

ELINDA LARA

JAVA PROGRAMMING FOR ABSOLUTE BEGINNERS



A QUICK COURSE FOR MASTERING THE BASICS
OF JAVA

ELINDA LARA

ELINDA LARA

JAVA PROGRAMMING FOR ABSOLUTE BEGINNERS



A QUICK COURSE FOR MASTERING THE BASICS
OF JAVA

ELINDA LARA

Java Programming for absolute beginners

Table of Contents

[Java Programming for absolute beginners](#)

[Table of Contents](#)

[Introduction](#)

[Chapter 1: getting Started With java](#)

[Chapter 2: Writing Your First java Program](#)

[Chapter 3: data types](#)

[Chapter 4: Variables](#)

[Chapter 5: operators](#)

[Chapter 6: objects and classes](#)

[Chapter 7: decision making](#)

[Chapter 8: Loop control](#)

[Chapter 9: File handling](#)

[Chapter 10: Exception handling](#)

[Conclusion](#)

Introduction

java is a cross-platform, high-level language that was developed by Sun microsystems under the leadership of james gosling. The first version of this

Language was released in 1995 in the form of Java 1.0 [J2SE]. Since then, Java has come a long way and we are presently working on Java 8. Besides this, several versions of Java like J2ME (Java for mobile applications) and J2EE (Java for enterprise applications) have also been released. Java was released as open source software under GNU GPL by Sun in 2006. The process of this transformation was completed in 2007.

This book is a beginner's course on Java fundamentals. Therefore, it has been created keeping in mind that the reader has little to no background knowledge about Java. However, a little background of programming languages shall be helpful for better understanding. Through this course, we hope to instill the basics of programming in the reader from the perspective and with special focus on Java.

Before you begin, it is good for you to understand that programming concepts are generic and apply well to a range of programming languages. Therefore, if you work towards building programming concepts instead of learning syntax, you will be able to learn many programming languages with minimal effort. Focus on the concept instead of the syntax!

Copyright 2017 - All rights reserved.

This document is geared towards providing exact and reliable information in regards to the topic and issue covered. The publication is sold with the idea that the publisher is not required to render accounting, officially permitted, or otherwise, qualified services. If advice is necessary, legal or professional, a practiced individual in the profession should be ordered.

- From a declaration of Principles which was accepted and approved equally by a committee of the American Bar Association and a committee of Publishers and Associations.

In no way is it legal to reproduce, duplicate, or transmit any part of this document in either electronic means or in printed format. Recording of this publication is strictly prohibited and any storage of this document is not allowed unless with written permission from the publisher. All rights reserved.

The information provided herein is stated to be truthful and consistent, in that any liability, in terms of inattention or otherwise, by any usage or abuse of any policies, processes, or directions contained within is the solitary and utter

Responsibility of the recipient reader. Under no circumstances will any legal responsibility or blame be held against the publisher for any reparation, damages, or monetary loss due to the information herein, either directly or indirectly.

Respective authors own all copyrights not held by the publisher.

The information herein is offered for informational purposes solely, and is universal as so. The presentation of the information is without contract or any type of guarantee assurance.

The trademarks that are used are without any consent, and the publication of the trademark is without permission or backing by the trademark owner. All trademarks and brands within this book are for clarifying purposes only and are the owned by the owners themselves, not affiliated with this document.

Chapter 1: getting Started With java

Before we move on to java programming, let us give you a brief background of java, the programming language, and the prerequisites of learning this programming language. One of the most important features of this programming language is that it is cross-platform. This is evident from the fact that java was first advertised with the tagline 'Write once, Run Anywhere'. So, all you need to do is write a code, compile it and then you can run it on any system, be it Windows, Unix, Linux or macintosh.

Besides the above-mentioned, other features of java include –

- Platform independent
- Object oriented
- Simple
- Secure
- Portable
- Architecture-independent
- Interpreted
- Robust
- distributed
- multithreaded
- dynamic
-

high Performance

-

Requirements to get Started

IN order to get started with java programming, you need to first have a working computer with LINUX or WINDOWS or macintosh. ON this system, you are going to use two tools for java programming. The first of these tools is a text editor. If you are working ON WINDOWS, you can use Notepad.

For other operating systems, you can try textedit or VI or whichever text editor is locally available. The second requirement for java programming is java 8. You will need to install java 8 ON your system. If you do not wish to

Work ON a local ENVIRONMENT setup, you can also try one of the online options available to compile and run your java programs.

getting a Local Setup Ready

Assuming that you have the system ready and you already have a text editor installed ON the system, we are directly jumping to the installation of java ON the system. To start with, you will need to download java. java SE is the free version that you can download using the link that google Search for the same provides. Be sure to download the version that is developed for your operating system.

Once the download is complete, you will have an executable, which you are required to run for installation. Upon completion of installation, you will get a message stating the same. however, you are NOT done yet. java will NOT work until you set ENVIRONMENT variables. You can set the ENVIRONMENT variables ON your system using the following instructions –

Windows

IN all probability, the java version that you installed BY clicking ON the .exe file that came along with the installation package must have installed IN the directory: c:\Program Files\Java\jdk. Assuming this, you just need to open properties BY right clicking ON ‘my computer’. IN the dialog BOX that appears, open ‘Advanced’ and go to ‘ENVIRONMENT variables’. Append the existing contents of the field BY ‘, c:\Program Files\Java\jdk\BIN’.

LINUX/freeBSD/UNIX/Solaris

As a rule, the path of the ENVIRONMENT variable is the path at which the BINARIES of the java installation are placed. You can look for these BINARIES ON your system to get the actual path and then edit the .bashrc file. After you open this file using any text editor, you need to add the statement:

Export path = /path/to/java:\$path

While the standard way to run java programs is to create .java files IN notepad or any text editor for that matter and then execute the file using COMMAND-line COMMANDS. however, over all these years, several IDEs have also come INTO EXISTENCE to make this task hassle-free for the programmer.

netbeans and Eclipse are two such popular IDEs. You can read about them online to see if they interest you.

Chapter 2: Writing Your First Java Program

Java implements object-oriented programming principles. Therefore, a Java program is a class file that upon execution creates a set of objects. These objects communicate with each other by means of methods. Some of the terms that you will commonly encounter in the Java terminology have been explained below –

The foundation element of a Java program is **class**. It is simply a template that defines the state and behavior of an object.

-

The state of the object is defined by **Variables** and the behavior is described by **methods**. Therefore, a class includes variables and methods. It is important to mention here that each object created using a class has its own set of variables and thus, variables are also called **Instance Variables**.

-

When we instantiate a class, we create an **object**, which is the second fundamental element of Java programming.

-

In order to help you understand this concept, let us take an example. Suppose we create a class called car. Car is a general term for all four-wheeled small vehicles. However, every car has a different set of specifications like brand, color, engine, transmission type and features. These features can be modified. For example, you may get the color changed or a feature enhanced.

Therefore, car is a template (or in our case a class) but specific cars are objects or instances of this class. The set of specifications are variables and any initializations or manipulations that we perform on these variables are done using methods.

This example can also be best used to understand the object oriented principles of encapsulation and abstraction. The class encapsulates variables and methods into it. So, as the user, you are dealing with only the class car and its implementation is hidden from you. This is how Java implements abstraction.

The best way to learn programming is by writing programs. So, we have given you a sample program below for you to try. We will also use this program to explain the basic syntax of Java programming language.

```
Public class samplejavaprogram {
```



```
/* this program will print hello World! ON the screen */
```

```
Public static void main(String []args) {
```

```
    System.out.println("hello World!"); // statement for print
```

```
}
```

```
}
```

In order to write, compile and run this program, you need to follow the instructions given below –

Using any text editor, create a text file and copy this code into the same. Save the file as samplejavaprogram.java

- •

Compile and run the code using the ide. For manual execution, you need to type the following commands:

-

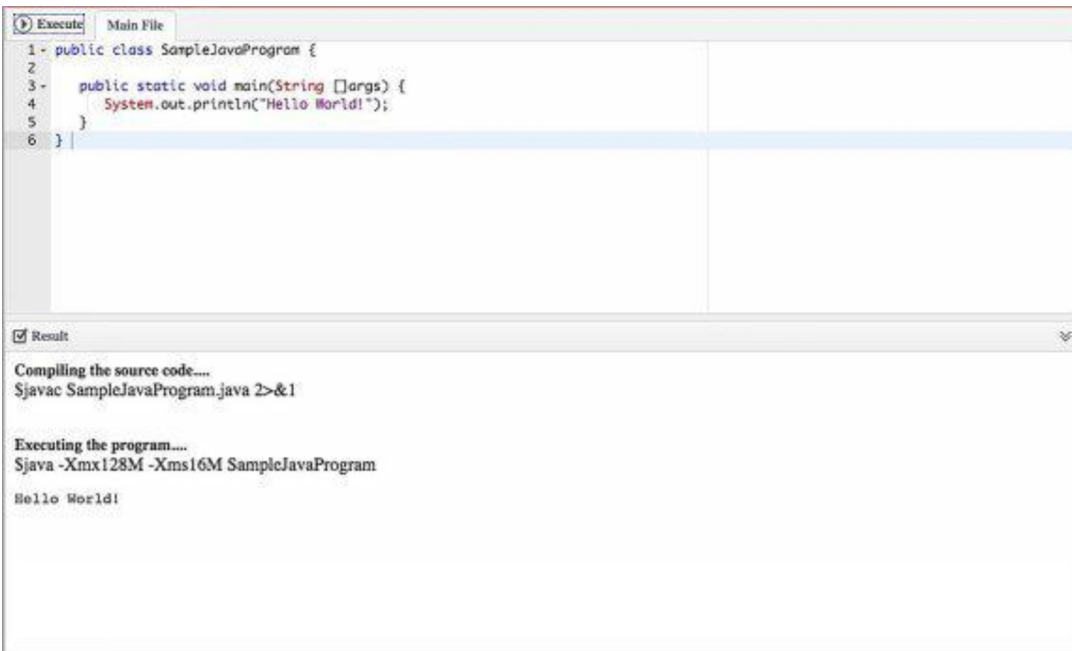
```
javac samplejavaprogram.java
```

-

```
java samplejavaprogram
```

-

The first command will create a class file from the java file while the second command will execute the program to give the desired result. A sample run of the code has been illustrated in the image shown below.



The screenshot shows a Java IDE with two main panels. The top panel, titled 'Main File', contains the source code for a class named `SampleJavaProgram`. The code is as follows:

```
1- public class SampleJavaProgram {  
2-  
3-     public static void main(String []args) {  
4-         System.out.println("Hello World!");  
5-     }  
6- }
```

The bottom panel, titled 'Result', shows the output of the compilation and execution. It contains the following text:

```
Compiling the source code....  
$javac SampleJavaProgram.java 2>&1  
  
Executing the program....  
$java -Xmx128M -Xms16M SampleJavaProgram  
Hello World!
```

If you have reached so far, you have successfully executed your first java program. now it is time to inspect the program to see how we wrote it and why

It gave the result that it did.

Firstly, writing a program in java requires you to follow some conventions. These are listed below –

java is a case-sensitive language. In other words, class and Class are two different words for java. Be careful when you are using keywords or any other elements in the program.

-

As a convention, the first letter of every word of the class name is supposed to be capital. For example, if the name of the class is car, the 'c' is supposed to be capital. On the other hand, if the name of the class is Sportscar, then S and c must be capital as both of them are first letters of the two words that form the name of the class.

-

Method names also follow a capitalization convention according to which the first letter of the method name must be small in case. However, if the name is made from more than one word, then except for the first word, every other word's first letter must be capital. For example, if the method's name is add, then 'a' must be small, but if the method's name is addNumbers, then 'a' will be small and 'N' will be capital.

-

The name of the java file must be same as the name of the class, followed by .java extension. For example, if the name of your class is sampleApplication, then the name of the java file must be sampleApplication.java. Also remember that the case of the letter must also match. If you do not fulfill any of these requirements, you will not be able to compile the program.

-

The execution of a java program begins with the main method. Therefore, every java program must have a main method and the main method in java programming is defined as

-

```
Public static void main(String args[])
```

but again, if your program doesn't have a main method, the program will not run.

Identifiers

Any programming language is made up of English language words. Some of these words are a part of the syntax of the programming language while others like variable names and the user defines names of classes and methods. The

Words that are a part of the syntax are called keywords. On the other hand, all user-defined names are called identifiers.

There are some important things to know and remember about java. These include –

Names given to identifiers can begin with alphabets (lower case or upper case) and special characters – underscore (_) and currency (\$). This also means that an identifier's name cannot begin with a number or any other special character.

-

The use of a defined keyword as an identifier is not allowed.

-

Lastly, always remember that java is case-sensitive and `icount` and `ICount` are two different identifiers.

