}

    }

  }
```

Output:

```
Welcome to javaTpoint
```