JAVA PROGRAMMING GUIDE

Table of Contents

Introduction to Java
Chapter 1: Getting started
Chapter 2: Object Oriented Programming
Chapter 3: Variable Declaration & Data Types
Chapter 4: Control Statements
Chapter 5: Encapsulation
Chapter 6: Polymorphism and Inheritance
Chapter 7: Life and Death Of An Object
Chapter 8: Type Casting
Chapter 9: Exception Handling
Chapter 10: Introduction to JavaScript
Chapter 11: Functions and Events in JavaScript
Chapter 12: Debugging in JavaScript
Chapter 13: Objects in JavaScript
Conclusion

Introduction to Java

Java – one of the most popular programming languages of the Internet age was conceived and developed in 1991 at Sun Microsystems. The evolution of any programming language is driven by necessity that arises due to development in the computing ecosystem – and java was no exception.

Though code written in C or C++, the powerful predecessors of Java could also run on all devices, the effort required to compile them each time to make it work on specific devices was very expensive.

Java overcame this by simply taking the text source code (.java) and compiling it to an intermediate byte code (.class) file, which can then be interpreted by any device running an interpreter - Java Virtual machine (JVM). The idea of byte code meant the code became architecture neutral, needs to be compiled only once and can be interpreted on any device which can run a JVM on it.

Today, Java has grown into an important programming language and it is become almost imperative for anyone interested in programming to know at least the basics of Java. This book has been written exactly for this purpose. I want to help anyone with little or no prior programming experience get an easy and effective introduction to java programming.

Chapter 1: Getting started

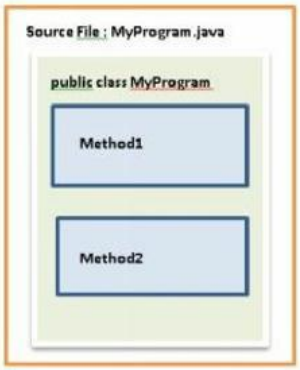
Installation

Before anything else, we need to get our development environment ready. Beginners can use online development environment (https://ideone.com/) to execute small snippets of code, but if the code to be executed is a full-fledged program, then one must install java to their system and setup their own offline development environment. Java comes in three types –

- The Micro Edition (J2ME)
- Standard Edition (J2SE)
- Enterprise Edition (J2EE)

For personal use, one can download .exe of latest SE version corresponding to their operating system from the download page. After installing, check that the PATH variable points to the java installation. Executing "java" from the terminal window will display the version of java installed on the screen.

Code Structure



Every java code is saved with the extension ". Java". Each java program contains one or more classes. Each of these classes contains one or more method definitions and each method carries a few statements in it.

Syntax and meanings of keywords

```
import java.io. *;
public class MyFirstProgram {
        public static void main(String[] args) {
                //This is my first java program
                System.out.println(" Hello World !");
        }
}
```

import java.io.*; - The keyword "import" is used for including java libraries called "packages" which contain pre-defined functions that can be used in the program.

Public class MyFirstProgram - this is the class definition, the key word "public" indicates that this class is available to all and "class" denotes to be a class definition.

MyFirstProgram – This is the name of a class. Name of the class having the main () function should be the same as the source file name.

The class definition is enclosed within the parenthesis {}

Public static void main (String args []): This is the main part of the program. Since main () is the driver function, it should always be "public".

Static: You need to keep in mind that main() method should always be "static" because it makes the program memory efficient.

Void: This is the return type of the method. This means that the main() method will not return anything.

Main: name of the method. The interpreter starts execution from the main() method only. Running a program means instructing the JVM to "load " the

.class file which was compiled from the source code and start executing from the main() method.

The two uses of the main method can be summarized as:

1. To Test all the other methods of the program
2. To Start execution of the program.

String[] args : is the argument that is passed to the "main" method. This method always takes an array of strings as its input parameter, and the array name should always be "args".

Note that the data type starts with a upper case "S", this is because "Strings" is a class in Java and all class names start with upper case.
The definition of the ***method*** is enclosed within flower braces{}, in this case, the only thing that we want our program to do is print "Hello World!"

To print a statement to standard output console***, "System.out.println("")*** ;"is used. Note that all statements must end with a semi-colon (;)
You can also use System.out.print () to display output, the difference is that print"ln" introduces a new line at the end of each statement it prints.

The other statement within the main() method in our program is called a "***comment"*** statement. These are not compiled, the purpose of these statements which are preceded by "//" are only to give the programmer more information about code.

Java doesn't have the concept of global variables at all; the same is achieved using keywords such as "final, "static", "public".

Using the Java library

When working with Java, you can see that there are a number of predefined classes present. All these classes can be used within the Java program leaving the user the task of reinventing the wheel. But for making use of these classes, you should have some minimum knowledge of what you need to use and where

to find it. These are called as the Java APIs. The library is nothing but a huge pile of predefined classes that can be used anywhere within the program. They contain predefined code and all the user needs to know is how to use it. These libraries believe the user from typing long lines of code. This is also called as the Java standard code library. Within the Java platform, the Java standard code library serves three main purposes. They are

- Just like other code library is the Java libraries facilitate the programmer with a set of useful facilities. These facilities include regular expression processing, container classes etc.
- The standard code library provides the user with abstract interfaces with which he can execute tasks independently. With this the user need not write the complete code for complex processes like file access and network access.
- There are a few underlying platforms that do not support all of the Java applications. In such cases you can use the Java library to emulate those features or you can use it to provide a way by which you can check for specific features presence.

Implementation and configuration of the Java standard code library.

Most of the code in the standard library is written in Java. The code, which is needed for the direct access of the operating system or the hardware are exceptions. The Java native interface wrappers are used in such cases for accessing the APIs of the operating system. Almost all of the Java code libraries are stored in a single archive called as the "rt.jar". This file is distributed by the JDK and the JRE distributions. The Java class library can be found in the bootstrap class path.

Main features of the Java class Library.

We use classes to access the Java class library features. These come under packages. Some of the main features of the Java class library are given below.

- java.lang has all the fundamental interfaces and classes. They are closely tied to the runtime system and the language.
- There is a special package for mathematical operations which includes

mathematical expressions, evaluations and arbitrary precision decimal numbers. They are all present in the java.math library.

- Utilities and collections: Java provides the user with its built-in Utility classes and data structures. These utility classes and data structures can be used for data compression, blogging, concurrency and regular expressions.
- 2D graphics and graphical user interface: The users can use the AWT package which provides the graphical user interface operations. This package binds to the native system. You can use this for designing platform independent toolkits and also pluggable feel and look. This package can also be used for dealing text components that are editable and non-editable.
- Sound: this package provides the user with classes and interfaces that can be used for writing, reading, synthesising and sequencing of sound data.
- Text: The java.text interface is used for dealing with text, numbers, messages and dates.
- Databases: user can access the database (sql) using the java.SQL library.
- The Java.Security provides security along with encryption services. There is encryption services are present in the javax.crypto library.
- Java also enables you to access scripting engines. You can use any conforming scripting language using the javax.script package.
- Applets: the Java.applet package allows you to download applications using the network and allows you to run them in a guarded sandbox.

Chapter 2: Object Oriented Programming

Java has its syntaxes similar to C while programming philosophy from C++ - that is, java is an object-oriented language.

In the first example that we saw, we had a class declaration and said that the main() function takes care what gets done in our program, well, that doesn't make java a OOP language, let's see an example of to understand OOP - objects and classes better.

Procedural Approach

It is not that simple, because the previous shapes were definite and animation code could have assumed a center point and rotated the shape, while in case of an indefinite shape like amoeba, it'll not rotate using the code for other shapes, instead the API needs to take couple of more arguments to define the co-ordinates on which the shape should rotate.

Object Oriented Approach

So you can write one class file each for the shapes, so adding amoeba or modifying to rotate method is done without disturbing existing code, by simple adding a new class. You need to test only the new class. Any new issue introduced can be easily tracked to the new changes made.

Example 1:

```
class  Movie{

  //variable Declaration
  //methods
}m1;   //m1 is an object of class type "Movie"
```

Example 2:

```
class Movie{
   //variable Declaration
//methods
}

class  TestMovie{
public static void main(String[] args){
Movie m1 = new Movie();  // m1 is an object of type "Movie"
m1.variable = value;
m1.method();
}
}
```

Chapter 3: Variable Declaration & Data Types

The next most important and fundamental concept to learn is about "variables" and "data types" to create useful programs. Variables are basically named memory location, which are used to hold values. Java is a strongly typed language, so the variable of type "x" can hold a value of type "x" only.

For example:

```
int n;
```

Here a variable "n" is declared to hold value of type "int". Declaring variable type ensure "type-safety" feature of java which in turn contributes to "security".

Variables in java can be either

- Local Variables (primitive data types like int, float, double, Boolean etc.)
- Reference Variables (objects – user defined instances of class)

Assume that you declared a variable of type int, you can probably use it to save a short value. If done otherwise, it'll overflow.

Also note that java thinks anything with decimal point is by default a "double", therefore it's important to append an "f" at the end of a given decimal number to make it a double.

```
float f = 1.56f;
```

Rules for declaring variable names:

- Do not use any predefined reserved words as variable names.
- Variable names cannot start with a number.