-

modifiers

Classes, methods and variables can be modified only using modifiers. There are two types of modifiers, which are as follows –

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

- •

Access modifiers determine how the class, method or variable can be accessed and who all have the permission to access them. Other modifiers, which do not work on similar lines are classified as non-access modifiers.

Variables

Java supports three types of variables namely class variable, local variables and instance variables. Class variables are also called static variables while instance variables have been termed as non-static variables.

Arrays

A collection of elements of the same type is called an array. Java supports array objects of different types.

ENUMS

There are some variables that can hold only a few values. As a programmer, you may wish to restrict such variables from taking any other value. One such variable is gender, which can have only a selective number of values. ENUMS

Allows you to implement such variables. The use of `enums` is an error-reduction strategy and it was introduced in java 5.0. It is possible to define `enum` inside a class or outside it. moreover, `enums` can have constructors, variables and methods.

Keywords

The words that are part of the java programming language and cannot be used as identifiers are called keywords. These keywords are –

Assert	Abstract	break	boolean
Catch	Char	byte	Case
Const	Class	default	Continue
Else	ENUM	do	double
Final	Extends	Float	Finally
If	Implements	For	goto
InstanceOf	Import	Interface	Int
new	Package	Long	native
Protected	Private	Return	Public
Strictfp	Super	Short	Static
Synchronized	Switch	Throw	This
Try	Void	Throws	Transient
While	Volatile		

Comments

In java, there is support for single-line and multi-line comments. Single line comments start with `//` while multi-line comments start with `/*` and end with `*/`. You can look for these comments in the sample code given above.

Blank Lines

Any line that has a comment or no characters or whitespace characters is not considered by Java. It is simply ignored.

Implementing Inheritance

The inheritance characteristic of object-oriented programming is implemented in Java using the concept of deriving a new class from an existing class. This allows you to reuse existing code. The existing class is called a superclass while the derived class is referred to as a derived class.

Interfaces

In case two objects need to communicate with each other, then a protocol of communication needs to be established between these objects. This protocol of communication is called an interface. This concept is usually used in the implementation of the inheritance concept. An interface is used to define the methods that a subclass must have. However, it is the subclass that implements those methods.

Chapter 3: data types

When we declare variables, we basically block space in the memory and give it a reference name so that we can access it later and manipulate its contents. The amount of memory reserved for a variable depends on the data type given at the time of declaration. Therefore, by giving different data types to variables, you can reserve different amounts of memory for the variable in the memory. There are two main types in java, which are primitive datatypes and object datatypes.

Primitive data types

java supports eight primitive data types and there is a keyword corresponding to each of these data types to allow declaration of variables under this data type. These data types include –

Byte

maximum value = 127 (inclusive) $(2^7 - 1)$

-

minimum value = -128 (-2^7)

-

8-bit signed 2's complement integer

-

default value = 0

-

This data type is generally used in cases where memory is to be saved.

-

byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an integer.

-

Short

maximum value = 32,767 (inclusive) $(2^{15} - 1)$

-

default value = 0.

-

minimum value = $-32,768$ (-2^{15})

-

16-bit signed 2's complement integer

-

It is another data type that is used when memory saving is desired.

-

Int

default value = 0

-

maximum value = $2,147,483,647$ (inclusive) ($2^{31} - 1$)

-

32-bit signed 2's complement integer. MINIMUM value
= - 2,147,483,648 (-2^{31})

- •

This is the default data type used for all integral values.

-

Long

default value is 0L

-

MINIMUM value = -9,223,372,036,854,775,808 (-2^{63})

-

64-bit signed 2's complement integer

-

maximum value = 9,223,372,036,854,775,807 (inclusive) ($2^{63} - 1$) this is used when
the variable is expected to hold large NUMBERS.

- •

Float

default value = 0.0f

-

32-bit IEEE 754 floating point

-

Single-precision

-

This data type is used as the default data type for storing floating
point NUMBERS.

-

The use of this data type for storing values that require precise
calculations MUST BE avoided.

-

double

64-bit IEEE 754 floating point

-

double-precision

-

default value = 0.0d

-

This data type is used for variables that need to store decimal values. however, like float, the value of variables stored using double may vary. Thus, these data types are NOT used for critical calculations like that CONCERNING currency.

-

boolean

default value = false

-

1 Bit

-

Possible values: true and False

-

This data type is used for implementing flags.

-

Char

maximum value = '\uffff' (or 65,535 inclusive)

-

minimum value = '\u0000' (or 0)

-

16-Bit Unicode character

-

This data type is specifically used for storing characters.

-

Reference data types

As the name suggests, reference variables are variables that are used for accessing an object. They are usually created using class constructors and their type, once defined, cannot be changed. Array variables and class objects are reference variables. Lastly, the default value in this case is null.

Java Literals

Literal is used to represent a fixed value. They are declared like variables and have a data type. Sample declaration of a literal is –

Char x = 'X';

You may notice integers and float with different bases. As a rule, any literal value that has 0, as prefix is octal numbers while if 0x prefix indicates hexadecimal numbers. If you are working with strings, then string literals are always defined as sequence of characters enclosed within double inverted commas. Another set of characters that you will work with is escape characters. A list of escape characters has been given below –

Carriage Return (\r)

-

newline (\n)

-

backspace (\b)

-

Formfeed (\f)

-

- Tab (\t)
- Space (\s)
- Single quote (\')
- double quote (\")
- hexadecimal character (\uxxxx)
- Octal character (\ddd)
- Backslash (\\)
-

Chapter 4: Variables

The simplest definition of a variable is that it is a name given to some memory area. However, there are many types of variables, which Java supports. On the basis of the type of variable, the following parameters are determined –

- Size of memory blocked for the variable.
- Layout of the blocked memory.
- Operations that can be performed on the variable.
- Range of values that the variable can hold.

As a rule, every variable needs to be declared before use. The standard statement for variable declaration is –

`data_type variable_name;`

Here, `data_type` is the data type of the variable while `variable_name` is the identifier used for the variable. If you are using multiple variables of the same type, you can club them in the same statement by specifying the name of the variables in a comma-separated manner. As mentioned previously, Java supports three types of variables namely, class variable, instance variable and local variables. This chapter explains these variable types in detail.

Local Variables

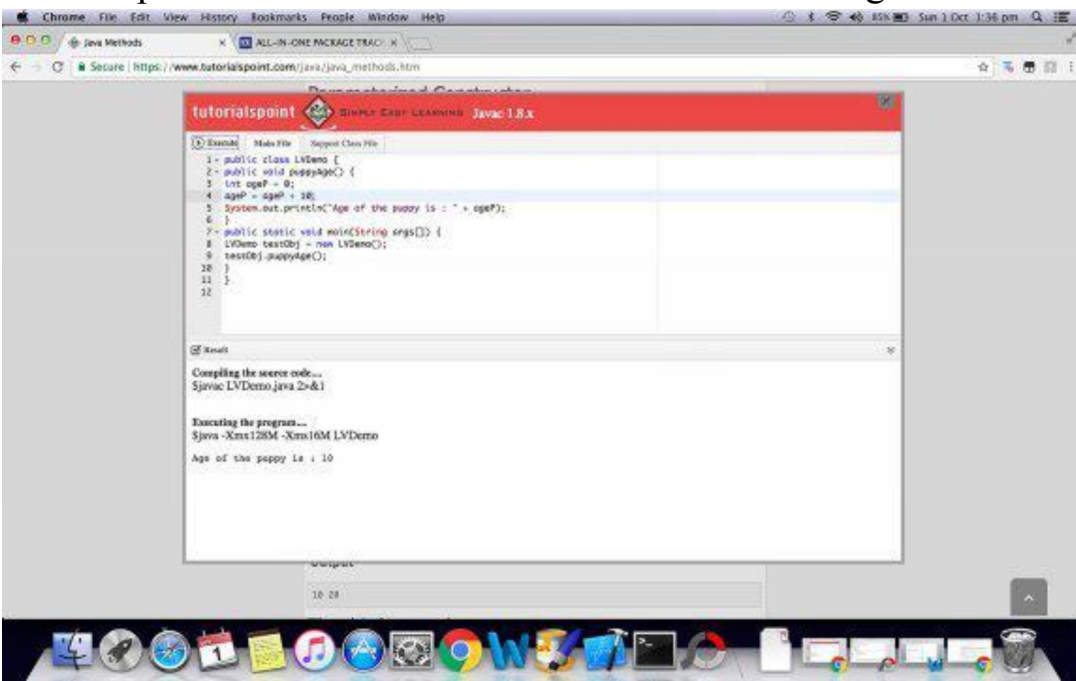
Variables that are declared inside a method or block of code are local to the scope and are called local variables. The variable is created as and when it is declared and defined in the block. Moreover, the variable is destroyed once the scope of the block gets over. Variables of this type cannot make use of access modifiers (described later in this chapter).

Since, this type of variable is alive only within the scope of the method or block, it can only be accessed within this block of code. Internally, local variables are created at the stack-level. So, when they are created, they are pushed onto stack and as soon as the block finishes execution, the stack is emptied. Lastly, the default value for local variables is garbage. So, these variables have to be initialized for use.

Sample code to illustrate the working of local variables has been given below. It implements a local variable age in the method in the class, which is accessed from the main method.

```
Public class lvdemo {  
  
    Public void puppyage() {  
  
        Int agep = 0;  
  
        Agep = agep + 10;  
  
        System.out.println("Age of the puppy is : " + agep);  
  
    }  
  
    Public static void main(String args[]) {  
  
        lvdemo testobj = new lvdemo();  
  
        Testobj.puppyage();  
  
    }  
  
}
```

The output of this code has been illustrated in the image shown below.



Instance Variables

Variables that are created as part of the class, but do not lie within the scope of any method or constructor are referred to as instance variable. The reason why these variables are called instance variables is that they are created only once the object is created using that class. Therefore, these variables are created for every object instantiated for this class using the new keyword.

Moreover, these variables are destroyed as soon as the object is deleted or the program execution terminates. These variables can be accessed from the method or constructors implemented for this class. It would not be wrong to state that instance variables are class-level variables that can make use of access modifiers.

These variables have default values. For instance, the default values for all numbers is 0 and that for boolean is false. The object references created as reference variables are initialized as null. While declaration of variables is a must, assignment can be done inside the methods or constructors. If you are working inside the same class, you can directly access the variable by its name. However, if you wish to access it from any static method, then you will need to access it using the format –

`Object_NAME.variable_NAME`

Sample code has been given below to help you understand this concept better. This class implements a class for employees and initializes the variables of the class using initialization methods.

```
Import java.io.*;
```

```
Public class demoemployee {
```

```
    Public String nameemp;
```

```
    Private double salaryemp;
```

```
    Public demoemployee (String initname) {
```

```
        nameemp = initname;
```

```
    }
```

```
    Public void initsalary(double initsal) {
```

```
        Salaryemp = initsal;
```

```

}

Public void printemployee() {

    System.out.println("Employee   name   =   "   +   nameemp   );
    System.out.println("Employee Salary = " + salaryemp);

}

Public static void main(String args[]) {

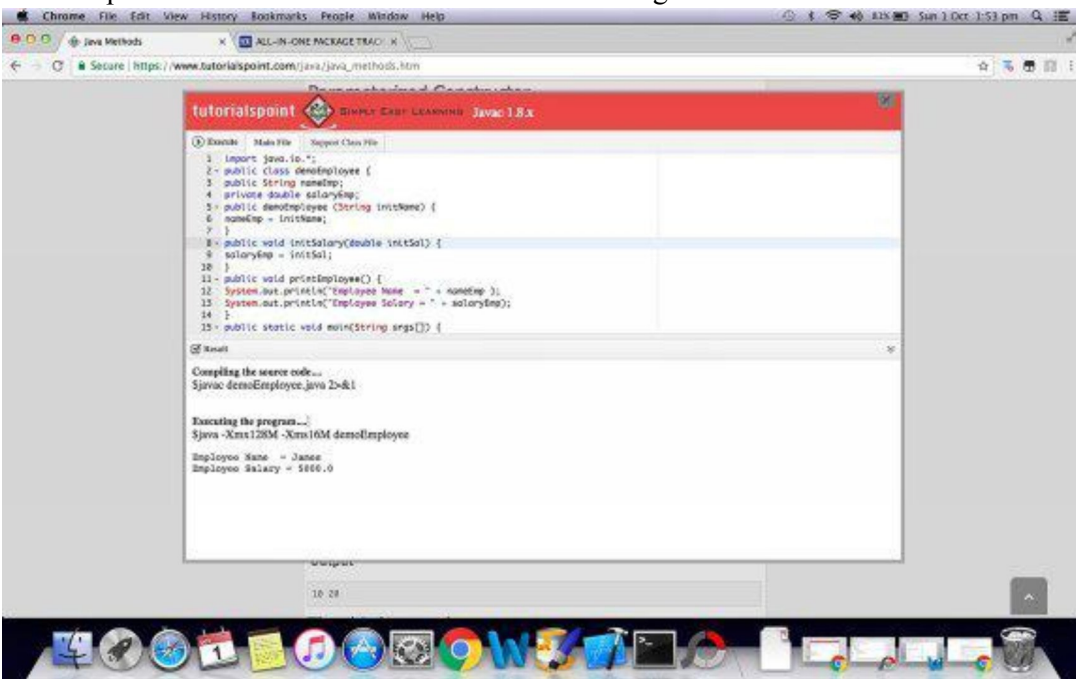
    demoemployee empobj = new demoemployee("James");
    empobj.initsalary(5000);
    empobj.printemployee();

}

}

```

The output of this code has been illustrated in the image shown below.



