Java Programming

Notes

Index

1

Java Overview

Java programming language was originally developed by Sun Microsystems which was initiated by James Gosling and released in 1995 as core component of Sun Microsystems‘ Java platform (Java 1.0 [J2SE]). As of December 2008, the latest release of the Java Standard Edition is 6 (J2SE). With the advancement of Java and its widespread popularity, multiple configurations were built to suite various types of platforms. Ex: J2EE for Enterprise Applications, J2ME for Mobile Applications. Sun Microsystems has renamed the new J2 versions as Java SE, Java EE and Java ME, respectively. Java is guaranteed to be **Write Once, Run Anywhere.** Java is:
Object Oriented: In Java, everything is an Object. Java can be easily extended since it is based on the Object model.

Platform independent: Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by virtual Machine (JVM) on whichever platform it is being run.

Simple: Java is designed to be easy to learn. If you understand the basic concept of OOP, Java would be easy to master.

Secure: With Java's secure feature, it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.

Architectural-neutral: Java compiler generates an architecture-neutral object file format, which makes the compiled code to be executable on many processors, with the presence of Java runtime system.

Portable: Being architectural-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C with a clean portability boundary which is a POSIX subset.

Robust: Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.

Multithreaded: With Java's multithreaded feature, it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct smoothly running interactive applications.

Interpreted: Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and lightweight process.

High Performance: With the use of Just-In-Time compilers, Java enables high performance.

Distributed: Java is designed for the distributed environment of the internet.

Dynamic: Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

History of Java:

James Gosling initiated the Java language project in June 1991 for use inone of his many set-top box projects. The language, initially called Oak after an oak tree that stood outside Gosling's office, also went by the name Green and ended up later being renamed as Java, from a list of random words. Sun released the first public implementation as Java 1.0 in 1995. It promised **Write Once, Run Anywhere** (WORA), providing no-cost run-times on popular platforms. On 13 November 2006, Sun released much of Java as free and open source software under the terms of the GNU General Public License (GPL). On 8 May 2007, Sun finished the process, making all of Java's core code free and open-source, aside from a small portion of code to which Sun did not hold the copyright.

Tools you will need:

For performing the examples discussed in this tutorial, you will need a Pentium 200-MHz computer with a minimum of 64 MB of RAM (128 MB of RAM recommended). You also will need the following software:

Linux 7.1 or Windows 95/98/2000/XP operating system. Java JDK 5

Microsoft Notepad or any other text editor

2

Java Environment Setup

Before we proceed further, it is important that we set up the Java environment correctly. This section guides you on how to download and set up Java on your machine. Please follow the following steps to set up the environment.

Java SE is freely available from the link Download Java. So you download a version based on your operating system. Follow the instructions to download Java and run the **.exe** to install Java on your machine. Once you installed Java on your machine, you would need to set environment variables to point to correct installation directories:

Setting up the path for windows 2000/XP:

Assuming you have installed Java in *c:\Program Files\java\jdk* directory: Right-click on 'My Computer' and select'Properties'.

Click on the 'Environment variables' button under the 'Advanced' tab.

Now, alter the 'Path' variable so that it also contains the path to the Java executable. Example, if the path is currently set to 'C:\WINDOWS\SYSTEM32', then change your path to read 'C:\WINDOWS\SYSTEM32;c:\Program Files\java\jdk\bin'.

Setting up the path for windows 95/98/ME:

Assuming you have installed Java in *c:\Program Files\java\jdk* directory:

- Edit the 'C:\autoexec.bat' file and add the following line at the end: 'SET PATH=%PATH%;C:\Program Files\java\jdk\bin'

Setting up the path for Linux, UNIX, Solaris, FreeBSD:
Environment variable PATH should be set to point to where the Java binaries have been installed. Refer to your shell documentation if you have trouble doing this. Example, if you use *bash* as your shell, then you would add the following line to the end of your '.bashrc: export PATH=/path/to/java:$PATH'

Java popular editors:

To write your Java programs, you will need a text editor. There are even more sophisticated IDEs available in the market. But for now, you can consider one of the following:

Notepad: On Windows machine, you can use any simple text editor like Notepad.

Netbeans: Is a Java IDE that is open-source and free which can be downloaded fromhttp://www.netbeans.org/index.html.

Eclipse: Is also a Java IDE developed by the eclipse open-source community and can be downloaded from http://www.eclipse.org/.

3

Basic Syntax

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

Object - Objects have states and behaviors. Example: A dog has states-color, name, breed as well as behaviors -wagging, barking, eating. An object is an instance of a class.

Class - A class can be defined as a template/blue print that describes the behaviors/states that object of its type support.

Methods - A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.

Instance Variables - Each object has its unique set of instance variables. An object's state is created by the values assigned to these instance variables.

First Java Program:

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

```
public class MyFirstJavaProgram{
/* This is my first java program. * This will print 'Hello World' as the output */ public static void
main(String[]args){
```

```
System.out.println("Hello World");
// prints HelloWorld
      }
 }
```

Let's look at how to save the file, compile and run the program. Please follow the steps given below:

Open notepad and add the *code* as above. Save the

file as: MyFirstJavaProgram.java.

Open a command prompt window and go o the directory where you saved the class. Assume it's C:\.

Type ' javac MyFirstJavaProgram.java ' and press enter to compile your code. If there are no errors in your code, the command prompt will take you to the next line(Assumption : The path variable is set).

Now, type ' java MyFirstJavaProgram ' to run your program. You will be able

to see ' Hello World ' printed on the window.

C :> javac MyFirstJavaProgram.java C :> java MyFirstJavaProgram HelloWorld Basic Syntax:
About Java programs, it is very important to keep in mind the following points.