Some popular Java keywords that one should know:

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				

Table 1-1 The Java Keywords

Strings

String is declared as a class. Some of the commonly used operations on string are:

```
String str =  "Hello World";  //declaration
int len = str.length() ; // Givesthe length of the string
int len = "hello Wolrd".length(); // also gives length of the string
String concatStr = "hello"+"world"; // the + operator can be
overloaded to perform concatenation of two strings.
String str1 = "Heloo";
String str2 = "World";
if (str1 == str2)  //compare two strings
   System.out.println("strings are equal");
```

Reference Variable

Movie m1 = new Movie()

To the left is the declaration,which instructs the JVM to reserve space for the reference variable which holds the data to reach the actual object.

```
int[] n;  // declare an array of integers
n = new int[5];   //creates an array of 5 integers
```

Note that even if it's an array of primitive data type values, an array becomes an object.

Movie[] mArray; // declarean array of objects oftype "Movie"

Size: There is no predefined size for reference variables like primitive types.

Equals Operation

We know that comparison operators like == and != can be used on primitive operators, but in order to perform comparison operation on objects, one cannot use the default operators, instead java provides keywords "equals".

Array List

This is available in the java.util.ArrayList package.

```
ArrayList<String> myList = new ArrayList<String>();
```

It allows the declaration of an array with different data types. It offers multiple methods to perform common array operations like add(elem) remove(index) to add/remove elements from the arrayList. Also API's like size() to find the length of array, to find if isEmpty() or if the arrayList contains() a given element.

Let's see an example using the concepts learned so far,

```
Class MyProgram{
    public static void main(Strings[] args)
    {
        //declare and assign variables
        int num = 10;
        float fNum = 10.5;
        //print statements to output consoles
        System.out.println("The value of integer num is :
        "+num);
        System.out.println("The Value of integer fNum is :
        "+fNum);
        num = num/2;
        fNum = fNum+num;
        System.out.println("The value of integer num after
```

```
                operation is : "+num);
                System.out.println("The Value of integer fNum after
                operation is : "+fNum);
        }
}
```

Output:

```
The value of integer num is : 10
The value of integer fNum is : 10.5
The value of integer num after operation is : 5
The value of integer fNum after operation is  : 20.5
```

You just have to use + operator in the SoP (System.out.println) statement.

Wrapper Classes

As we know, Java is an object oriented programming language and everything in Java can be viewed as an object. Anything from simple files, images, an address of a system, etc. can all be considered as objects in Java. We can change the date of type of an object using the wrapper classes. In this tutorial we will look into the wrapper classes and their details. Using Wrapper classes, you can convert any given data type into an object.

All the primitive datatypes in Java are not objects. These primitive data types don't belong to any class. These are defined in the Java language itself. We may encounter situations where we will have two changes these primitive data types into objects. What example, you can add a data type to vector or a stack by converting it into an object. The Java design introduced wrapper classes for making these conversions.

About wrapper classes

As the name suggests, wrapper classes enclose (wrap) around any data type to give it an appearance of an object and it can be used whenever the data type is needed. This is a process that can be reversed. Wrapper classes can not only change the data types into classes but also can un-wrap the object and return back the data type. This is just like chocolate wrapper on the chocolate. The manufacturer wraps the chocolate in the wrapper and ships it. The use of removes the wrapper for having the chocolate.

It is have a look at the following conversion where the int datatype is converted into an object.

```
int k = 100;
Integer it1 = new Integer (k);
```

In the above example, the int data type k, using wrapper classes, is converted into an object it1. This is done by using the integer class. And whenever K is required, the object it1 can be used.

The following code shows you how to wrap the object and get the int datatype from the integer object it1.

```
int m = it1.intValue();
System.out.println(m*m); // prints 10000
```

List of Wrapper classes

In the Java language, there are 8 data types present and for each of them have their own wrapper classes.

Primitive data type	Wrapper class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
Boolean	Boolean

All the wrapper classes can be found in the java.lang package. They are placed there so that they can be made available implicitly to the programmer whenever imported. To avoid inheritance, the Java designers made these wrapper classes as final.

Importance of Wrapper classes

We use wrapper classes for serving two purposes.

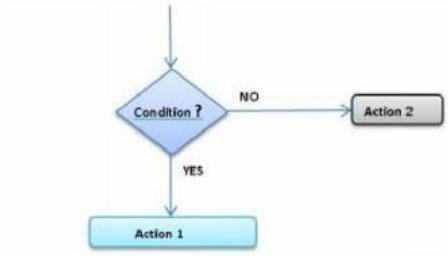
- For converting simple data types into objects. This is done by giving the object form to a given data type.
- For converting the strings into data types. For this type of conversion we use the methods like parseXXX().

Here there is an example program showing the conversion of a datatype to an object and also retrieving the data type from at the same time.

```
public class WrappingUnwrapping
{
 public static void main(String args[])
 {                                                    //  data types
  byte grade = 2;
  int marks = 50;
   float price = 8.6f;                                        // observe a suffix of
<strong>f</strong> for float
  double rate = 50.5;
                                          // data types to objects
  Byte g1 = new Byte(grade);                 // wrapping
  Integer m1 = new Integer(marks);
  Float f1 = new Float(price);
  Double r1 = new Double(rate);
                                                    // let us print the values from
objects
  System.out.println("Values of Wrapper objects (printing as objects)");
  System.out.println("Byte object g1:  " + g1);
  System.out.println("Integer object m1:  " + m1);
  System.out.println("Float object f1:  " + f1);
  System.out.println("Double object r1:  " + r1);
              // objects to data types (retrieving data types from objects)
  byte bv = g1.byteValue();              // unwrapping
  int iv = m1.intValue();
  float fv = f1.floatValue();
  double dv = r1.doubleValue();
                                                  // let us print the values from data
types
  System.out.println("Unwrapped values (printing as data types)");
  System.out.println("byte value, bv: " + bv);
  System.out.println("int value, iv: " + iv);
  System.out.println("float value, fv: " + fv);
  System.out.println("double value, dv: " + dv);
 }
}
```

Chapter 4: Control Statements

The control statements in a program, as the name itself suggests, control the general flow of every program. It includes conditional statements like if else, or iteration statements like while (), do while() etc.



If else :

Syntax :

```
if(condition)
    do action1;
else
    do action2;
```

An action is taken when the 'condition' is true, or else the action in the else part is taken.

Switch case

This operation comes in handy if you had to check for multiple conditions. Switch case takes action based on a given value.

Syntax :

```
Switch(id)
```

```
{
        Case id1:
                Action;
                Break;

        Case id2:
                Action;
                Break;
                ….

        Default:
        ….
}
```

While ()

This is an iteration statement, which performs action as long as the condition is true.

Syntax:

```
        while(condition)
{
   Do something while condition is true;
}
```

Do - while ()

This too is an iteration operation, quite similar to the while() loop, the difference being that the loop will execute at least once irrespective of the validity of the condition, because the condition is evaluated only after the "do" statement.

Syntax

```
do{
     do Something;
}while(condition);
```

For loop

Syntax: *for (initialization ;condition; counter updation);*

Example : *for (int i=0; i<5; i++)*

This will now execute the statement for exactly 5 times. The first part declares and initializes a value to the counter variable, the second part checks the condition, if true, it enters loop. On completion of loop, it performs the increment. Then it again checks the condition and continues the same steps till the condition turns false. Once the condition is false, it exits out of the loop.

Example:

```
Class MyProgram{
public static void main(Strings[] args){
        //declare and assign variables
        int  num = 5, i;
        while( num > 0){
                System.out.println("The Value of integer Num
                is : "+num);
                num--;
        }
        for(i =0;i<5;i++)
                System.out.println("The Value of integer Num
                is : "+num);
        }
}
```

Output:

```
The Value of integer Num  is : 5
The Value of integer Num  is :4
The Value of integer Num  is :3
The Value of integer Num  is :2
The Value of integer Num  is :1
```

The Value of integer Num is :5
The Value of integer Num is :5
The Value of integer Num is :5
The Value of integer Num is :5
The Value of integer Num is :5

The above stated program used two looping statements to perform the operations.

Differences between for and while loops:

As you know, the 'for' loop has three parts. They are Initialization, Boolean set and iteration expression. Here in the while loop there will be no built-in initialization or iteration expressions. It consists of only the Boolean test. While loops are good to be used when you don't know the number of iterations in which the condition will return true. The number of iterations is known, it is wise to use the 'for' loop as it is cleaner.

Here is an example where a simple while loop is used.