Static Variables

Any variable that is declared with the keyword `static` and is placed within the class, but outside the methods, constructors or any other code block inside the

Class. Only one copy of this variable is maintained for the whole class irrespective of the number of objects that may be created for this class. Therefore, these variables are also referred to as class variable.

As a recommendation, these variables must not be used. However, there may be scenarios that require such an implementation. However, whenever such variables are used, you must use them as constants, for which values cannot be changed. These variables are created when the program execution starts. However, they are destroyed only once the program's execution terminates.

Class variables are visible for all elements of the class. Therefore, their visibility is similar to that of instance variables. In fact, the default values for class variables are also same as that of instance variables. The access to these variables can be made using the following statement –

Class_NAME.variable_NAME

Sample code for implementation of static variables is given below.

```
Import java.io.*;
```

```
Public class svdemo {
```

```
    Private static double salaryemp;
```

```
    Public static final String departmentemp = "devrd ";
```

```
    Public static void main(String args[]) {
```

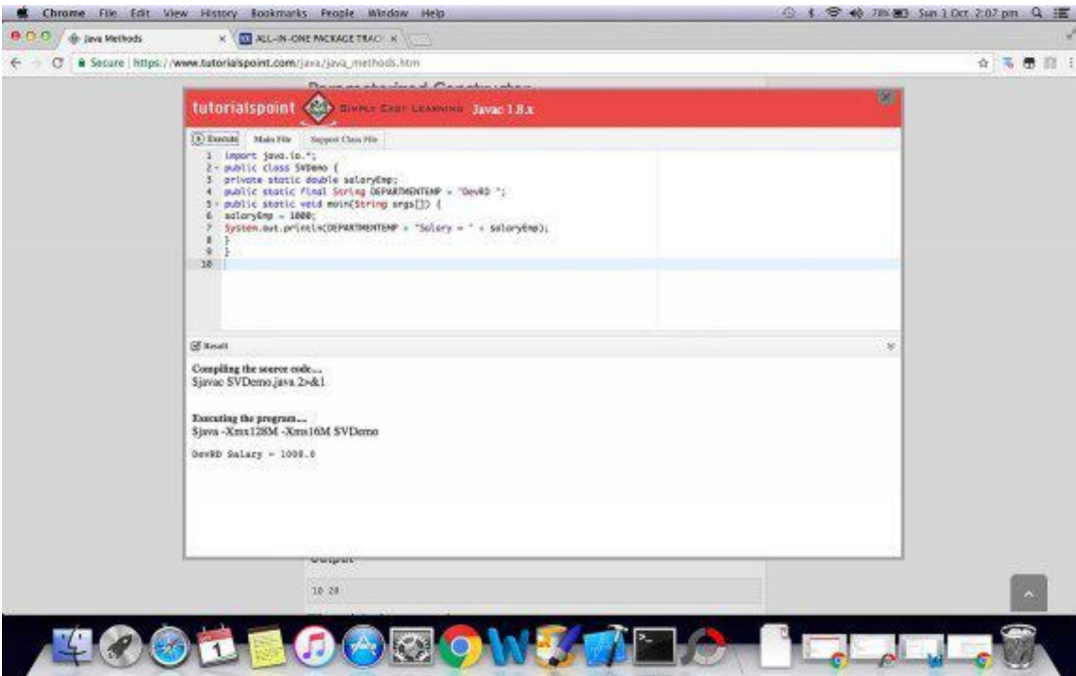
```
        Salaryemp = 1000;
```

```
        System.out.println(departmentemp + "Salary = " +  
        salaryemp);
```

```
    }
```

```
}
```

The output of the code can be seen in the image shown below.



modifier types

The behavior and access to the variables can be modified using modifiers. There are two main types of modifier types namely access modifiers and NON-access modifiers.

Access control modifiers

- java supports several access levels and each of this level has a corresponding keyword for it. These include –
 - Public
-

- Accessible to everyone
- Private
- Accessible to the class
- Protected
- Accessible to the class and its subclasses
- default
- If no keyword is mentioned, the variable is accessible to the package.
- NON-access modifiers
- There are other modifiers, which are also available in java. These include –

Static

-

Final

-

Abstract

-

Volatile and synchronized

-

Chapter 5: operators

Variables can be manipulated using operators. There are six classes of operators supported by Java. Each of these classes has been discussed in detail below.

Arithmetic operators

All mathematical operations are performed using operators that fall under the category of arithmetic operators. List of supported operators have been given below.

Addition (+).

This operator performs addition between two numbers. If a and b are two variables holding 5 and 6 respectively. Then, $c = a + b$, will yield $c = 11$.

Multiplication (*).

This operator performs multiplication between two numbers. If a and b are two variables holding 5 and 6 respectively. Then, $c = a * b$, will yield $c = 30$.

Subtraction (-).

This operator performs subtraction between two numbers. If a and b are two variables holding 5 and 6 respectively. Then, $c = a - b$, will yield $c = -1$.

Division (/).

This operator performs division between two numbers. If a and b are two variables holding 10 and 5 respectively. Then, $c = a / b$, will yield $c = 2$.

Modulus (%).

This operator returns the remainder left after dividing the two numbers. If a and b are two variables holding 10 and 5 respectively. Then, $c = a \% b$, will yield $c = 0$.

Increment (++).

This operator adds 1 to the value of the variable. If a is a variable holding the value 1, then $a++$ will yield the value 2.

Decrement (--).

This operator reduces 1 from the value of the variable. If a is a variable holding the value 1, then `a--` will yield the value 0.

Relational operators

All operations involving comparison are performed using relational operators. A list of the operators supported by this programming language is given below –

not EQUAL to (\neq).

This operator is applied on two operands. It checks equality between two operands. If they are equal, it returns FALSE else it return true.

EQUAL to ($=$).

This operator is applied on two operands. It checks equality between two operands. If they are equal, it returns true else it return FALSE.

greater than ($>$).

This operator is applied on two operands. It checks equality between two operands. If the first operand is greater than the second operand, it returns true else it return FALSE.

LESS than ($<$).

This operator is applied on two operands. It checks equality between two operands. If the first operand is less than the second operand, it returns true else it return FALSE.

greater than or EQUAL to

This operator is applied on two operands. It checks equality between two operands. If the first operand is greater than or equal to the second operand, it returns true else it return FALSE.

LESS than or EQUAL to

This operator is applied on two operands. It checks equality between two operands. If the first operand is less than or equal to the second operand, it returns true else it return FALSE.

Bitwise operators

The only difference between normal operators and bitwise operators is that these operators perform operations bit-by-bit. Moreover, these operators can be applied on byte, int, short, long and char. The list of bitwise operators supported in java include –

Bitwise And (&).

This operator works ON two operands and the output for each set of input bits is 1 if BOTH the inputs are one else the output is zero.

Bitwise or (|).

This operator works ON two operands and the output for each set of input bits is 1 if ONE or BOTH of the inputs is one else the output is zero.

Bitwise xor (^).

This operator works ON two operands and the output for each set of input bits is 1 only if ONE of the inputs is one else the output is zero.

Bitwise complement (~).

This operator works ON a single operand and negates the values from 0 to 1 and from 1 to 0.

Left shift (<<).

This operator works ON two operands. The bits of the left operand are shifted left. The NUMBER of places is equal to the NUMBER specified as value for the second operand.

Right shift (>>).

This operator works ON two operands. The bits of the left operand are shifted right BY the NUMBER specified as value for the second operand.

Zero FILL right shift (>>>).

This operator works ON two operands. The bits of the left operand are shifted right BY the NUMBER specified as value for the second operand. moreover, 0 replaces the bits that are shifted.

IN order to understand how bitwise operations take place, let us take an example. Assume two variables, a and B, having the values 60 and 13 respectively. The BINARY equivalent of a and B are as follows –

A = 00111100

B = 00001101

Using the AND truth table according to which the output is 1 only if BOTH the inputs are 1. Correspondingly, the output of $a \& B = 00001100$. Similarly other bitwise operations can also be performed.

Logical operators

All the logical operations in java are performed using logical operators. The list of operators supported by this programming language is given below.

Logical And

This operator works on two operands. If both the operands are non-zero, the expression returns true else it returns FALSE.

Logical or

This operator works on two operands. If one or both the operands are non-zero, the expression returns true else it returns FALSE.

Logical not

This operator works on one operand. If the operand is zero, this expression returns true else it returns FALSE.

Assignment operators

The list of assignment operators supported by java

include – Simple Assignment operator (=).

This operator assigns the value computed by the right hand side to the variable on the left hand side. If a, b and c are three variables and the expression is $c = a + b$, then the value of addition of a and b is assigned to c.

Add And Assignment operator (+=).

This operator assigns the value computed by the addition of right hand side and left hand side to the variable on the left hand side. If a and c are two variables and the expression is $c += a$, then the value of addition of a and c is assigned to c.

Subtract And Assignment operator (-=).

This operator assigns the value computed by the subtraction of right hand side from left hand side to the variable on the left hand side. If a and c are two variables and the expression is $c -= a$, then the value of $c - a$ is assigned to c.

Multiply And Assignment operator (*=).

This operator assigns the value computed by the multiplication of right hand side and left hand side to the variable on the left hand side. If a and c are two variables and the expression is $c *= a$, then the value of $c * a$ is assigned to c.

Divide And Assignment operator (/=).

This operator assigns the value computed by the division of left hand side with right hand side to the variable on the left hand side. If a and c are two variable and the expression is $c /= a$, then the value of c/a is assigned to c.

modulus And Assignment operator (/=).

This operator assigns the value computed by the modulus of left hand side with right hand side to the variable on the left hand side. If a and c are two variable and the expression is $c \% = a$, then the value of $c\%a$ is assigned to c.

Left Shift And Assignment operator (<<=).

If a and c are two variable and the expression is $c << = a$, then the value of $c << a$ is assigned to c.

Right Shift And Assignment operator (>>=).

If a and c are two variable and the expression is $c >> = a$, then the value of $c >> a$ is assigned to c.

bitwise And Assignment operator (&=).

If a and c are two variable and the expression is $c \& = a$, then the value of $c \& a$ is assigned to c.

bitwise xor And Assignment operator (^=).

If a and c are two variable and the expression is $c \wedge = a$, then the value of $c \wedge a$ is assigned to c.

bitwise Inclusive or and Assignment operator (|=).

If a and c are two variable and the expression is $c |= a$, then the value of $c|a$ is assigned to c.

Operator Precedence

If an expression uses multiple operators, then operator precedence rules are used to determine the order to execution of operations. The precedence of operators in terms of whether they will be evaluated from right to left or left to right is defined for groups of operators in the following manner –

Right to Left

Unary

•

Conditional

•

Assignment

-

Left to Right

Postfix

- - multiplicative
- - Additive
- - Shift
- - Relational
- - Equality
- - bitwise AND
- - bitwise xor
- - bitwise or
- - Logical AND
- - Logical or
-

For example, if we have to evaluate the expression $a = 3 + 5 * 5$, then multiplication is performed before addition. Therefore, $a = 28$ instead of 40.

Chapter 6: objects and classes

To start with, let us understand the concept of classes and objects. Class is a template, which is used to create objects. Objects are real entities that interact and form the core element of the java program. Three types of variables can exist in a class. Firstly, variables are defined inside the class, but outside the method. Such variables are defined for each object and are termed as instance variables.

Besides this, variables can exist inside the method block of the class. These variables are private to the method and are local variables. Any variable that is declared within a class, outside the method block, using the word static is called class variable.

Creating objects

Evidently, as the class is a template, you will need to create objects using this template. In order to create objects, the keyword 'new' is used. Creation of an object involves three steps. The first step is to declare a variable for the object. The second step is to create an object using the new keyword. Lastly, the object needs to be initialized using the constructor for the class.

In case you omit the third step, java uses the default constructor method for object initialization. All these three steps are combined into one statement for object creation. For example, we have a class democlass and we wish to create an object o1, then this can be done using the statement –
democlass o1 = new democlass

The variables and methods of the class for which the object is created can be accessed using '.' operator. Therefore, if we have a class democo with the variable x and we create an object o1, then we can access x using the following statement: o1.x.