Example:

```
public class Test {


  public static void main(String args[]) {
    int x = 10;


    while( x < 20 ) {
      System.out.print("value of x : " + x );
      x++;
      System.out.print("\n");
    }
  }
}
```

Output:

value of x : 10

value of x : 11

value of x : 12

value of x : 13

value of x : 14

value of x : 15

value of x : 16

value of x : 17

value of x : 18

value of x : 19

More about for loops

Here we will look into the details of loops. We will look at the 'for' loop now. Each 'for' loop consists of three parts. They are

- **Initialisation**: Here in the initialisation part, we will declare and initialise the loo with a suitable variable. We use this within the body of the loop. This variable that we use in the loop will be mostly used as a counter. The number of variables that can be placed inside a loop is not just limited to 1. You can place any number of variables that can be used as counters in the body of the loop.
- **Boolean set**: The conditional test will be placed in this part of the loop. The given condition, whatever it maybe, must resolve to a Boolean value. You can give an expression or you can use methods which return Booleans.
- **Iteration expression**: in this part of the loop, you can place one or more desired actions that you want to happen every time the loop is run. Always keep in mind that the execution will only happen at the end of the loop.

Nested for loop:

You can place a 'for' loop inside another for loop. This arrangement is called

as the nested for loop. You can place any number of loops inside loops. But it is advised not to go beyond five or six levels of nested loops as it will make the program look clumsy. The odometer of the car works the same principle of the nested for loop. Another example of this is the date or calendar. As we all know, the day consists of hours, minutes and seconds. In a digital clock, the last branch of the nested for loop is the seconds counter. Above it we have the minute counter. About the minutes counter we have the hour counter. It is a nested loop in a nested loop in a nested loop. First the syntax of the nested for loops is given below. There are three forms in for loops. They are.
Form 1 syntax:

```
for(initialization; test; increment)
{
statements;
while(expression)
{
statements;
|
|
|
do
{
statements;
}while(expression);
}
}
```

Form 2 syntax:

```
for(initialization; test; increment)
{
statements;
for(initialization; test; increment)
{
statements;
|
```

```
|
|
for(initialization; test; increment)
{
statements;
}
}
}
```

Form 3 syntax:

```
While(expression)
{
statements;
|
|
|
While(expression)
{
statements;
}
}
```

Here are a few examples which will make you understand the working of the nested for loops.

Example 1 : Program to display triangle of * using nested for loop

```
class NestedForLoopDemo
{
 public static void main(String args[])
 {
for(int i = 1; i <=5 ; i++)
{
for(int j = 1; j <= i; j++)
```

```
{
System.out.println("* ");
}
System.out.println("");
}
 }
}
```

Output

```
*
* *
* * *
* * * *
* * * * *
```

Example 2 : Program to print tables

```
class NestedWhileLoop
{
 public static void main(String args[])
 {
int i=1, j=1;
System.out.println("Tables");

while(i <= 2) // change to 2 to 10 or 20 as many tables user want
{
while(j <= 10)
{
System.out.println(i + " * " + j + " = " + (i*j));
j++;
}
i++;
System.out.println("");
System.out.println("");
}
 }
```

```
}
```

Output

```
Tables
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
1 * 6 = 6
1 * 7 = 7
1 * 8 = 8
1 * 9 = 9
1 * 10 = 10

2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
2 * 6 = 12
2 * 7 = 14
2 * 8 = 16
2 * 9 = 18
2 * 10 = 20
```

Example 3 : Program to print triangle of numbers

```
class NestedDOWhileLoop
{
 public static void main(String args[])
 {
int i=1, j=1;
do
{
k = 3;
```

```
do
{
      System.out.println(" ");
k--;
}while(k>=i);
j = 1;
do
{
System.out.println(i + " ");
j++:
}while(j<=i);

System.out.println("");
i++;

      }while(i<=5);

 }
}
```

Output

```
   1
  2 2
 3 3 3
```

Chapter 5: Encapsulation

Encapsulation, as the name suggests, capsules the data. In simpler words, it hides the dada, thereby providing data security.

Arguments passed can be either by value or reference. The data type is declared after determining which one to use.

A method of the class is accessed by the object using the dot (.) operator.

```
Movie m1 = new Movie()
M1.printName(); //print the name of the movie

Or M1.play(count);  //play the movie n times
```

In the following example, a getter is used to read the value of a class variable and then set a value to the variable.

```
public class GetterSetter{
//Declaring instance variables
        private String name;
        private int age;

      //getters
       public int getAge(){
          return age;
        }

        public String getName(){
          return name;
        }

      //setters
```

```
    public void setAge( int newAge){
      age = newAge;
    }

    public void setName(String newName){
      name = newName;
    }

}
```

```
//class to test the getter and setter methods
    public class TestGetterSetter{
        public static void main(String args[]){
        TestGetterSetter gs = new TestGetterSetter();
        gs.setName("John");
        gs.setAge(30);
        System.out.println("Your name is: "+gs.getName());
        System.out.println("Your age is: "+getAge());
        }
}
```

```
        Output :
        Your name is: John
        Your age is: 3
```

Chapter 6: Polymorphism and Inheritance

In Java, polymorphism can be defined as the ability of a single object to take on multiple forms. In object oriented programming, polymorphism occurs when a child class object uses the reference of its parent class object.
Any object that has the ability to pass more than a single IS-A relation can be considered polymorphic. In other words, every object in Java is polymorphic, as it will have an IS-A relation with its own type and the class object.
Using the reference by variable is the only way to access an object in Java. And that reference variable can only be of a particular type. Once the variable is declared, the type of that reference variable stays the same and cannot be changed.
If the reference variable is not declared final, we can reassign that reference variable to other objects. The methods that different valuable can invoke on a given object depends on the type of the reference variable declared.
Any object of a given declared type or subtype can be referred using that reference variable. Here is an example showing the polymorphic behavior.

Example:

```
public interface Vegetarian{}

public class Animal{}

public class Deer extends Animal implements Vegetarian{}
```

Here, the deer class can be said to be polymorphic as it has multiple inheritance to it. The following statements are true for the example given above.

- A Deer IS-A Animal
- A Deer IS-A Vegetarian
- A Deer IS-A Deer
- A Deer IS-A Object

The following declarations will be lethal when we apply the deer object reference to the

reference variable facts.

```
Deer d = new Deer();
Animal a = d;
Vegetarian v = d;
Object o = d;
```

Here, all of those reference variables refer to the object deer in the heap memory

Virtual Methods

Here in this section we will look at the behavior of the overridden methods. In Java, you can make use of polymorphism when you are designing your class. In method overriding, a child class has the ability to override a method in its parent class. A method that is overridden is essentially hidden. This over and done my third cannot be invoked until the super keyword is used by the child class. The super keyword should be used within the method that is overridden. Consider this following example.

```
/* File name : Employee.java */
public class Employee
{
 private String name;
 private String address;
 private int number;
 public Employee(String name, String address, int number)
 {
   System.out.println("Constructing an Employee");
   this.name = name;
   this.address = address;
   this.number = number;
 }
 public void mailCheck()
 {
```

```
    System.out.println("Mailing a check to " + this.name
     + " " + this.address);
  }
  public String toString()
  {
    return name + " " + address + " " + number;
  }
  public String getName()
  {
    return name;
  }
  public String getAddress()
  {
    return address;
  }
  public void setAddress(String newAddress)
  {
    address = newAddress;
  }
  public int getNumber()
  {
   return number;
  }
}
```

If we extend the employee class as given below:

```
/* File name : Salary.java */
public class Salary extends Employee
{
```

```
private double salary; //Annual salary
public Salary(String name, String address, int number, double
  salary)
{
   super(name, address, number);
   setSalary(salary);
}
public void mailCheck()
{
   System.out.println("Within mailCheck of Salary class ");
   System.out.println("Mailing check to " + getName()
   + " with salary " + salary);
}
public double getSalary()
{
   return salary;
}
public void setSalary(double newSalary)
{
   if(newSalary >= 0.0)
   {
      salary = newSalary;
   }
}
public double computePay()
{
  System.out.println("Computing salary pay for " + getName());
  return salary/52;
```

```
  }
}
```

Now have a look at the following program and try to tell the output

```
/* File name : VirtualDemo.java */
public class VirtualDemo
{
  public static void main(String [] args)
  {
    Salary s = new Salary("George Martin", "Ambehta, UP", 3, 3600.00);
    Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);
    System.out.println("Call mailCheck using Salary reference --");
    s.mailCheck();
    System.out.println("\n Call mailCheck using Employee reference--");
    e.mailCheck();
  }
}
```

Output:

```
Constructing an Employee
Constructing an Employee
Call mailCheck using Salary reference --
Within mailCheck of Salary class
Mailing check to George Martin with salary 3600.0

Call mailCheck using Employee reference--
Within mailCheck of Salary class

Mailing check to John Adams with salary 2400.0
```

In this example, two salary objects had been instantiated. One of them is using the salary reference s while the other is Employee reference e.

The compiler looks at the mailCheck() when invoking s.mailCheck() during the compile time. The Java virtual machine then invokes the mailCheck() that is present in the salary class. This happens during the runtime.

Now because 'e' is an employee reference, invoking the mailCheck() will be different. When the e.mailCheck() is called, the compiler goes to the mailCheck(). The mailCheck() is present in the Employee class.
The compiler used to mailCheck() from the Employee for validating the statement meant during compile time. But during runtime, the Java virtual machine will go and invoke the mailCheck() from the salary class. And this particular behaviour in Java is referred to as the virtual invocation method or 'virtual method invocation '. In Java all methods behave in this manner. And overridden method will always be invoked during the runtime and does not depend on the reference of the data type used during the compile time.