Constructors

When you get down to creating classes and objects, the first thing that you will need to do is initialize the object. Java has a dedicated method implementation for initialization, which is called constructor. These elements are implemented

Like methods, but they carry the same name as the class and have no return type.

So, where can you use a constructor? You can use it whenever you are creating an object of a class and you need to initialize some parameters or perform some function that is common to all the objects of the class. For example, if you create a class called rectangle. You can create a constructor to initialize the length and breadth of the rectangle.

If you don't define a constructor, Java uses the default constructor for your program. This constructor program initializes the value of all the parameters of the class to zero. So, if you have created a class, your program will have an active constructor irrespective of whether you define it explicitly or not.

Let us now create a class with an explicit constructor and see how the concept works.

```
Class Sampleclass1 {  
    Int i;  
    Sampleclass1() {  
        I = 100;  
    }  
}  
  
Public class demosampleclass1 {  
    Public static void main(String args[]) {  
        Sampleclass1 c1 = new  
        Sampleclass1(); Sampleclass1 c2 =  
        new Sampleclass1();  
        System.out.println(c1.i + "\n" + c2.i); }  
}
```

The screenshot shown below illustrates the code and the output generated for the code upon execution. In order to run this code, you need to create two files, each with the name of the class it contains.

ExecuteMain FileSupport Class File

```
1- public class DemoSampleClass1 {
2-     public static void main(String args[]) {
3-         SampleClass1 c1 = new SampleClass1();
4-         SampleClass1 c2 = new SampleClass1();
5-         System.out.println(c1.i + "\n" + c2.i);
6-     }
7- }
8
```

Result

Compiling the source code....

```
$javac DemoSampleClass1.java SampleClass1.java 2>&1
```

Executing the program....

```
$java -Xmx128M -Xms16M DemoSampleClass1 SampleClass1
```

```
100
100
```

ExecuteMain FileSupport Class File

```
1- class SampleClass1 {
2-     int i;
3-     SampleClass1(){
4-         i = 100;
5-     }
6- }
7
```

Result

Compiling the source code....

```
$javac DemoSampleClass1.java SampleClass1.java 2>&1
```

Executing the program....

```
$java -Xmx128M -Xms16M DemoSampleClass1 SampleClass1
```

```
100
100
```

IN the example shown above, we gave the parameter static values. however, your program may require dynamic initialization of parameters. For this, you need to implement parameterized constructors. Such constructors accept the value in the form of a parameter to the constructor method. Sample implementation has been given below.

```
Class Sampleclass2 {
```

```
    Int i;
```

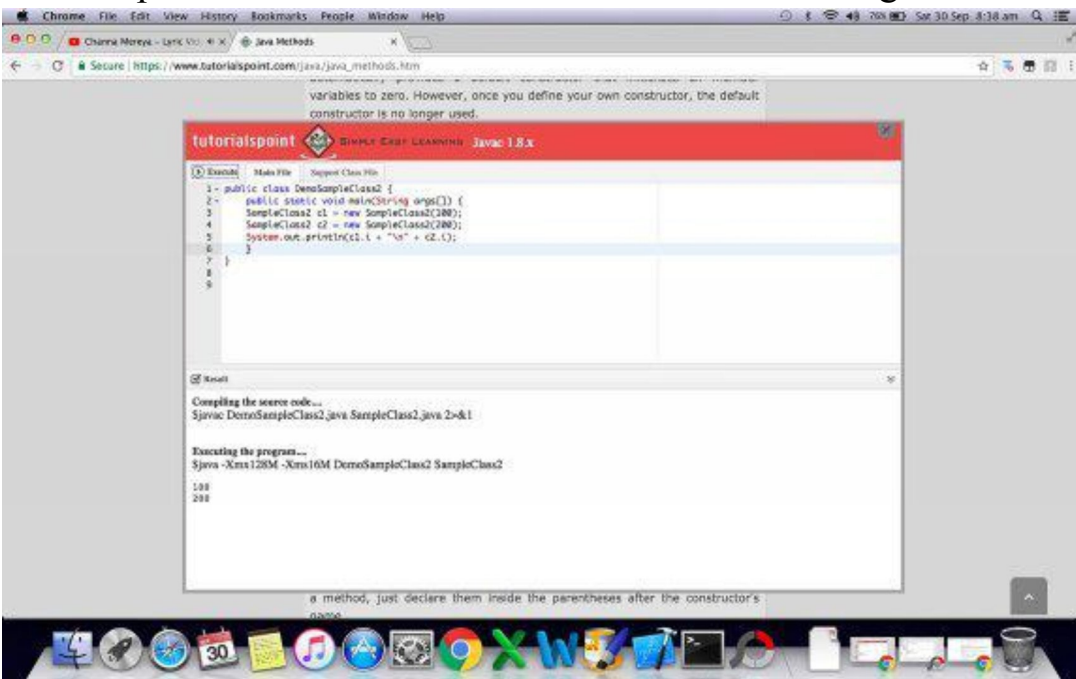
```

    Sampleclass2(int x) {
        I = x;
    }
}

Public class demosampleclass2 {
    Public static void main(String args[]) {
        Sampleclass2 c1 = new
        Sampleclass2(100); Sampleclass2 c2 =
        new Sampleclass2(200);
        System.out.println(c1.i + "\n" + c2.i); }
}

```

The output for this code has BEEN illustrated in the image shown below.



IN reference to the implementation of constructors, a keyword is specifically used. This keyword is the ‘ **this** ’ keyword. It is used to refer to the object of the class under consideration. With the help of this keyword, variables of the class can be accessed inside the constructor or methods. The use of this keyword has been illustrated in the code given below.

```
Class Sampleclass3 {
```

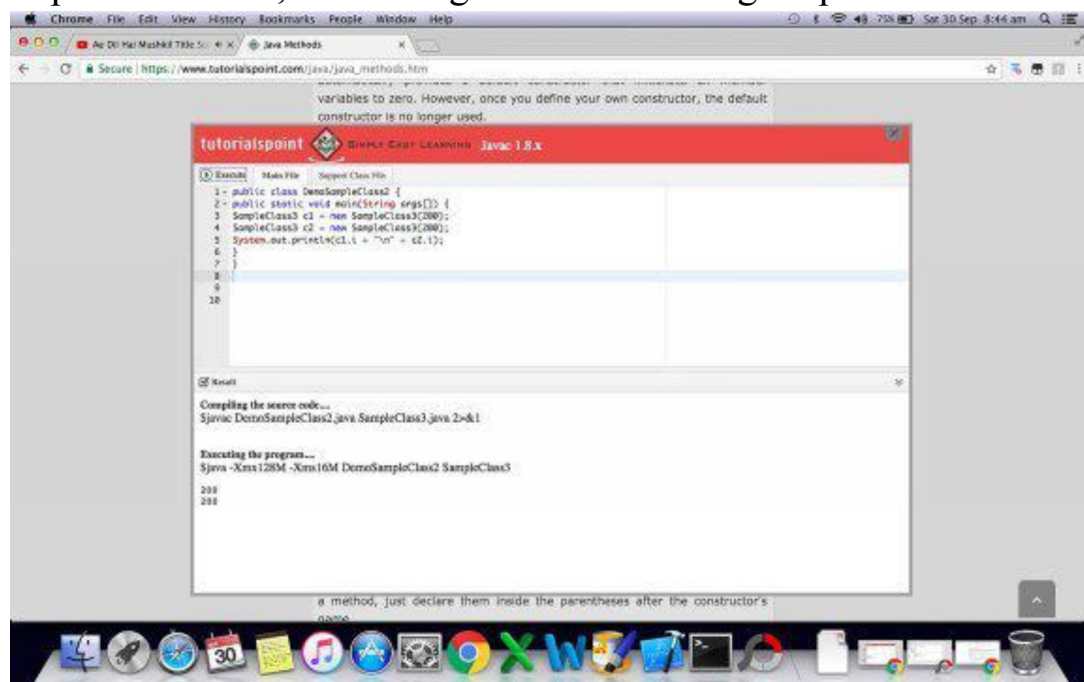
```

    Int i;
    Sampleclass3(int i) {
        This.i = i;
    }
}

Public class demosampleclass2 {
    Public static void main(String args[]) {
        Sampleclass3  c1    =    new    Sampleclass3(200);
        Sampleclass3  c2    =    new    Sampleclass3(200);
        System.out.println(c1.i + "\n" + c2.i);
    }
}

```

Upon execution, this code gives the following output –



Variable Arguments

In some programs, there may be a requirement where you don't know the exact number of arguments that the main method will give during execution. To allow dynamicity of this level, var-args or variable arguments are used. Var-args are declared in the method parameter using the syntax:

<type> ... <var_name>

The *var_name* given in the variable declaration is an array that contains all the variable values sent by the main method while executing. Before implementing this concept, you need to remember two things –

Only one var-args can be implemented per method.

•

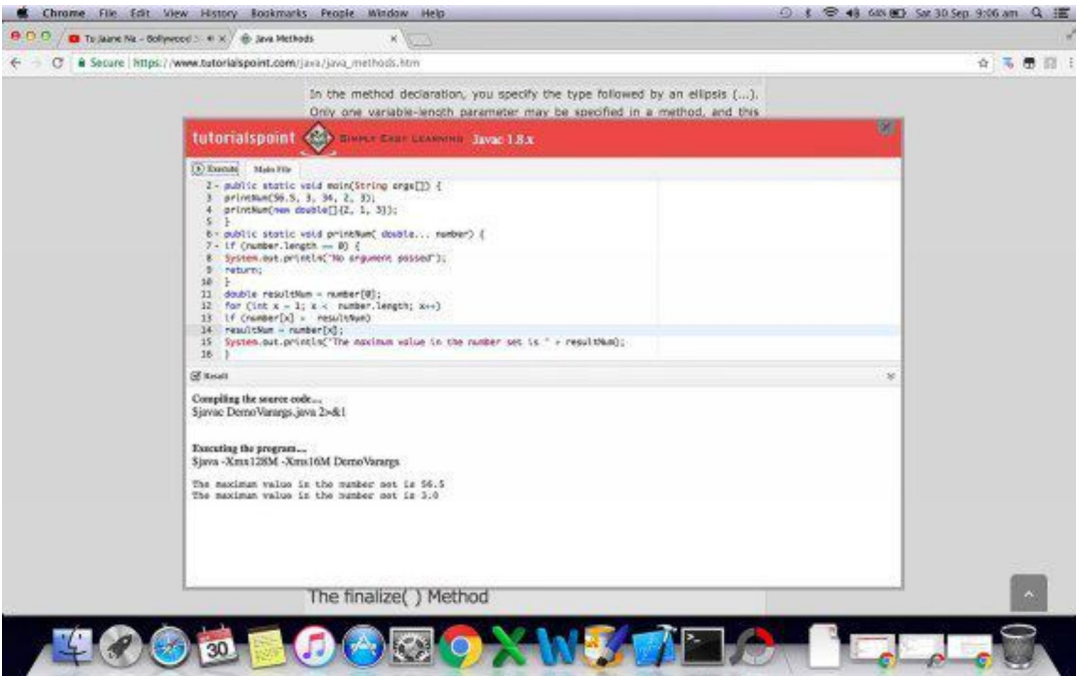
If the method has other arguments (regular in nature), these arguments must appear before the var-args. In other words, var-args must be the last argument in a method.

•

A sample implementation of this concept is given below.

```
Public class demovarargs {  
    Public static void main(String args[]) {  
        PrintNUM(56.5, 3, 34, 2, 3);  
        PrintNUM(new double[]{2, 1, 3});  
    }  
  
    Public static void printNUM( double...  
        NUMBER) { if (NUMBER.length == 0) {  
            System.out.println("no argument  
                passed"); return;  
        }  
        double resultNUM = NUMBER[0];  
        For (int x = 1; x < NUMBER.length; x++)  
            If (NUMBER[x] > resultNUM)  
                ResultNUM = NUMBER[x];  
        System.out.println("the maximum value in the NUMBER set is " +  
            resultNUM);  
        }  
  
    }
```


The output for the code is illustrated in the image shown below.



Implementing methods

A group of instructions that are used for performing an operation is referred to as a method. You can see that the print statement in most of the programs shown above calls a method `system.out.println()`. This method prints the string supplied to it on the screen.

Just like this inbuilt method, the developer can also create methods specific to the problem statement. User defined methods may or may not have parameters and may or may not return values. The standard statements for creating a method is –

Public static int methoddemo (int i, int j) {

```
/*Body*/
```

```
}
```

here, methoddemo is the name of the method and i and j are int parameters that this method accepts. The logic for the method implementation is given inside the curly brackets. Besides this, you shall also notice three words written in the function declaration – public, static and int. here, int is the return type of the method while public static is the modifier. This sample method can be modified to suit the requirements of the developer.