Abstract Methods and Classes

An abstract class can be defined as a class that has been declared abstract. It doesn't matter if it has abstract methods in it or not. These abstract classes can be subclasses but they cannot be instantiated.

You can call him at 8 to be an abstract method if it is declared and doesn't have an implementation. An example for an abstract method is given below

```
abstract void moveTo(double deltaX, double deltaY);
```

A class should be declared abstract if it has abstract methods in it. For example.

```
public abstract class GraphicObject {
  // declare fields
  // declare nonabstract methods
  abstract void draw();
}
```

If you subclass an abstract class, usually the subclass will provide the implementations for every single abstract method that is present in its parent class. But if it doesn't, that is a class should not be declared as an abstract class.
The methods that are present in an interface which are not declared as static or default are implicitly considered as abstract. When dealing with interface methods, the abstract modifier will not be used.

Abstract Classes Compared to Interfaces
Abstract classes and interfaces are similar to each other considering the following similarities.

- One cannot instantiate the abstract classes or interfaces.
- Both of them cannot have a mix of methods that are declared with or without implementation.

Abstract classes I love you to declare Final and static, lets you define public, private and protected concrete methods. But interfaces all fields will automatically be defined as public, final and a static. And all the methods you declare define will be public by default. And you cannot extend them beyond one class but can implement multiple interfaces to it.

Which one is a better option?
You can consider going with abstract classes if any one of the following statements relate to your situation.

- You are planning to share the code with other closely related classes.
- If you think that the classes with which you extend your abstract class have a common fields or methods.
- If you require access modifiers in private or protected.
- If you wish to declare non final or non-static fields.

You can use interfaces if any of the following statements apply to your required situation.

- If you think that your interface will be implemented by unrelated classes. Eg. Comparable and cloneable.

- If you are planning to make use of the multiple inheritance.
- If you want to specify particular datatypes behaviour without the concern about who will implement its behaviour.

Inheritance in JAVA

Inheritance in Java can be defined as A process in which an object can acquire the properties of a different object. With inheritance, the properties of the class can be shared with other classes to which the inheritance directs. Information can be made manageable with inheritance. The objects follow the hierarchical order when inheritance is used on them. There are two types with which inheritance can be performed in Java. They are extends and implements keywords. These keywords to determine if an object is an 'IS-A' type to another. Using the above keywords, we can make the object acquire another object's properties.

IS-A Relationship:

IS-A relation means: when we define an object is a type of another object they are said to have IS-A relationship between them. Here we will see how to use the extent keyword for achieving inheritance.

```
public class Animal{

}


public class Mammal extends Animal{

}


public class Reptile extends Animal{

}


public class Dog extends Mammal{

}
```

Basing on the above given example it can be deduced that

- Animal class is the superclass of the mammal class
- Animal class is superclass of the reptile class

- The mammal and the reptile classes are both subclasses of the animal class
- The dog class is subclass of the mammal class which is in return a subclass of the animal class.

We can view the same using the IS-A relationship. It looks as follows

- Mammal IS-A Animal
- Reptile IS-A Animal
- Dog IS-A Mammal
- Hence : Dog IS-A Animal as well

Using the 'extends' keyword allows the subclass to inherit every single property of this superclass excluding its private properties.
Using the instance operator we can ensure that the normal class is actually an instance of the animal class.

Example:

```
public class Dog extends Mammal{


  public static void main(String args[]){


    Animal a = new Animal();
    Mammal m = new Mammal();
    Dog d = new Dog();


    System.out.println(m instanceof Animal);
    System.out.println(d instanceof Mammal);
    System.out.println(d instanceof Animal);
  }
}
```

Output: This would produce the following result

```
true
true
true
```

The instanceof operator

In Java, if you want to determine A class if it belongs to A specific given class or for implementing a given interface, you can make use of the instanceof operator. The instanceof operator is a type comparison operator. The instanceof operator returns true or false depending on the state of the class. The instanceof operator returns true if a given object belongs to a certain class or if the given object implements an interface. If not it will simply return false.
Here, in the following example you can see the use of the instanceof operator.

```
class Vehicle {

        String name;
        Vehicle() {
                name = "Vehicle";
        }
}

class dirtbike extends Vehicle {

        dirtbike() {
                name = "dirtbike";
        }
}

class atv extends dirtbike {

        atv() {
                name = "atv";
```

```
	}
}

class LightVehicle extends Vehicle {

	LightVehicle() {
		name = "LightVehicle";
	}
}

public class InstanceOfExample {

	static boolean result;
	static dirtbike hV = new dirtbike();
	static atv T = new atv();
	static dirtbike hv2 = null;
	public static void main(String[] args) {
		result = hV instanceof dirtbike;
		System.out.print("hV is an dirtbike: " + result + "\n");
		result = T instanceof dirtbike;
		System.out.print("T is an dirtbike: " + result + "\n");
		result = hV instanceof atv;
		System.out.print("hV is a atv: " + result + "\n");
		result = hv2 instanceof dirtbike;
		System.out.print("hv2 is an dirtbike: " + result + "\n");
		hV = T; //Successful Cast form child to parent
		T = (atv) hV; //Successful Explicit Cast form parent to child
	}
```

```
}
```

Output
hV is an dirtbike: true
T is an dirtbike: true
hV is a atv: false
hv2 is an dirtbike: false
Note: hv2 does not yet reference a dirtbike object, instanceof returns false. You should keep in mind that you cannot use the instanceof operator on siblings.

Implements keyword

Now that you have a good understanding about the keyword 'extends', we will look at the implements keyword and how it is used for obtaining and IS-A relationship. We use the implements keyword for classes to inherit from interfaces. You should keep in mind that the interfaces cannot be extended by classes, there can only be implemented.
Example:

```
public interface Animal {}


public class Mammal implements Animal{

}


public class Dog extends Mammal{

}
```

The instanceof Keyword:

Here we will have a look at the instanceof operator. We use this operator for determining because of class is an instance of its superclass.

```
interface Animal{}
```

```
class Mammal implements Animal{}

public class Dog extends Mammal{
  public static void main(String args[]){

    Mammal m = new Mammal();
    Dog d = new Dog();

    System.out.println(m instanceof Animal);
    System.out.println(d instanceof Mammal);
    System.out.println(d instanceof Animal);
  }
}
```

Output:

true

true

true

HAS-A relationship:

Usage is the main best for these relationships. We say that two objects have has a relationship between them if a certain class HAS-A certain thing. The main use of this kind of relationship is that it helps in reducing the bugs and duplication of code.

Here is the example showing objects that are in a HAS-A relationship between them.

Example:

```
public class Vehicle{}
public class Speed{}
public class Van extends Vehicle{
        private Speed sp;
```

}

The above example tells us that the van HAS-A speed. If we make a separate class especially for speed, we can use it again and again in different applications.

In Java, the class can only have a single inheritance. Java does not support multiple inheritances. Every class is limited to have a single inheritance using the extends keyword. However, the class is allowed to implement more than one interfaces. With this multiple inheritance can be obtained in Java. The following statement has two classes with the 'extends' keyword and it is not possible to use the keyword in that way.

Public class extends Animal, Mammal {}

Chapter 7: Life and Death Of An Object

Constructor

In Java, the constructor can be defined as a special method with which you can initialise the object. Constructors invoked during object creation. It is called a constructive because it constructs the values for an object.

Rules for creating constructors in Java
There are two basic rules that are defined for constructors in Java

- The name of the constructor must be exactly the same name of its class
- They should have no explicit return values given for a constructor.

Types of constructors
In Java, we have two types of constructors.

- The default constructor or the no-arg constructor
- Parameterized constructor.

Here we will look at them in detail.
Java default constructor:
Java default constructor is the constructor which has no parameters to it.
Syntax of default constructor:
<class_name>(){}
Here is an example showing a simple default constructor.

Example:

```
class Bike1{
Bike1(){System.out.println("Bike is created");}
public static void main(String args[]){
Bike1 b=new Bike1();
}
}
```

Output:
Bike is created

Here is an example of a default constructor, which gives default values.
Individual example you are not creating any constructors. Here, the compiler provides your code with a default constructor as none are defined and the values 0 and null are given by the default constructor.
Example:

```
class Student3{
int id;
String name;

void display(){System.out.println(id+" "+name);}

public static void main(String args[]){
Student3 s1=new Student3();
Student3 s2=new Student3();
s1.display();
s2.display();
}
}
```

Output:

```
null
null
```

Parameterized constructors in Java

In Java constructor is said to be a parameterized constructor if it has parameters with it.
Why should we use parameterized constructors?
We use parameterized constructors in cases when we give different values to different objects. Here is an example of a parameterized constructor. In the following example to parameters are given to the student class. A parameterized constructor can have any number of parameters defined in it.
Example:

```
class Student4{
  int id;
  String name;
```

```
  Student4(int i,String n){
  id = i;
  name = n;
  }
  void display(){System.out.println(id+" "+name);}

  public static void main(String args[]){
  Student4 s1 = new Student4(111,"David");
  Student4 s2 = new Student4(222,"Aryan");
  s1.display();
  s2.display();
 }
}
```

Output:

111 David

222 Aryan

Constructor overloading in Java

Constructor overloading is a technique used in Java through which any number of contractors, which have different parameters can be given to a class. They can have the same number of parameters but not the same parameters. The compiler compares the parameters taking the list and their type into account. Here is an example of constructor overloading.