A sample code for method implementation is presented below

```
– public static int mincaldemo (int num1, int num2) {  
    Int minval;  
    If (num1 > num2)  
        minval = num2;  
    Else  
        minval = num1;  
    Return minval;  
}
```

This method determines which value is the smallest between the two parametric values supplied to it and returns this result to the calling method. Now that you have an idea about methods and how they are implemented, it is time we discuss about how these methods are used in the overall code.

Calling a method

A method needs to be called or invoked for it to be used. As you can see in the sample code given above, the method is performing some computation and returning the result of the computation. Such a method returns a value whose return type is already specified in the method declaration. Apart from this, another type of method also exists. Such methods do not return anything and are popularly referred to as methods with no return value.

When a program calls a method, the control moves from the program to the method and the same starts executing. Once a return statement is encountered, the control is transferred back to the calling program. In the absence of a return statement, the control is automatically transferred back once a closing brace of the method is encountered. This is usually the case for methods that have no return value.

For methods that return a value, the return type is mentioned in the method declaration. On the other hand, for methods that do not return anything, the return type is **void**. In order to call the method we implemented above, the following statement needs to appear somewhere in the calling program.

```
Int result;
```

```
Result = mincaldemo(5, 6);
```

Since the method returns an int value, this value needs to be gathered in an int variable (result). The call for the method gives the two parametric values desired by the method. This type of method calling is also called parametric method calling or passing parameters by value.

You may come across situations where you may have parameters of different types and many parameters in number. In such cases, be sure to give the parameter values in the desired order to avoid any uninvited compilation or runtime errors.

method overloading

It is possible to have methods with the same name, but different parameters. When a method is called, the system detects the method with the parameter type and number given in the calling statement and transfers control to the corresponding method. This form of method implementation is called method overloading. do not confuse this concept with overriding. In method overriding, the method name, parameters and number of parameters are also same.

In order to understand this concept, let us take an example –

```
Public class overloadingdemo {
```

```
    Public static void main(String[] args) {
```

```
        Int x = 34;
```

```
        Int y = 56;
```

```
        double z = 10.5;
```

```
        double u = 7.8;
```

```
        Int resultxy = mincaldemo(x, y);
```

```
        double resultzu = mincaldemo(z, u);
```

```
        System.out.println("min Value (Integers) = " + resultxy);
```

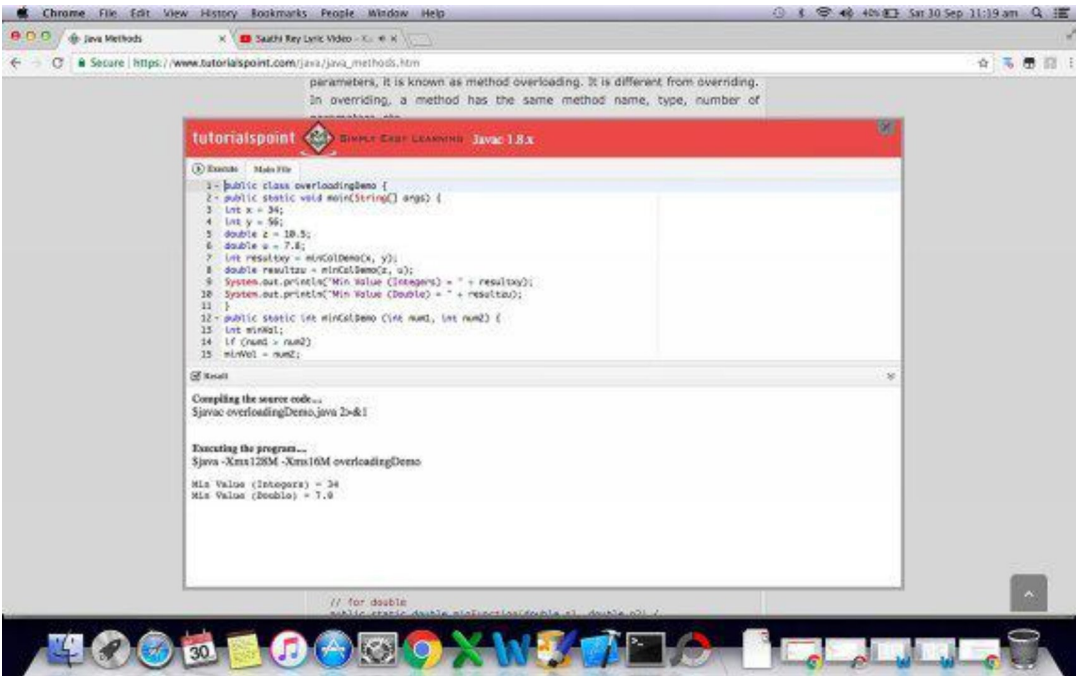
```
System.out.println("MIN Value (double) = " + resultz);
```

```

    }
    Public static int mincaldemo (int num1, int num2) {
        Int minval;
        If (num1 > num2)
            minval = num2;
        Else
            minval = num1;
        Return minval;
    }
    Public static double mincaldemo (double num1, double num2) {
        double minval;
        If (num1 > num2)
            minval = num2;
        Else
            minval = num1;
        Return minval;
    }
}

```

The code implements two methods with the same name mincaldemo with two different parameter types. One of the methods uses two parameters of int type and has the return type int. On the other hand, the other method implementation uses double parameters and returns a double. The output for this code has been illustrated in the image shown below.



making Use of command Line Arguments

There are some programs that require you to pass parameters while executing the program at runtime. Such parameters can be passed to the program in the form of command line arguments. These parameters are given to the program along with the name of the program while giving the execution command on the command line.

The manner in which these parameters can be accessed inside a program has been shown in the sample code given below.

```
Public class commandlineargdemo {  
  
Public static void main(String args[]) {
```



```
For(int x = 0; x<args.length; x++) {  
    System.out.println("args[" + x + "]: " + args[x]);  
}  
  
}
```

give the following command from the command line to execute the program and see what result you get.

\$java commandlineargdemo 200 100 10 8 7 6 this

is the finalize() method

You are dealing with a lot of objects in java programming and all these objects take up their space in the memory. If you are working on a large scale code, you may be required to clear up the space occupied by these objects before terminating the program. To facilitate this operation, a garbage collector method is available in java, which is called **finalize()** method.

This method can be used to implement all the operations that you would like the program to do before it terminates. Some of these operations include closing all the files opened, deleting all the objects created and releasing any other memory occupied by program elements. This method is implemented in the following manner –

```
Protected void finalize()  
                {  
  
    /*Implementation*/  
}
```

The access modifier ‘protected’ is used in the declaration of the finalize() method to prevent any external program from destroying the code even before it executes.

declaration Rules for Source File

There are some rules that you need to follow while writing import statement, class declarations and package statements in the source file. These rules are as follows –

- A source file cannot have more than one public class.
- There are be many non-public classes in a source file.
- The name of the source file should be same as the name of the public class in that file. moreover, the extension of the file should be .java.
- If you are working on a package and the class being implemented as part of the package, then the first statement in the source file must be a package statement.

-

The package statement must be the first statement in the source file. Any other import statements must follow this package statement. Lastly, the class implementation must be present.

-

It is important for you to understand that the package and import statements are common to all the classes in the source file. It is not possible to have different package and import statements for different classes in one source file.

-

Before we end this discussion, let us give you a closing statement on packages. It is just a form of classification for interfaces and classes. When you work on large Java projects, the use of packages helps in organizing the code and making it manageable as well as readable.

When you are using a class in a source file, but the implementation of the class is present in another file, then you will need to tell the compiler the location of this file. This is where import statements get their use. Using import statement, you link source files and classes that are used in the concerned source file, but their implementation is present in other files.

Chapter 7: decision making

The decision making structures are used in situations where a set of instructions have to be executed if the condition is true and another set of instructions have to be executed when the condition is determined to be false. There are several constructs available in java for programming such scenarios. These structures include –

If statement

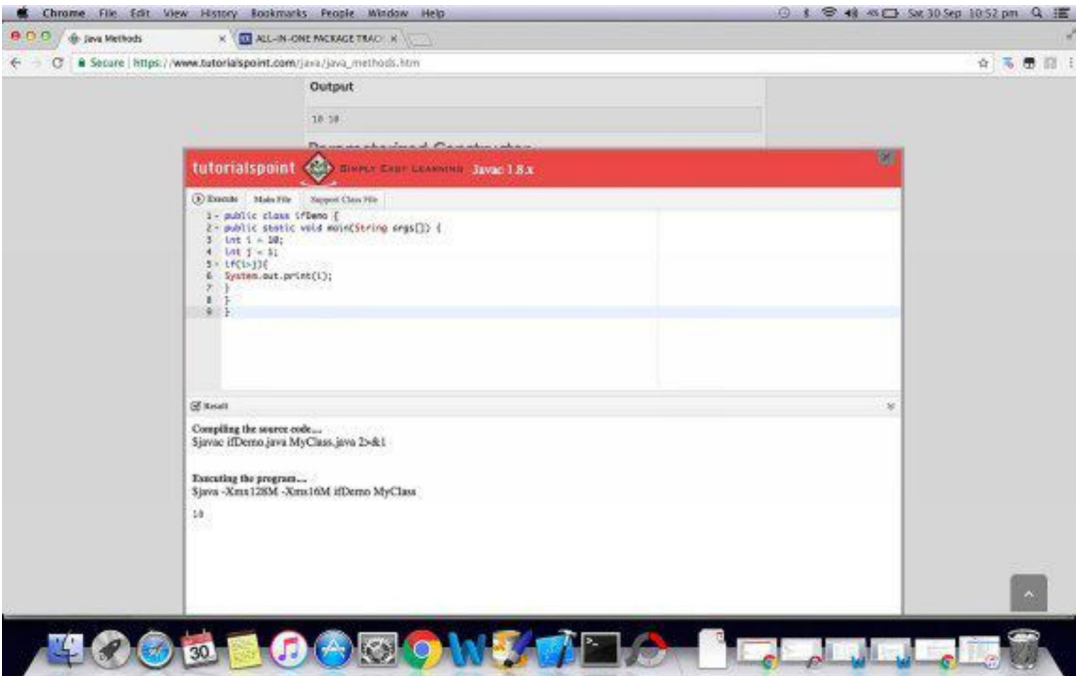
This statement is used in situations where a condition needs to be tested and if the condition is found true, the block of code that follows this statement needs to be executed. The syntax for this construct is –

```
If(condition){  
/*Body*/  
}
```

Sample implementation for this construct is given below –

```
Public class ifdemo {  
    Public static void main(String args[]) {  
        Int i = 10;  
        Int j = 1;  
        If(i>j){  
            System.out.print(i);  
        }  
    }  
}
```

The output of this code upon execution is as follows –



If else statement

This statement is used in situations where a condition needs to be tested and if the condition is found true, the block of code that follows this statement needs to be executed else the block of code that follows the else statement is executed. The syntax for this construct is –

If(condition){

*/*Body*/*

}

Else(

```
/*Body*/
```

```
}
```

Sample implementation for this construct is given below –

```
Public class ifelseDemo {
```

```
    Public static void main(String args[]) {
```

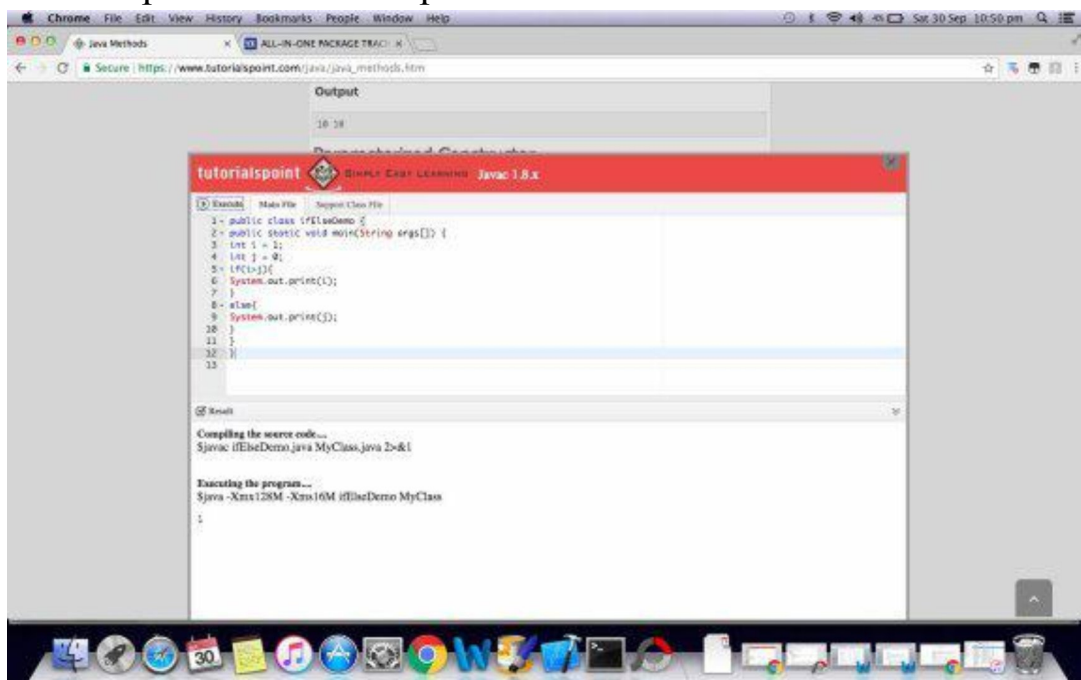
```
        Int i = 1;
```

```

Int J = 0;
If(i>J){
    System.out.print(i);
}
Else{
    System.out.print(J);
}
}
}

```

The output of this code upon execution is as follows –



Nested if statement

This statement is used in situations where a condition needs to be tested and if the condition is found true, the block of code that follows this statement needs to be executed else the next condition is tested. If this condition is found true, the block of code corresponding to the if statement for this condition is executed. If none of the conditions are found true, the block of code that

Follows the else statement is executed. Multiple conditions can be tested using the nested if statements. The syntax for this construct is –

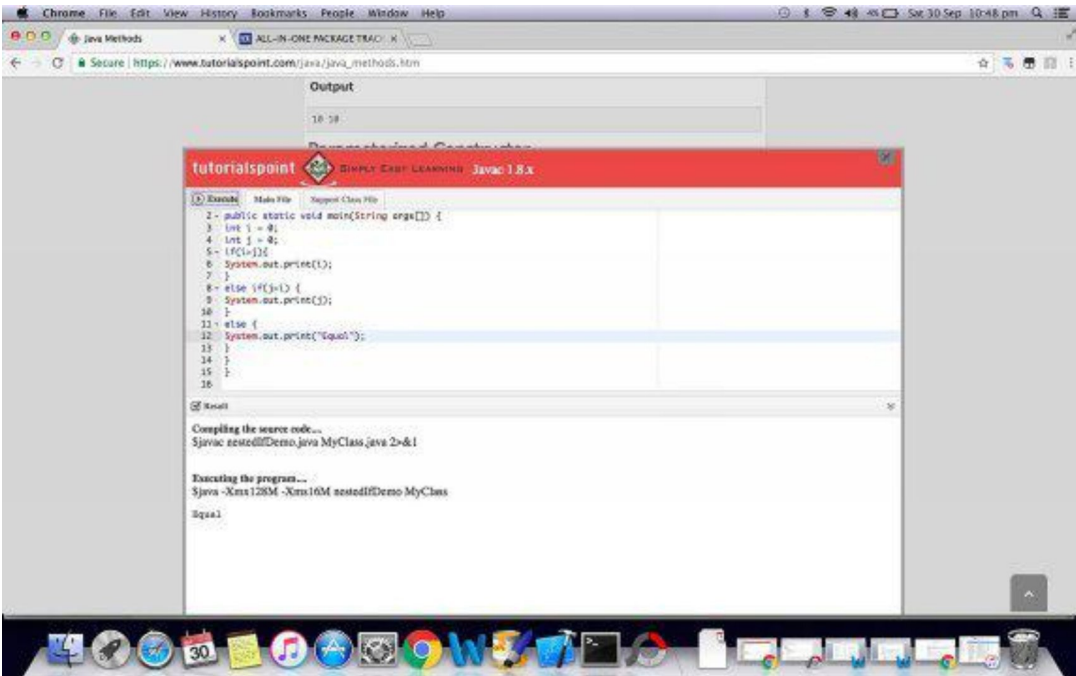
```
If(condition1) {  
    /*Body*/  
}  
Else if (condition2) (  
    /*Body*/  
}  
Else {  
    /*Body*/  
}
```

Sample implementation for this construct is given below –

```
Public class nestedifdemo {  
    Public static void main(String args[]) {  
        Int i = 0;  
        Int j = 0;  
        If(i>j){  
            System.out.print(i);  
        }  
        Else if(j>i) {  
            System.out.print(j);  
        }  
        Else {  
            System.out.print("Equal");  
        }  
    }  
}
```

}

The output of this code upon execution is as follows –



Switch

If you have a variable and different blocks of code need to be executed for different values of that variable, the ideal construct that can be used is the switch statement. The syntax for this construct is –

Switch(variable){

Case <value1>:

*/*Body*/*

break;

Case <value2>:

*/*Body*/*

break;

Case <value3>:

*/*Body*/*

break;

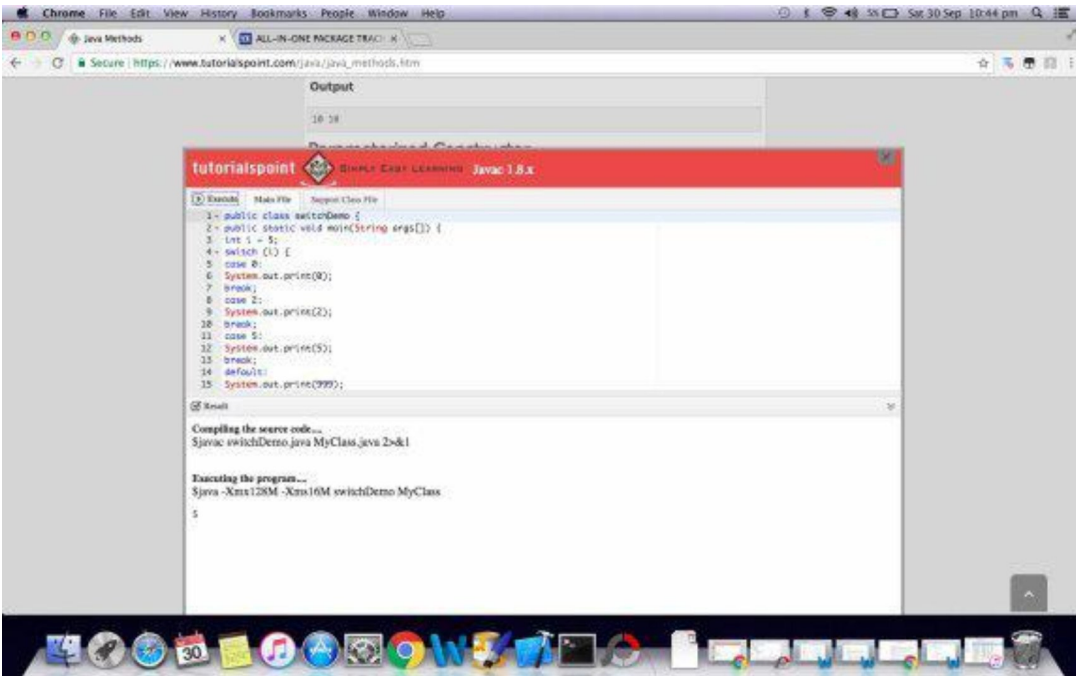
default:

```
    /*Body*/  
    break;  
}
```

Sample implementation for this construct is given below

```
– public class switchdemo {  
    Public static void main(String args[]) {  
        Int i = 5;  
        Switch (i) {  
            Case 0:  
                System.out.print(0);  
                break;  
            Case 2:  
                System.out.print(2);  
                break;  
            Case 5:  
                System.out.print(5);  
                break;  
            default:  
                System.out.print(999);  
                break;  
        }  
    }  
}
```

The output of this code upon execution is as follows –



Conditional operator

java also supports the conditional operator, which is also known as the ?: operator. This operator is used to replace the 'if else' construct. Its syntax is as follows –

Expression1 ? Expression2 : Expression3

here, Expression1 is the condition that is to be tested. If the condition is true, Expression2 is executed else Expression3 is executed.

Chapter 8: Loop control

There are several situations that require you to iterate the same set of instructions a number of times. For instance, if you need to sort a set of numbers, you will need to scan and rearrange the set several times to get the desired arrangement. This flow of execution is known as loop control.

Simply, a loop is a construct that allows execution of a block of code many times. Java supports several constructs that can be used for implementing loops. These include while loop, for loop and do while loop.

A while loop executes a block of code iteratively until the condition specified for the while loop is true. The moment this condition fails, while loop stops.

-

For loop allows the programmer to manipulate the condition and loop variable in the same construct. Therefore, you can initialize a loop variable, increment/decrement it and run the loop until a condition on this variable is true.

-

do while loop is similar to while loop. However, in the while loop, the condition is checked before executing the block code. On the other hand, in a do while loop, the block of code is executed and then the condition is checked. If the condition is satisfied, the loop execution is again initiated else the loop is terminated. It would not be wrong to state that the do while loop once implemented will execute at least once.

-

Loop statements

There are two keywords that are specifically used in connection with loops and are also termed as control statements as they allow transfer of control from one section of the code to a different section. These keywords are –

break

This keyword is used inside the loop at a point where you want the execution flow to terminate the loop and directly start execution from the first instruction that appears after the loop.

Continue

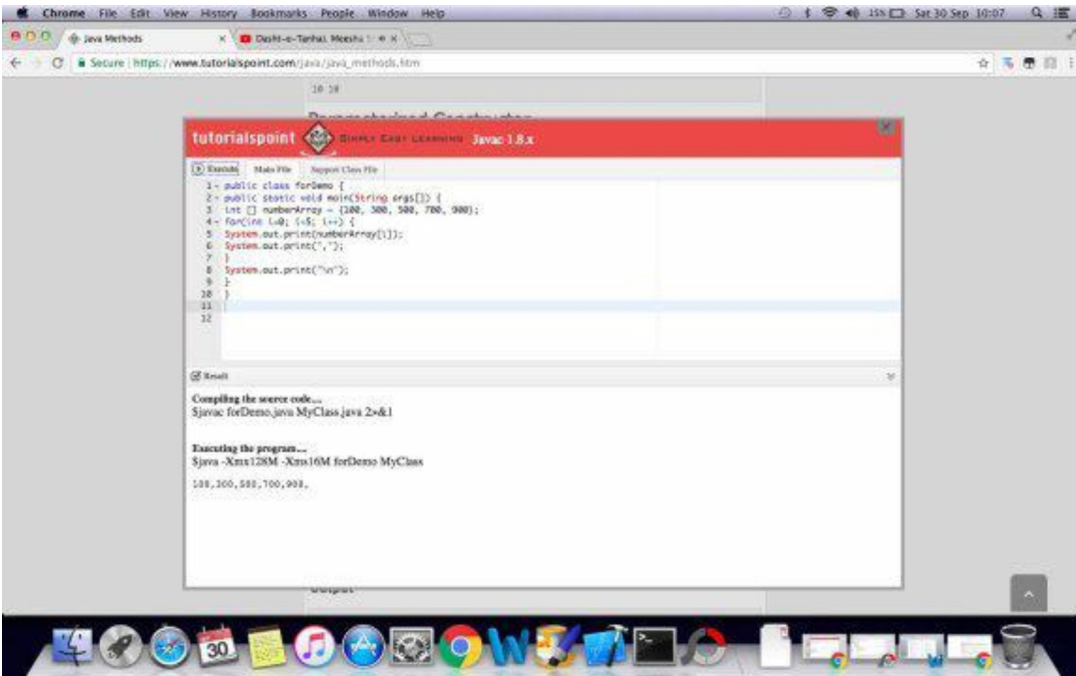
This keyword is used inside the loop at a point where the programmer wants the computer to overlook the rest of the loop and move the control to the first statement of the loop.

In order to help you understand how loops are executed, let us take an example and implement it using all the three types of loop control.