Example:

```
class Student5{
  int id;
  String name;
  int age;
  Student5(int i,String n){
  id = i;
  name = n;
  }
  Student5(int i,String n,int a){
  id = i;
  name = n;
```

```
  age=a;
  }
  void display(){System.out.println(id+" "+name+" "+age);}

  public static void main(String args[]){
  Student5 s1 = new Student5(111,"David");
  Student5 s2 = new Student5(222,"Aryan",25);
  s1.display();
  s2.display();
 }
}
```

Output:

111 David 0

222 Aryan 25

Copy constructor in Java

Though we call it a copy constructor, there is no such thing in Java. If you wish to copy the values of a particular object to different objects you can use the following ways.

- Using constructors
- Assigning the values of an object into a different object
- You can use the clone() method. It belongs to the object class.

Here is an example in which we will copy an object's values into another constructor.

Example:

```
class Student6{
  int id;
  String name;
  Student6(int i,String n){
  id = i;
  name = n;
  }

  Student6(Student6 s){
```

```
  id = s.id;
  name =s.name;
  }
  void display(){System.out.println(id+" "+name);}

  public static void main(String args[]){
  Student6 s1 = new Student6(111,"David");
  Student6 s2 = new Student6(s1);
  s1.display();
  s2.display();
 }
}
```

Output:

```
111 David
111 David
```

Copying values without using constructors

As we have discussed earlier, we can copy values of an object into a different objects without using constructors. In such cases you don't need to create a constructor. You can simply copy by assigning the values of that object to another object. Here, the following example we are not using constructors but assigning the values for copying.

Example:

```
class Student7{
  int id;
  String name;
  Student7(int i,String n){
  id = i;
  name = n;
  }
  Student7(){}
  void display(){System.out.println(id+" "+name);}

  public static void main(String args[]){
  Student7 s1 = new Student7(111,"David");
  Student7 s2 = new Student7();
```

```
  s2.id=s1.id;
  s2.name=s1.name;
  s1.display();
  s2.display();
 }
}
```

Output:

111 David

111 David

Garbage collection in Java

Garbage, in Java can be defined as the unreferenced objects in Java. And the process of clearing garbage for reclaiming the unused runtime memory is called garbage collection. In simple words garbage collection means to destroy all the objects that are on used. Unlike C and C++, garbage collection is an automatic process in Java. This is one of the reasons why Java provides the user with a better and efficient memory management.

Advantages of garbage collection:

- Garbage collection makes the memory efficient by removing all the objects that are not referred. These objects use of the heap memory and garbage collection clears this memory.
- There is no need for the user to perform garbage collection manually as the process of garbage collection is automatic in Java. So we don't have to make extra efforts for managing memory. The Java virtual machine takes care of clearing this memory.

There are three ways in which you can de-allocate an object. They are

- Nullifying the reference: An object can be removed from the heap memory by removing the object's reference. Here is an example.

```
Employee e=new Employee();
e=null;
```

- By assigning the object's reference to another object: removing the reference of an object and giving it to another object makes the first object without reference. And as we know, objects without reference will be de-allocated. Here is an example given below. In the following

example, the references of the object e1 are given to e2. And that makes e1 qualified for garbage collection.

```
Employee e1=new Employee();
Employee e2=new Employee();
e1=e2;
```

Making use of anonymous object: This is the third way of de-allocating an object. Here we will create a new object. Example: new Employee();

Chapter 8: Type Casting

Type casting by object reference

In Java, you can type the cost of an object reference into a different object reference. This is called as type casting. Typecasting is not limited to the class itself. Enjoy how you can type test one of your superclass or subclass types or interfaces. For type casting there are both compile time and runtime rules. When typecasting for a given type object, reference of any object can be assigned. This is because the object class that we use is the superclass of every class in Java. There are two types of typecasting. They are

- Upcasting
- Down casting

Typecasting can be done implicitly or explicitly. It will be an implicit type casting if the casting is done from a lower data structure to a higher data structure. . Java does not perform down casting on its own. Down testing should only be done explicitly by the user. Type Casting from a higher data structure to know data structure should be done carefully as it involves lots of data if it is made forcibly. The Java virtual machine performs upcasting when needed. For example if there are a few data types of which only one of them is a higher data type, the JVM will cast all the other seven lower data types to the higher data type. Performing down casting on the case as above will result in the loss of data in the higher data type object. During the run time, there may arise a situation called as the ClassCastException, where the object on which type casting is being cast is not compatible with the other type with which it is being cast to. Here is an example explaining the ClassCastException.

Example:

```
//X is a supper class of Y and Z which are siblings.

public class RunTimeCastDemo {


        public static void main(String args[]) {
                X x = new X();
```

```
        Y y = new Y();
        Z z = new Z();
        X xy = new Y(); // compiles ok (up the hierarchy)
        X xz = new Z(); // compiles ok (up the hierarchy)
        //          Y yz = new Z();  incompatible type (siblings)
        //          Y y1 = new X();  X is not a Y
        //          Z z1 = new X();  X is not a Z
        X x1 = y; // compiles ok (y is subclass of X)
        X x2 = z; // compiles ok (z is subclass of X)
        Y y1 = (Y) x; // compiles ok but produces runtime error
        Z z1 = (Z) x; // compiles ok but produces runtime error
        Y y2 = (Y) x1; // compiles and runs ok (x1 is type Y)
        Z z2 = (Z) x2; // compiles and runs ok (x2 is type Z)
        //          Y y3 = (Y) z;   inconvertible types (siblings)
        //          Z z3 = (Z) y;   inconvertible types (siblings)
        Object o = z;
        Object o1 = (Y) o; // compiles ok but produces runtime error
    }
}
```

Casting Object References: Implicit Casting using a Compiler

Whenever an object of reference is assigned, an implicit cast will be performed. Here the reference of the object will be assigned to

- A reference variable which is of the same type as the class. This is the class from which the given object was instantiated. The superclass of every class is an object as object.
- A reference available with the data type is an interface implemented by the same class from which the object was instantiated.

Here is an example showing the interface automobile, a superclass bike and a subclass Ducati. Here you can see the automatic conversion of the given object

references that are handled by the compiler.

```
interface Automobile{
}
class Bike implements Automobile{
}class Ducati extends Bike{
}
```

Example: Let c be a variable of type bike class and f be of class Ducati and v be an vehicle interface reference. We can assign the Ducati reference to the bike variable:

I.e. we can do the following

Example 1

```
c = f; //Ok Compiles fine
```

Where c = new bike();

And, f = new Ducati();

The compiler automatically handles the conversion (assignment) since the types are compatible (subclass – super class relationship), i.e., the type bike can hold the type Ducati since a Ducati is a bike.

Example_2

```
v = c; //Ok Compiles fine
c = v; // illegal conversion from interface type to class type results in compilation error
```

Where c = new bike();

And v is a Vehicle interface reference (Vehicle v)

The compiler automatically handles the conversion (assignment) since the types are compatible (class – interface relationship), i.e., the type bike can be cast to Vehicle interface type since bike implements Vehicle Interface. (bike is a Vehicle).

Casting Object References: Explicit Casting

Sometimes we do have to do type casting explicitly in Java when the implicit typecasting doesn't work when they are not efficient. This kind of casting is simple and it only has the name of the new "type" inside matched pair of parenthesis. Here we will look at the same example with bike and Ducati class.

```
class bike {
```

```
void bikeMethod(){
}
}class Ducati extends bike {
void DucatiMethod () {
}
}
```

Here, you can see that we also have a brakingsystem() function. This function takes the reference bike as its input parameter. Regardless the object type, the method will invoke called the bikemethod(). This also invokes the Ducatimethod(). If you wish to determine the type of the object during runtime, you can make use of the instanceof operator.

```
public void breakingSystem (bike obj) {
  obj.bikeMethod();
 if (obj instanceof Ducati)((Ducati)obj).DucatiMethod ();
}
```

The operation tells the compiler to treat the object bike, which is a reference by object, like it is a Ducati object. Here, without casting, the company gives an error message saying that the Ducatimethod() cannot be found in the definition of bike.

The following example program will show you the use of cast operator with references.

In this class Honda and Ducati at siblings. Both of these classes are some classes of the bike class. Both of them have dirtvehicle class, which is an extended object class. Any class that cannot explicitly extend another class will be extended automatically by the object by default. The following code instantiates a given object of a particular class, in this case the Ducati class and will assign the reference of the object to the variable of the bike type. This following assignment is allowed because the class bike is the superclass of Ducati. The method should be defined in order to make use of a reference of the class type to invoke a method. And the matter should be defined at or above the class hierarchy of that given class.

Example program:

```
class bike extends Object {


        void bikeMethod() {
```

```
	}
}

class dirtbike extends Object {
}

class Ducati extends bike {

	void DucatiMethod() {
		System.out.println("I am DucatiMethod defined in Class Ducati");
	}
}

class Honda extends bike {

	void DucatiMethod() {
		System.out.println("I am DucatiMethod defined in Class Ducati");
	}
}

public class ObjectCastingEx {

	public static void main(String[] args) {
		bike obj = new Ducati();
		//	Following will result in compilation error
			//	obj.DucatiMethod();		//As the method DucatiMethod is
undefined for the bike Type
		//	Following will result in compilation error
```

```
                // ((dirtbike)obj).DucatiMethod();
                                        //DucatiMethod is undefined in the dirtbike Type
                //  Following will result in compilation error
                ((Ducati) obj).DucatiMethod();
                //Following will compile and run
                  //            Honda hondaObj = (Ducati)obj;          Cannot convert as they
are sibblings
        }
}
```

The user is allowed to cast an object reference into a string in Java. This is one of the common casting that is done when collections are being dealt with.
Example:

```
import java.util.Vector;


public class StringCastDemo {


        public static void main(String args[]) {
                String username = "asdf";
                String password = "qwer";
                Vector v = new Vector();
                v.add(username);
                v.add(password);
                //            String u = v.elementAt(0); Cannot convert from object to String
                Object u = v.elementAt(0); //Cast not done
                System.out.println("Username : " + u);
                String uname = (String) v.elementAt(0); // cast allowed
                String pass = (String) v.elementAt(1); // cast allowed
                System.out.println();
```

```
                System.out.println("Username : " + uname);
                System.out.println("Password : " + pass);
        }
}
```

Output

Username : asdf

Username : asdf

Password : qwer

Chapter 9: Exception Handling

Exception

An exception can be defined as anything that interrupts the flow of a given program. Exceptions want to let the program continue with its execution and will make them terminate without letting them continue any further. In cases like that, the system will generate error messages to notify the user about the problem occurred. But there is nothing to worry about exceptions as they can be handled.

When can an exception occur?

Exceptions can occur at both runtime and compile time. Not all exceptions occur at runtime. When there is a problem during the compilation of the program, the Java virtual machine gives an error message regarding that exception. Most of the exceptions occur during runtime. The exception is that occur during compile time are called as a compile time exceptions and those which occur during run time are called as run-time exceptions.

What are the reasons for exceptions?

There are many reasons for an exception to occur. For example, when the system cannot open the file, it throws an exception. Network connection problems, class file missing scenarios, handling operands which are out of range are some of the common reasons which throw exceptions.

There are two types of exceptions that can be found in the Java language. They are checked exceptions and unchecked exceptions.

Here we will learn about checked exceptions and unchecked exceptions in detail.

Checked exceptions

The exceptions that are checked during the compile time are called as the chat exceptions. In this case, if a method throws a checked exception, it should be handled either by using the try-catch block or they should be declared using the throws keyword. If neither of those given methods is followed, the compiler

will give a compilation error. As these exceptions checked during the compile time, they are called as checked exceptions. Here is an example that will help you understand about the checked exceptions.
In this example there is a file myfile.txt. Data will be read from that file and will be displayed on screen. There are three checked exceptions in this program and they are mentioned below.
Example:

```
import java.io.*;
class Example {
 public static void main(String args[])
 {
        FileInputStream fis = null;
        /*This constructor FileInputStream(File filename)
        * throws FileNotFoundException which is a checked
        * exception*/
    fis = new FileInputStream("B:/myfile.txt");
        int k;

        /*Method read() of FileInputStream class also throws
        * a checked exception: IOException*/
        while(( k = fis.read() ) != -1)
        {
                System.out.print((char)k);
        }
        /*The method close() closes the file input stream
        * It throws IOException*/
        fis.close();
 }
}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problems:
Unhandled exception type FileNotFoundException
Unhandled exception type IOException
```
Unhandled exception type IOException

In the above example, the check and exceptions got checked by the compiler during the compile time and as we did not declare those exceptions, the program displayed a compilation error message.

A few more check the exceptions are given below.

- SQLException
- IOException
- DataAccessException
- ClassNotFoundException
- InvocationTargetException

Unchecked exceptions

Unchecked exceptions are nothing but exceptions that are not checked during compile time. These exceptions do not give compilation errors as they cannot be checked during compile time. It doesn't matter if you have declared an exception or not. Uncheck the exceptions mostly occur due to the data that the user provides during his interaction with the program. The only way to handle these exceptions is to anticipate them and writing an appropriate handler for that exception. The RuntimeException class is the superclass of all unchecked exceptions. Here is an example which will help you to get a better understanding on unchecked exceptions.

```
class Example {
 public static void main(String args[])
 {
        int num1=10;
        int num2=0;
        /*Since I'm dividing an integer with 0
        * it should throw ArithmeticException*/
        int res=num1/num2;
        System.out.println(res);
 }
```

```
}
```

If you compile the above code you can see that the compilation will be successful but however if you run it, the program would throw an exception called ArithmeticException. With this, you know that check exceptions do not throw errors during compile time but at runtime. Here is another example.

Example:

```
class Example {
 public static void main(String args[])
 {
        int arr[] ={1,2,3,4,5};
        /*My array has only 5 elements but
        * I'm trying to display the value of
        * 8th element. It should throw
        * ArrayIndexOutOfBoundsException*/
        System.out.println(arr[7]);
 }
}
```

In this example, just like the previous one, there will be no compile time errors. However this code when run, throws an unchecked exception called the ArrayIndexOutOfBoundsException. These exceptions cannot be found during compile time and it does not mean that these are being unnoticed by the compiler. In the above example, it is wise to add a message advising the user to give a value within the range.

The code after adding the message is given below.

```
class Example {
        public static void main(String args[])
        {
                try{
                int arr[] ={1,2,3,4,5};
                System.out.println(arr[7]);
                }catch(ArrayIndexOutOfBoundsException e){
```

```
                System.out.println("The specified index does not exist " +
                                "in array. Please correct the error.");
            }
        }
}
```

Here are some of the unchecked exceptions that are frequently seen during runtime.

- NullPointerException
- ArrayIndexOutOfBoundsException
- ArithmeticException
- IllegalArgumentException

Difference between an exception and an error:

Exceptions: exceptions occur within the code and they can be handled by the developer. Developers can take the required actions when they occur. Few examples of exceptions are given below

- DivideByZeroException
- NullPointerException
- ArithmeticException
- ArrayIndexOutOfBoundsException

Errors: Errors are different from exceptions. They are abnormal conditions which the system tries not to handle. When an error occurs it indicates that the system encountered a serious problem. They define problems that cannot be got under usual conditions by the program. A few examples of errors are

- Hardware error
- Memory error
- JVM error

Exception Handling

Exception handling is a powerful mechanics in with which runtime errors can be handled. Exception handling ensures that the normal flow of the program is maintained.

A few advantages of exception handling

- Exceptional funding will allow the user to control the flow of execution of a program if it has exception handling written in it.
- Whenever there is an error, it will throw an exception. There are calling method will take care of the encounter error.
- With exception handling, differentiating and organising different types of errors when separate blocks of code are used. The try-catch blocks can be used for it.

Here we will look at a few exception handling examples.

Example:

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


 int a, b, result;


 Scanner input = new Scanner(System.in);
 System.out.println("Input two integers");


 a = input.nextInt();
 b = input.nextInt();


 // try block


 try {
  result  = a / b;
  System.out.println("Result = " + result);
 }


 // catch block


 catch (ArithmeticException e) {
```

```
  System.out.println("Exception caught: Division by zero.");
 }
 }
}
```

When an exception is caught, the catch block corresponding to it will be executed. From the example problem given above, the code will throw an ArithmeticException. There is a simple way with which exceptions can be captured. It is to use the object of the exception class as an inherited class of another class. Here is an example showing that.

Example:

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


 String languages[] = { "C", "C++", "Java", "Perl", "Python" };


 try {
  for (int c = 1; c <= 5; c++) {
    System.out.println(languages[c]);
  }
 }
 catch (Exception e) {
  System.out.println(e);
 }
 }
}
```

Output:

C++

Java

Perl

Python

java.lang. ArrayIndexOutOfBoundsException: 5

Exception handling is a very important mechanism in java and a good java developer anticipates the exceptions beforehand and writes handlers for those exceptions that might show up.

Chapter 10: Introduction to JavaScript

JavaScript is an object based dynamic scripting language. Netscape is the developer of this language. This language is used for the development of server applications and web pages. It is the most widely used scripting language in the world.

The only thing similar in Java and JavaScript is their similar sounding names. Rest, the languages are completely different with regard to their functions and concepts. What makes JavaScript stand out from the object oriented programming languages like Java and C++ is that its objects can be created or modified during the run time. For doing this, Methods and properties are added to the empty objects. An object once created can be used like a prototype in the process of creating other similar objects. Variable parameter lists, source-code recovery, dynamic script creation, dynamic object creation function variables etc are the dynamic capabilities that JavaScript has.