For Loop Implementation

```
Public class fordemo {  
  
    Public static void main(String args[]) {  
  
        Int [] numberarray = {100, 300, 500, 700, 900};  
  
        For(int i=0; i<5; i++) {  
  
            System.out.print(numberarray[i]);  
  
            System.out.print(", ");  
  
        }  
  
        System.out.print("\n");  
  
    }  
  
}
```

The output of this code upon execution is as follows –



While Loop Implementation

```
Public class whiledemo {
```

```
    Public static void main(String args[]) {
```

```
        Int [] NUMBERarray = {100, 300, 500, 700, 900};
```

```
        Int i = 0;
```

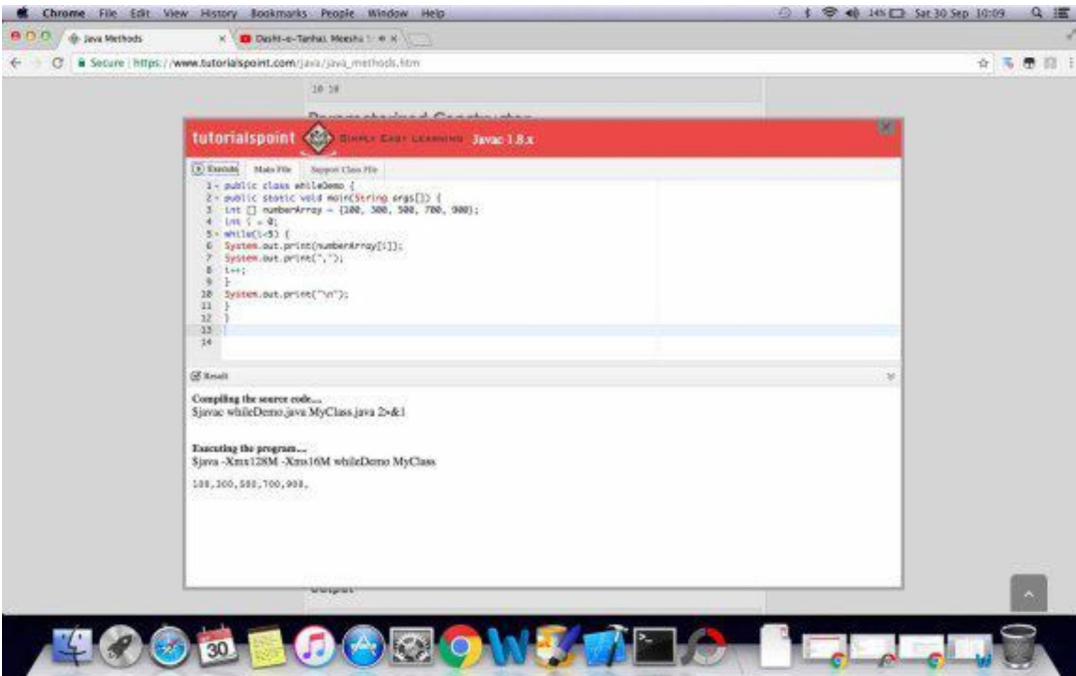
```
        While(i<5) {
```

```
            System.out.print(NUMBERarray[i]);
```

```
            System.out.print(", ");
```

```
        I++;  
    }  
    System.out.print("\n");  
}  
}
```

The output of this code upon execution is as follows –

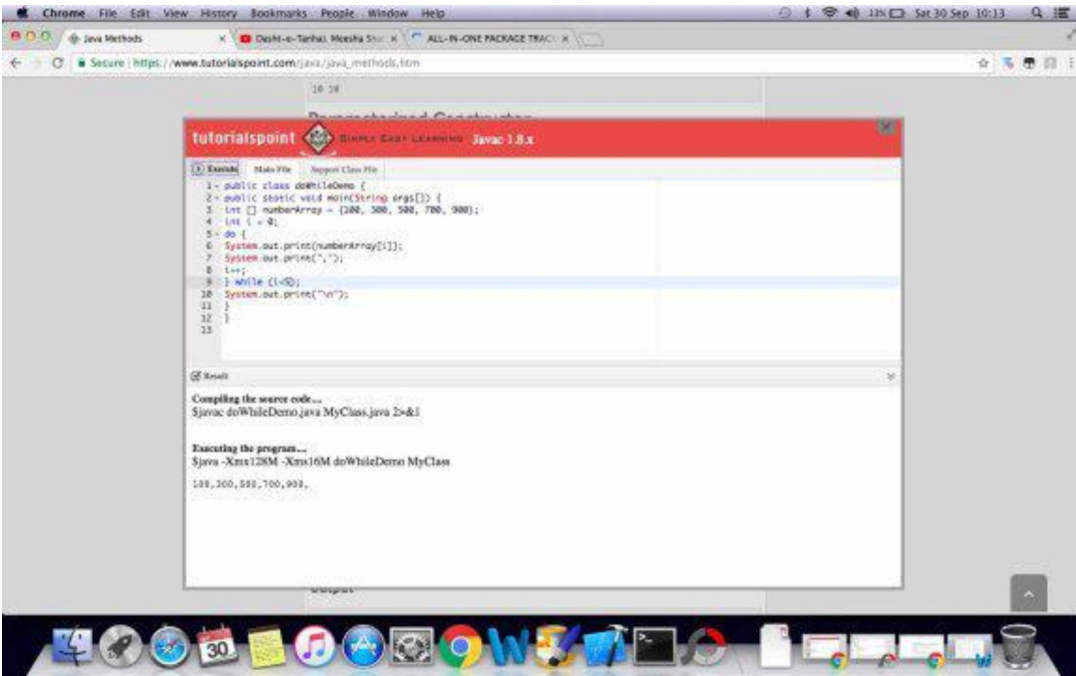


do While Loop Implementation

```
Public class dowhiledemo {  
    Public static void main(String args[]) {  
        Int [] NUMBERarray = {100, 300, 500, 700,  
        900}; int i = 0;  
        do {  
            System.out.print(NUMBERarray[i]);  
            System.out.print(", ");
```

```
        I++;  
    } while (i<5);  
    System.out.print("\n");  
}  
}
```

The output of this code upon execution is as follows –



Enhanced For Loop

java also supports an enhanced loop structure, which can be used for array elements. The syntax for this loop construct is –

```
For(declaration : expression) {  
  
/*Body*/  
  
}
```

The declaration part of the Enhanced for loop is used to declare a variable. This variable shall be local to the 'for loop' and must have the same type as the type of the array elements. The current value of the variable is

always equal to the array element that is being traversed in the loop. The expression is an array or a method call that returns an array.

Sample implementation of the enhanced for loop has been given

below - *public class forarraydemo {*

Public static void main(String args[]) {

Int [] numberarray = {100, 300, 500, 700, 900};

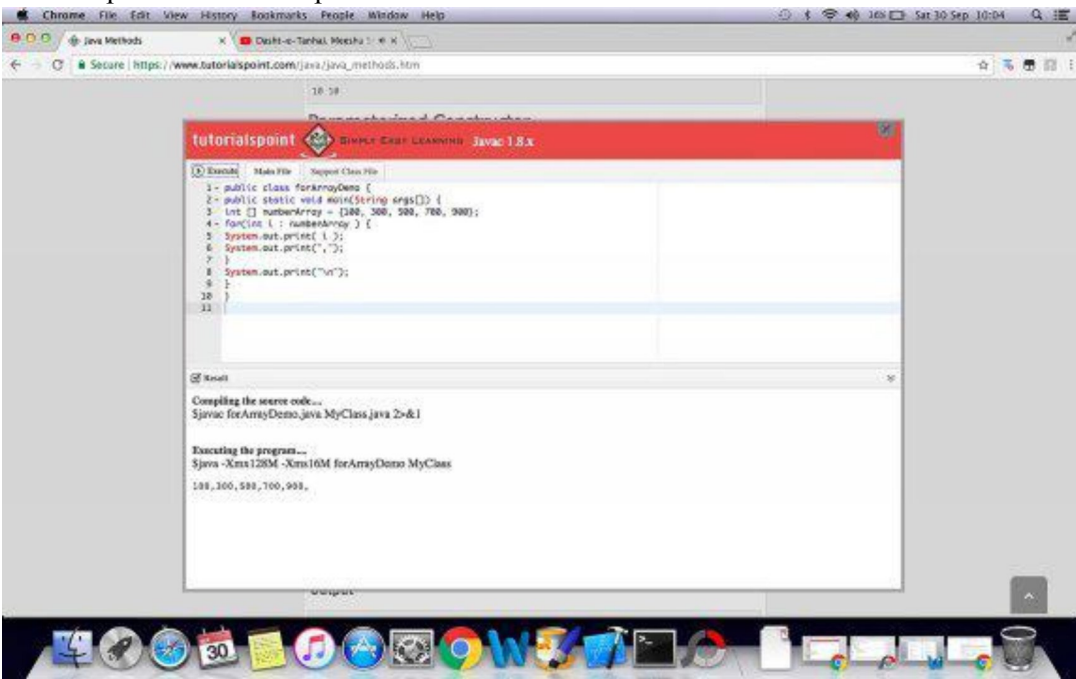
For(int i : numberarray) {

```

        System.out.print( i );
        System.out.print(",");
    }
    System.out.print("\n");
}
}

```

The output of this code upon execution is as follows –



Chapter 9: File handling

All the file operations are supported by java under the java.io package and the implementation of these functions is based on the concept of streams. Stream is simply a sequence of data, which is of two types namely, inputstream and outputstream. While the former is used to read data, the latter is used to write data.

Byte Streams

The first type of stream supported by java is byte stream. The I/o operations that are restricted to 8-bit size fall under this category. Before we get down to coding, you need to realize that the classes used for performing input and output are fileinputstream and fileoutputstream respectively. Let us now look at an example that copies the contents of one file to the other.

```
Import java.io.*;
```

```
Public class copyfileBS {
```

```
    Public static void main(String args[]) throws ioexception{
```

```
        Fileinputstream inputfile = null;
```

```
        Fileoutputstream outputfile = null;
```

```
        Try {
```

```
            Inputfile = new fileinputstream("in.txt");
```

```
            Outputfile = new fileoutputstream("out.txt");
```

```
            Int i;
```

```
            While ((i = inputfile.read()) != -1) {
```

```
                Outputfile.write(i);
```

```
            }
```

```
        } finally {
```

If (inputfile != null) {

$$\}$$

execute the code to see what it does.

Character Streams

MOVING FROM 8-BIT DATA OPERATIONS TO 16-BIT UNICODE DATA OPERATIONS, WE NEED TO SWITCH FROM BINARY STREAMS TO CHARACTER STREAMS. THE CLASSES USED FOR PERFORMING INPUT AND OUTPUT IS `filereader` AND `filewriter` RESPECTIVELY. THE CODE SHOWN ABOVE CAN BE MODIFIED TO USE CHARACTER STREAMS IN THE FOLLOWING MANNER.

```

Import java.io.*;

Public class copyfilecs {

Public static void main(String args[]) throws ioexception {

    Filereader inputfile = null;

    Filewriter outputfile = null;

    Try {

        Inputfile = new filereader("in.txt");

        Outputfile = new filewriter("out.txt");

        Int i;

        While ((i = inputfile.read()) != -1) {

```

Out.write(i);

```

    }
    }finally {
        If (inputfile != null) {
            Inputfile.close();
        }
        If (outputfile != null) {
            Outputfile.close();
        }
    }
}
}
}

```

Keep an input file named in.txt in the same folder as the class file and execute the code to see what it does.

Standard Streams

In order to allow the user to take input from the keyboard and print the output on the screen, java supports the standard streams –

- System.in represents the standard input stream and allows user to give input via the keyboard.
- System.out represents the standard output stream and allows user to get output on the screen.
- System.err represents the standard error stream and allows user to error messages from the system.
- The sample program given below allow the system to take input from the user and print the input given to the screen until the user hits the 'q' key after which the program quits.

Import java.io.;*

Public class ssdemo {

Public static void main(String args[]) throws ioexception{

```

InputStreamReader inputc = null;
Try {
    Inputc = new inputStreamreader(System.in);
    System.out.println("Enter characters, 'q' to quit.");
    Char charin;
    do {
        Charin = (char) inputc.read();
        System.out.print(charin);
    } while(charin != 'q');
}finally {
    If (inputc != null) {
        inputc.close();
    }
}
}
}
}

```

This program can be executed to see what the output is and how the code works.

Chapter 10: Exception handling

Exception is any event that is unexpected, usually in manner that cannot be handled by the program concerned. Obviously, the normal flow of execution of the program is hampered in such a situation and the program may either terminate with an error or behave abnormally. This is the reason why exceptions need to be handled by the program only.

So, why do exceptions occur? Exceptions may occur because of a range of reasons. Some of the scenarios that may lead to this inadvertent situation include –

- Invalid data entry
- The program is trying to open a file that is not present at the specified location
- The JVM has fallen short of memory
- network communication is being carried out and the system loses network connection.

Evidently, exceptions may occur due to problems created by the system, user or issues at the program level. On the basis of this observation, three classes of exceptions have been created. These three classes of exceptions have been discussed in chapter.

Checked Exceptions

Exceptions that are checked and reported by the compiler at the time of compilation are termed as checked exceptions. They are also called compile time exceptions and need to be handled by the developer during the time he or she writes code.

An example of how exception handling for such an error is performed is shown in the sample code given below.

```
Import java.io.File;
```


Import java.io.filereader;

Public class demoeh {

```

    Public static void main(String args[]) {
        File filedemo = new File("E://samplefile.txt");
        FileReader inpfiler = new FileReader(filedemo);
    }
}

```

If the filename and location specified while creating filedemo handler is not present at the specified location, then the compiler gives the error *FileNotFoundException*.