Basics of Java Script

Storing data in JavaScript

- Local Storage: The local storage is used to permanently store the data. With this, the data on that page can be accessed anytime. The local storage doesn't have an expiration date. One can use local storage if they want their data to be reused again and again over the period of time.
- Session Storage: The session storage is not like local storage as it only holds the data for a single session and disposes off the data once the user closes the page. This kind of storage is used on web pages used for booking tickets etc.

Storing Data using Variable Data and Constant Data

The Variables and Constants store the data and values used in JavaScript. They both have their own importance in JavaScript. All the data stored cannot be made constant. Processes like time countdown use the variables and the Constants are used for non-changing data.
Variable Data: As the name suggests, this type of data that changes its value over time is called the Variable Data. This type of data is widely used in JavaScript. Variables are dynamic in nature.

Constant Data: The Constant data is something that never changes its value. This data will be constant through time. This type of data is static. We use this type of data for data which is which cannot be modified.

Creating Variables and Constants
Declaring a Variable

A variable is declared by preceding the name by the *var* keyword. For example if you want to declare 'group' a variable, just add the *var* keyword before 'group'. Every variable should end with a ' ; ' (semicolon). This makes the statement a self-containing. Care should be taken while using the words because JavaScript is case sensitive. For this the letters are to be used in the exact same declaration as in the original.

Eg: var Dell;

The object Dell above is declared as a variable.

Declaring a Constant

The keyword *'const'* is used before the name is to make it a constant. This declaration is just like declaring a constant.

Eg: const Book;

The Book is made a constant in the above example.

Decision making in JavaScript

There always rises a situation where we have to select one of the two paths that are before him. Then we are able to implement the conditional statements like if and else.

The 'if' and else statements can be used for decision-making. This will allow the program to choose the correct path.

The following conditional statements are supported by JavaScript.

1. if statement
2. if...else statement
3. if...else if... statement

if statement: The if is the fundamental statement used to make decisions. There a condition is added to the if statement and a set of statements can be given if the given condition holds true.

SYNTAX: if(condition){

statements to be executed}

EXAMPLE:

```
<script type="text/javascript">
<!--
var age = 20;
if( age > 18 ){
  document.write("<b>Allowed inside the pub.</b>");
}
//-->
</script>
```

OUTPUT: Allowed inside the pub. (if a no. above 18 is given)

***if else* statement**: This is like an extension of the if statement. The system runs the code and executes the statements in the else block if the "if" statement holds false.

SYNTAX:

```
if(condition)

 {statements}

else{statements}
```

EXAMPLE:

```
<script type="text/javascript">
<!--
var age = 15;
if( age > 18 ){
  document.write("<b>Allowed inside the pub</b>");
}else{
  document.write("<b>Is not allowed.</b>");
}
//-->
</script>
```

OUTPUT:

Allowed inside the pub (if a number above 18 is given)

Not allowed (if a number below 18 is given)

***if else if*:** This is like adding another step to the if else statement. this is used when a series of conditions are used.

SYNTAX: if(condition){statements};

else if(condition){statements}

else if(condition){statements}

else{statements}

EXAMPLE:

```
<script type="text/javascript">
<!--
var book = "JAVASCRIPT";
if( book == "JAVA" ){
  document.write("<b>JAVA Book</b>");
}else if( book == "JAVASCRIPT" ){
  document.write("<b>JAVASCRIPT Book</b>");
}else if( book == "DBMS" ){
  document.write("<b>DBMS Book</b>");
}else{
 document.write("<b>UNKNOWN Book</b>");
}
//-->
</script>
```

OUTPUT: JAVASCRIPT book

The *nested if*

The 'nested if' is an 'if' statement inside another 'if' statement.

SYNTAX: if(condition) {if(condition){statement} }

The switch and break statements:

In the switch statement there will be an expression followed by several different statements. The statements will be executed basing on the expression's value selected.

SYNTAX:switch (expression)

```
{
 case condition 1
: statement(s)
                break;
 case condition 2: statement(s)
                break;
.
.
,
 case condition n: statement(s)
                break;
 default:
statement(s)
}
```

The break statement is used to indicate that it is the end of the preceding switch case.

Looping in JavaScript

The *for* Loop:

The loops are used if one wants the same code to be repeated again and again. The 'for' loop is one of the three loops that can be made use of. The 'for loop' is simple and compact.
It contains the three following points.

1. The start statement is performed before the loop begins.
2. The loop is executed only when the conditions given are true. But a test code will be executed which runs the code for once.
3. The counter for iterations can be increased or decreased.

SYNTAX:

```
for (initialization; test condition; iteration statement){
   Statement(s) to be executed if test condition is true

}
```

EXAMPLE:

```
<script type="text/javascript">
<!--
var count;
document.write("This will repeat for 10 times" + "<br />");
for(count = 0; count < 10; count++){
 document.write("Currently repeated : " + count );
 document.write("<br />");
}
document.write("Done!!");
//-->
```

```
</script>
```

OUTPUT:

```
This will repeat 10 times
Currently repeated : 1
Currently repeated : 2
Currently repeated : 3
Currently repeated : 4
Currently repeated : 5
Currently repeated : 6
Currently repeated : 7
Currently repeated : 8

Currently repeated : 9

Currently repeated : 10
```

Chapter 11: Functions and Events in JavaScript

What is a Function?

A group of codes which, are reusable when desired are called anywhere in the program is called a function. It reduces the time and energy for one to write the same code again and again. This reduces code redundancy. Modular code can be written by using the functions. A large program can be divided into many or desired number of parts using the functions. JavaScript, like any other programming language, supports the features that are needed for writing modular code. This is done by using 'functions'.

Functions like write() and alert() are the default functions. JavaScript provides the user with many functions which can be used right away.

JavaScript also allows one to create our own custom functions.

Function Definition:

A function should first be defined before using it. The function key is a common method to define functions in JavaScript. It should be followed by the function name that is unique and lists of parameters. It also should contain a statement block with curly braces.

SYNTAX: <script type="text/javascript">

```
<!--


function functionname(parameter-list)
{
 statements



}
```

```
//-->
</script>
```

EXAMPLE: In the example given below, there is a simple functions program.

```
<script type="text/javascript">
<!--
function simplefunction()
{
  alert("This is a simple function");
}
//-->
</script>
```

Calling a Function:

To call a function in javascript, the name of that function should be written as follows.

```
<script type="text/javascript">
<!--
simplefunction();
//-->
</script>
```

Parameters of a function:

In JavaScript a function can be passed with parameters too. Different parameters can be passed in JavaScript while it is calling a function. Manipulation can be done by the passed parameters, which are in a function. Comma is used to separate multiple parameters in a function.

Example:

Here in the following example, we will modify our *simplefunction* function. Now we will make use of two parameters.

```
<script type="text/javascript">
<!--
function simplefunction(name, age)
{
  alert( name + " is " + age + " years old.");
}
//-->
</script>
```

The function can be called. And it can be called as given below.

```
<script type="text/javascript">
<!--
simplefunction('Zara', 7 );
//-->
</script>
```

The *return* Statement:

An optional return statement can be added to a JavaScript function if the user desires a return value from that function. A return statement should always be

the last statement of the function.
For example, the user may pass two different numbers in a particular function and from them he can expect the function to return the product of those two numbers in the program.

EXAMPLE:

```
<script type="text/javascript">
<!--



function concatenate(first, last)
{
  var full
;

  full = first + last;
  return  full;
}
//-->
</script>
```

Now the function can be called as follows:

```
<script type="text/javascript">
<!--
  var result;



result = concatenate('Zara', 'Ali'


);
```

```
  alert(result );
//-->
</script>
```

Events in JavaScript

EVENT:

Events handle the interactions between JavaScript and HTML. This is done when a page is manipulated by the user or a browser.

Everything is an event. The loading of a page is an event. When the user selects something by clicking, that click is considered an event too. Infact any action done by the user like closing a window, selecting a link, pressing a key, closing a tab, resizing a window, etc. are all examples of events.

The responses that are coded in JavaScript are executed using these events like closing windows, text to be displayed, alerts displayed, confirmation messages or any type of response that is possible to happen.

The events are part of the Document Object Model (DOM) Level3. A set of events are provided to every HTML element. Those events trigger the code in JavaScript.

The *onclick* event type:
This is an event, when clicked, displays desired validation or a warning etc. This very widely used.

EXAMPLE: The below code, when the hello button on the onclick is clicked, it triggers the 'events' function.

```
<html>
<head>
<script type="text/javascript">
<!--
function events() {
  alert("Hello World")
}
//-->
```

```
</script>
</head>
<body>
<input type="button" onclick="sayHello()" value="Say Hello" />
</body>
</html>
```

onsubmit event type:

The onsubmit event is another important event type. Validation occurs when submit a form here. The text will only be sent if the function returns true.

EXAMPLE:

```
<html>
<head>
<script type="text/javascript">
<!--


function validation() {
  all validation goes here


  .........


return either true or false


}
```

```
//-->
</script>
</head>
<body>
<form method="POST" action="t.cgi" onsubmit="return validate()">
.....
<input type="submit" value="Submit" />
</form>
</body>
</html>
```

Chapter 12: Debugging in JavaScript

There are many chances of making a mistake while writing a program.
Such mistakes in a script are referred as bugs.