Unchecked Exceptions

There are some exceptions that happen at runtime. Such exceptions are called unchecked exceptions or runtime exceptions. Logical errors, programming bugs and wrong usage of APIs are examples of this type of error. Another important characteristic of this type of errors is that they are not detected at the time of compilation.

To help you understand what runtime errors are, let us take an example. Consider that you have implemented an array of 5 elements as part of your program. However, part of the code is accessing the 6th element. This error will be detected at runtime and the system will generate the error *ArrayIndexOutOfBoundsException*.

Errors

There are some exceptions, which are beyond the control of the system or the programmer. A common example of such errors is stack overflow. There is hardly anything that you can do about these errors. So, it is best to ignore them. As and when they occur, they will interrupt the execution of your program, but nothing can be done about it.

Exception hierarchy

Java has an inbuilt class named `java.lang.Exception` and all the exceptions fall under this class. All the exception classes are subclasses of this class.

moreover, throwable class is the superclass of the exception class. Another

Subclass of the throwable class is the Error class. All the errors like stack overflow described above fall under this Error class.

The Exception class has two subclasses namely runtimeexception class and ioexception class. A list of the methods, for which definitions are available in java, as part of the throwable class, is given below.

Public String getMessage().

When called, this message returns a detailed description of the exception that has been encountered.

Public throwable getCause().

This method when called returns a message that contains the cause of the exception.

Public String toString().

This method returns the detailed description of the exception concatenated with the name of the class.

Public void printStackTrace().

The result of toString() along with a trace of the stack can be printed to the standard error stream, System.err, can be done by calling this method.

Public stacktraceelement [].getStackTrace().

There may be some scenarios where you may need to access different elements of the stack trace. This method returns an array, with each element of the stack trace saved to different elements of the array. The first element of array contains the top element of the stack trace while the bottom of the stack trace is saved to the last element of the array.

Public throwable fillInStackTrace().

Appends the previous information in the stack trace with the current contents of the stack trace and returns the same as an array.

Catching Exceptions

The standard method to catch an exception is using the 'try and catch' keywords along with their code implementations. This try and catch block needs to be implemented in such a manner that it encloses the code that is expected to raise an exception. It is also important to mention here that the

Code that is expected to raise an exception is termed as protected code. The syntax for try and catch block implementation is as follows –

```
Try {  
    /*Protected code*/  
}catch(ExceptionNAME exc1) {  
    /*catch code*/  
}
```

The code that is expected to raise an exception is placed inside the try block. If the exception is raised, then the corresponding action to be performed for exception handling is implemented in the catch block. It is imperative for every try block to either have a catch block or a finally block.

As part of the catch statement, the exception, which is expected to be raised needs to be declared. In the event that an exception occurs, the execution is transferred to the catch block. If the raised exception matches the exception defined in the catch block, the catch block is executed.

A sample implementation of the try and catch block is given below. The code implements an array with 2 elements. however, the code tries to access the third element, which does not exist. As a result, an exception is raised.

```
Import java.io.*;  
  
Public class demoexception {  
    Public static void main(String args[]) {  
        Try {  
            Int arr[] = new int[2];  
  
            System.out.println("Accesing the 3rd element of the array:" +  
arr[3]);  
        }catch(arrayindexoutofBoundsexception exp) {  
            System.out.println("catching Exception:" + exp);  
        }  
    }  
}
```

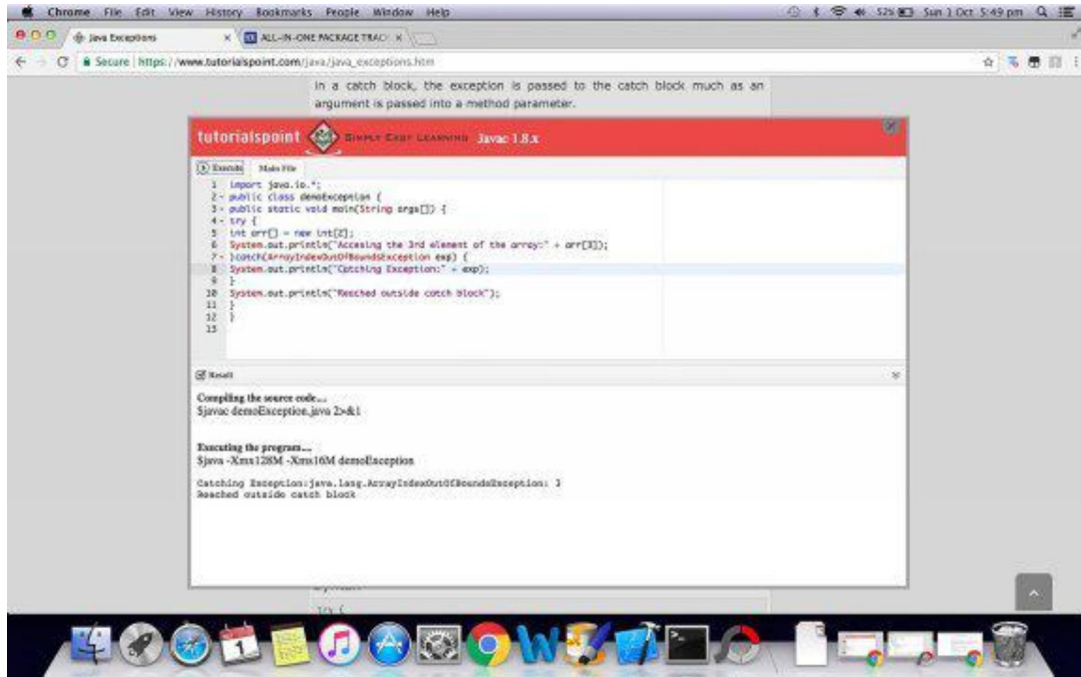
```
}
```

```
System.out.println("Reached outside catch block");
```

}

}

The output of this code is shown in the image below.



Implementing multiple catch blocks

A block of code may lead to multiple exceptions. In order to cater for this requirement, implementation of multiple catch blocks is also allowed. The syntax for such implementation is given below.

Try {

*/*Protected code*/*

```
}catch(Exptype1 exp1) {  
    /*catch block 1*/  
    }catch(Exptype2 exp2) {  
        /*catch block 2*/  
        }catch(Exptype3 exp3) {  
  
            /*catch block 3*/
```



```
}
```

The syntax shown above has illustrated the implementation of three catch blocks. however, you can implement as many catch blocks as you want. When this code is executed, the protected is executed. If an exception occurs, the type of exception is matched with the exception of the first catch block. however, if the exception type does not match, the catch block 1 is ignored and the exception type for the second catch block is tried for matching. Whichever catch block has the same exception type as that of the raised exception, the corresponding catch block is executed.

Keywords: throw/throws

These two keywords are similar words that are used for two entirely different purposes. The keyword throws is used at the end of the method name to declare that the code throws a checked exception, but the method does not handle it. In other words, this keyword is used to delay the handling of a thrown exception. On the other hand, throw keyword is used to literally throw an exception. This exception may be a new one or an exception that has previously occurred.

Sample code that illustrates the use of throws and throw is given below. In the method declaration, it has been declared that the method throws an exception rexception while the use of throw initiates the throwing of this exception.

```
Import java.io.*;
```

```
Public class cname {
```

```
    Public void depositMethod(double amountdep) throws
```

```
        rexception { /*Implementation*/
```

```
        Throw new remoteexception();
```

```
    }
```

```
    /*Implementation
```

```
}
```

It may happen that a method may throw more than one exception. In order to declare these exceptions in the method declaration statement, you can provide a comma-separated list of exceptions. A sample implementation of this concept has been given below.

```

Import java.io.*;

Public class cname {

Public void withdrawdemo(double amountdemo) throws rexception,
insufffundsexception {
        /*Implementation*/
    }

    /*Implementation*/
}

```

Finally block

NOW that you are thorough with the try and catch block, let us look at the third block used in exception handling, finally block. This is the last block that appears after the 'try and catch' blocks. Moreover, the execution of the catch block may be omitted if the exception is not found, but the finally block is always executed irrespective of whether the exception occurs or not.

The objective of this block is to perform any cleanup activities that may be deemed necessary by the programmer. This is the code that you want to execute no matter what happens to the code enclosed in the try block. The syntax of this block and how it can be implemented has been shown below.

```

Try {

    /*Protected code*/

}catch(Exptype1 exp1)
{

    /*catch block 1*/
    }catch(Exptype2 exp2) {

        /*catch block 2*/
    }catch(Exptype3 exp3) {

        /*catch block 3*/
    }
}

```

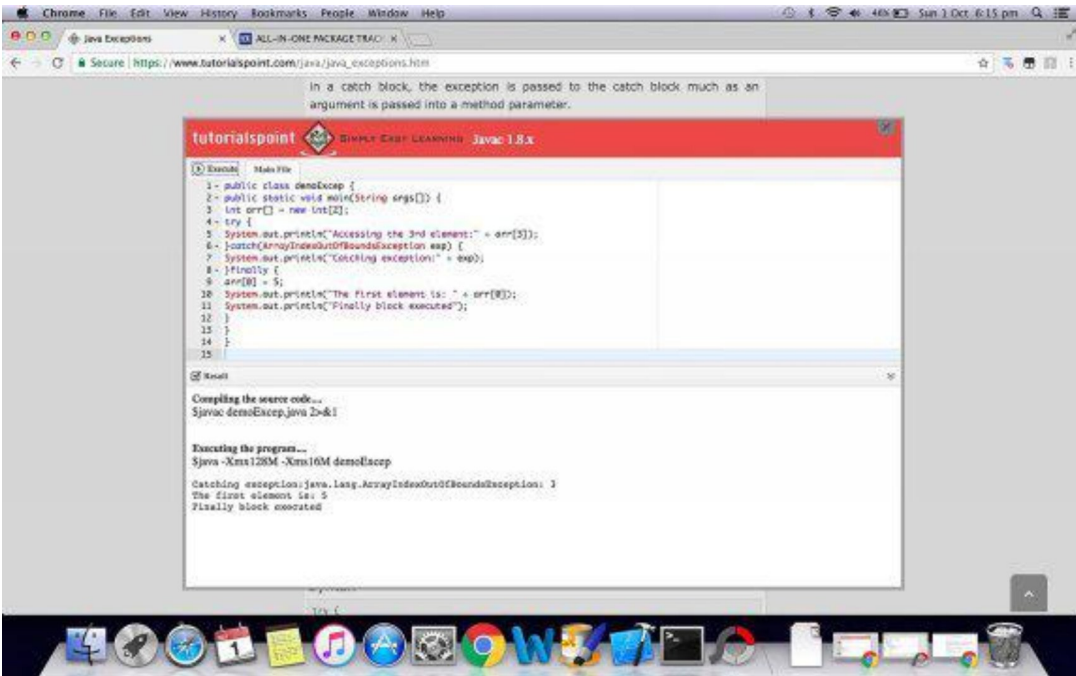
}finally {

```
        /*Finally block implementation*/
    }
}
```

Sample code that implements this concept is given below to make the concept clearer.

```
Public class demoexcep {
    Public static void main(String args[]) {
        Int arr[] = new int[2];
        Try {
            System.out.println("Accessing the 3rd element:" +
arr[3]); }catch(arrayindexoutofboundsexception exp) {
System.out.println("catching exception:" + exp);
}finally {
            Arr[0] = 5;
            System.out.println("the first element is: " +
arr[0]); System.out.println("Finally block
executed");
        }
    }
}
}
```

The output of the code is shown in the image below.



While implementing the try/catch/finally blocks, always remember a few things. Firstly, there cannot be a catch block unless you have implemented a try block. Analogously, there cannot be a standalone try block without a catch or finally block. Besides this, finally block is an optional construct and your try/catch will work just fine even if you don't have a finally block in your code.

User-defined Exceptions

Java allows you to define your own exceptions. However, before you can do that, there are a few things that you need to remember. Firstly, any exception needs to be derived or be a child of

throwable class. In order to implement a checked exception, you will have to extend the Exception class. however, the implementation of runtime exception has to extend runtimeexception class.

The standard method for defining an exception

is – *class demoexception extends Exception { }*

The following code illustrates the implementation and use of user-defined exceptions.

Import java.io.;*

Public class insuffundsexception extends Exception {

```

    Private double amo;

    Public insuffundsexception(double amo) {
        This.amo = amo;
    }

    Public double getamo () {
        Return amo;
    }
}

```

Conclusion

This book gave you a beginner's tutorial on java fundamentals. We have covered all the basic concepts of java programming and everything you need to get started with java programming. Now that you have had a run-through of all the basic concepts of java and assuming that you have tried all the codes given in this book, you are all set to jump to advanced java programming.

The key to becoming an expert at programming, be it any programming language, is practice. The more you will practice, the better your programming skills will become! We hope you enjoyed this book and that this book has helped you gain a basic level expertise in java. We hope to have laid the right foundation for you to hone your skills and become an expert java programmer.