Debugging is a process of fixing bugs. In the development process debugging is a normal part.
The code we write might have logical errors or syntax errors, which are difficult to identify. Many times, nothing will happen when JavaScript code contains errors. No messages will indicate that there's an error in the program.

Using a JavaScript Validator

If one wants to check code for bugs, then one must make sure that it is a valid code and check it if follows the official syntax rules of the language by running it using a program called Validating parsers, or in short, just Validators. They often come with JavaScript editors and commercial HTML editors.

After visiting the web page, click on the jslint button once you paste the code in the text area. This program will parse through our JavaScript code to ensure that all variables and function definitions are in correct syntax.

Adding Debugging Code to Your Programs:

The alert() or document.write() methods helps in debugging your code. For example:

```
var debugging = true;
var whichImage = "widget";
if( debugging )
 alert( "Calls swapImage() with argument: " + whichImage );
var swapStatus = swapImage( whichImage );
if( debugging )
 alert( "Exits swapImage() with swapStatus=" + swapStatus );
```

*

Using a JavaScript Debugger

A debugger is an application that gives the programmer a control over the execution of the script.

Debugger facilitates the user to examine the flow control. The script is run line by line and can be instructed to stop at particular breakpoints. The programmer can check the script's state and the variables after the execution completes.

Form Validation

After the client enters the required information and clicks the submit button, form validation occurs. The server must send all the data again to the user and request the user to resubmit the form with correct or valid information. This occurs, if the user fails to enter all of the required information or if he enters invalid data.

Before sending the form to the web browser, JavaScript facilitates the validation of the form's data on the client's computer itself.

Two functions will be generally performed by the form validation.

- **Basic Validation** - To ensure that the data has been provided into every mandatory field, first the form needs to be checked. To check the data, it needs to loop through each field.
- **Validation of the Data Format** –This is to check if correct or valid data has been entered in the form.

Basic Form Validation:

Let us see how a basic form validation is performed. We call the validate() function to validate data to validate data when submit event occurs.
Validate() function is implemented as follows:

```
<script type="text/javascript">
<!--
//
Form validation code will come here
.



function validate()
{

 if( document.myForm.Name.value == "" )
 {
  alert( "

Please enter your name

!" );
  document.myForm.Name.focus() ;
  return false;
 }
 if( document.myForm.EMail.value == "" )
 {
  alert( "
```

```
Please enter your Email id!" );
   document.myForm.EMail.focus() ;
   return false;
  }
  if( document.myForm.Zip.value == "" ||
        isNaN( document.myForm.Zip.value ) ||
        document.myForm.Zip.value.length != 5 )
  {
   alert( "
Please enter the
a zip in the format

#####." );
   document.myForm.Zip.focus() ;
   return false;
  }
  if( document.myForm.Country.value == "-1" )
  {
   alert( "Please enter your country name!" );
   return false;
  }
  return( true );
}
//-->
</script>
```

Data Format Validation

Validation of the data can be done before submission of the data to the web server

Validating an email id means the email id should contain an @ sign and a dot (.), can be seen in the below given example.

Also, the last dot should be minimum one character after the @ sign, and the email address shouldn't begin with an @ symbol.

```
<script type="text/javascript">
<!--

function validateEmail()
{

 var emailID = document.myForm.EMail.value;
 atpos = emailID.indexOf("@");
 dotpos = emailID.lastIndexOf(".");
 if (atpos < 1 || ( dotpos - atpos < 2 ))
 {
    alert("
Please enter a valid email ID
")
    document.myForm.EMail.focus() ;
    return false;
 }
 return( true );
}
//-->
```

</script>

Chapter 13: Objects in JavaScript

Java Script- objects:

Objects in JavaScript consist of attributes. An attribute is considered a method, if it has a function, or else an attribute is considered a property.

Object Properties:

The properties of the object can be in any of the data types.

The following syntax is used when one has to add property to an object

```
objectName.objectProperty = propertyValue;
```

Example:

The next example showcases how to use the "title" property of the document object:

```
var strg = document.title;
```

Object Methods:

The functions allowing the object to do a particular task.

Example:

The following example shows how the **write()** method is used:

```
document.write("This is text");
```

User-Defined Objects:

The *new* Operator

The *new* operator helps in creating instances of an object. For creating an object, the constructor method follows the new operator.

In the below given example, Object(), Array(), and Date() are the constructor methods, which are in-built functions of JavaScript.

```
var emp = new Object();
var textbooks = new Array("C", "Python", "Java");
var day = new Date("August 21, 1987");
```

The *Object()* Constructor:

An object can be created and initialized by a constructor. *Object() is a special constructor function that helps to build an object.*

The following example shows to create an object:

```
<html>
<head>
<title>objects defined by the user </title>
<script type="text/javascript">
var book = new Object();   // Creates the object
   book.subject = "Perl"; // Assign properties to the object
   book.author  = "Yashwanth";
</script>
</head>
<body>
<script type="text/javascript">
    document.write("Book name is : " + book.subject + "
<br>");
    document.write("Book author is : " + book.author + "
<br>");
</script>
</body>
</html>
```

JavaScript Native Objects

JavaScript consists of many in-built native objects. Their behavior is not affected by the browser or the platform on which they run.

The following is a list of most widely used native objects in JavaScript:

1. Number Object
2. Boolean Object
3. String Object
4. JavaScript Math Object

The Number Object

The number objects in JavaScript are used to represent numerical values. It can hold both integers and floating point values.

Whenever encountered with numerical values, the web browser converts those values into the number class instances automatically.

Syntax:

The syntax for creating a number object is as follows:

```
var value = new Number(number);
```

Number Methods

The following are the methods that are consisted in the number object.

Method	Description
constructor()	This method is used to return the function that created the instance of the object.
toExponential()	This methods results in the display of a standard range number in the exponential form.
toLocaleString()	This method takes a number and returns a string version for the number. It is done in a format that varies with the locale settings of the web browser.
toPrecision()	This method defines the number of digits to be displayed in a given number. This includes the digits that are present on either side of the decimal.
toString()	This method helps to convert a number into a string value.

valueOf()	This method represents the value of the number.

The Boolean Object

The Boolean object is used to represent a value that can be either a 'True" value or a 'False' value.

Syntax:

A boolen object can be created using the following syntax:

```
var bval = new Boolean(value);
```

The Boolean object's initial value is false if the value parameter holds any of the following values: empty string, null, 0, Nan, -0, null or undefined.

Boolean Methods:

The following are the methods that are consisted in the Boolean object.

Method	Description
toSource()	This method is used to return a string which consists of the Boolean object' source. An equivalent object can be created using that string.
toString()	This method is used to return a 'true' string or a 'false'string based on the object's value.
valueOf()	This method is used to return the Boolean object's primitive value.

The String Object:

The string object allows the user to work with a group of characters and wraps the string primitives of JavaScript with the helper methods.

Syntax:

A String object can be created using the following syntax

```
var value = new String(string);
```

The *string* parameter represents a group of properly encoded characters.

String Methods

The following are the methods that are consisted in the String object.

Method	Description
charCodeAt()	This method is used to return a number, that indicates a character's Unicode value at a given index.
concat()	This method is used to combine characters of two strings and return a newly created string.
match()	This method is used for matching a string with a regular expression.
toLocaleLowerCase()	Used to convert a string into lower case while maintaining the present locale.
toLocaleUpperCase()	Used to convert a string into upper case while maintaining the present locale.
tolowerCase()	Returns a string in which all uppercase letters are converted into lowercase letters
toupperCase()	Returns a string in which all lowercase letters are converted into

uppercase letters.

Math Object

It allows the user to perform many common mathematical calculations. The object methods are called by writing the object's name and then a dot and followed by the method's name. The argument(s) are written in the method parenthesis.

Ex: document.writeln(Math.sqrt(81));

Following are some widely used functions of Math object:

Methods	Description
abs(x)	Absolute value of x
ceil(x)	Rounds x to the smallest integer not less than x.
round(x)	Round x to closest integer.
pow(x , y)	X raised to the power of y
floor(x)	Rounds x to the largest integer not greater than x.

Conclusion

We've come to the end of the book on learning the basics of Java and JavaScript programming.

I want to thank you for purchasing this book. I hope you found this book useful and it helped you in understanding the concepts of programming using Java and Java Script. Being a high-level programming language, it makes it easy to learn the basics and then start writing programs on your own.

No language is rocket-science if studied properly. So is the case with the two languages covered in this book. Dive in and know the basics for these languages.

Thank you!

You May Enjoy My Other Books!

PYTHON: Programming Guide For Beginners: LEARN IN A DAY!
hyperurl.co/python

C ++ Programming : Programming Language For Beginners: LEARN IN A DAY!
hyperurl.co/cplusplus

SQL: Programming Guide: Javascript and Coding: LEARN IN A DAY!
hyperurl.co/sql

Programming: HTML: Programming Guide: Computer Programming: LEARN IN A DAY!

hyperurl.co/html