Case Sensitivity - Java is case sensitive, which means identifier **Hello** and **hello** would have different meaning in Java.

Class Names - For all class names, the first letter should be in Upper Case. If several words are used to form a name of the class, each inner word's first letter should be in Upper Case. Example *class MyFirstJavaClass*

Method Names - All method names should start with a Lower Case letter. If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case. Example *public void myMethodName()*

Program File Name - Name of the program file should exactly match the class name. When saving the file, you should save it using the class name (Remember Java is case sensitive) and append '.java' to the end of the name (if the file name and the class name do not match your program will not compile). Example : Assume 'MyFirstJavaProgram' is the class name, then the file should be saved as*'MyFirstJavaProgram.java'*

public static void main(String args[]) - Java program processing starts from the main() method, which is a mandatory part of every Java program.

Java Identifiers:

All Java components require names. Names used for classes, variables and methods are called identifiers. In Java, there are
several points to remember about identifiers. They are as follows:

All identifiers should begin with a letter (A to Z or a to z), currency character ($) or an underscore (_).
After the first character, identifiers can have any combination of characters. A keyword cannot
be used as an identifier.
Most importantly identifiers are case sensitive.
Examples of legal identifiers: age, $salary, _value, _1_value

Examples of illegal identifiers: 123abc, -salary

Java Modifiers:

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

Access Modifiers: default, public, protected, private

Non-access Modifiers: final, abstract, strictfp

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

Java Variables:

We would see following type of variables in Java: Local
Variables
Class Variables (Static Variables) Instance
Variables (Non-static variables)

Java Arrays:

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

Java Enums:

Enums were introduced in java 5.0. Enums restrict a variable to have one of only a few predefined values. The values in this enumerated list are called enums. With the use of enums, it is possible to reduce the number of bugs in your code. For example, if we consider an application for a fresh juice shop, it would be possible to restrict the glass size to small, medium and large. This would make sure that it would not allow anyone to order any size other than the small, medium or large.

Example:

```
Class FreshJuice{
enum FreshJuiceSize{ SMALL, MEDUIM, LARGE }
FreshJuiceSize size;
 }
public class FreshJuiceTest{
public static void main(String args[]){
FreshJuice juice =new FreshJuice();
juice.size =FreshJuice.FreshJuiceSize.MEDUIM ;
 }
}
```

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

Java Keywords:

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

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

transient	try	void	volatile	while

Comments in Java

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

```
public class MyFirstJavaProgram{

/* This is my first java program. * This will print 'Hello World' as the
output * This is an example of multi-line comments. */ public static void
main(String[]args) {

// This is an example of single line comment /* This is also an example of
single line comment.  */

System.out.println("Hello World");

   }
}
```

4

Basic Data Types

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables. There are two data types available in Java:

Primitive Data Types

Reference/Object Data Types

Primitive Data Types:

There are eight primitive data types supported by Java. Primitive data types are predefined by the language and named by a keyword. Let us now look into detail about the eight primitive data types.

byte:

Byte data type is an 8-bit signed two's complement integer.

- Minimum value is -128 (-2^7)
- Maximum value is 127 (inclusive)(2^7 -1)

- Default value is 0
- 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 int.
- Example: byte a = 100, byte b = -50

short:

Short data type is a 16-bit signed two's complement integer.

Minimum value is -32,768 (-2^15)

- Maximum value is 32,767(inclusive) (2^15 -1)
- Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an int
- Default value is 0.
- Example: short s= 10000, short r = -20000

int:

int data type is a 32-bit signed two's complement integer.

- Minimum value is - 2,147,483,648.(-2^31)
- Maximum value is 2,147,483,647(inclusive).(2^31 -1)
- Int is generally used as the default data type for integral values unless there is a concern about memory.
- The default value is 0.

- Example: int a = 100000, int b = -200000

long:

Long data type is a 64-bit signed two's complement integer.

- Minimum value is -9,223,372,036,854,775,808.(-2^63)
- Maximum value is 9,223,372,036,854,775,807 (inclusive). (2^63 -1)
- This type is used when a wider range than int is needed.
- Default value is 0L.
- Example: int a = 100000L, int b = -200000L

float:

Float data type is a single-precision 32-bit IEEE 754 floating point.

- Float is mainly used to save memory in large arrays of floating point numbers.
- Default value is 0.0f.
- Float data type is never used for precise values such as currency.
- Example: float f1 = 234.5f

double:

double data type is a double-precision 64-bit IEEE 754 floating point.

- This data type is generally used as the default data type for decimal values, generally the default choice.
- Double data type should never be used for precise values such as currency.
- Default value is 0.0d.
- Example: double d1 = 123.4

boolean:

boolean data type represents one bit of information.

- There are only two possible values: true and false.
- This data type is used for simple flags that track true/false conditions.
- Default value is false.
- Example: boolean one = true

char:

char data type is a single 16-bit Unicode character.

- Minimum value is '\u0000' (or 0).
- Maximum value is '\uffff' (or 65,535 inclusive).
- Char data type is used to store any character.
- Example: char letterA ='A'

Reference Data Types:

Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy, etc.

- Class objects and various types of array variables come under reference data type.
- Default value of any reference variable is null.
- A reference variable can be used to refer to any object of the declared type or any compatible type.
- Example: Animal animal = new Animal("giraffe");

Java Literals:

A literal is a source code representation of a fixed value. They are represented directly in the code without any computation. Literals can be assigned to any primitive type variable. For example:

byte a =68; char a

='A'

byte, int, long, and short can be expressed in decimal(base 10),hexadecimal(base 16) or octal(base 8) number systems as well. Prefix 0 is used to indicate octal and prefix 0x indicates hexadecimal when using these number systems for literals. For example:

int decimal=100;

int octal =0144; int
hexa =0x64;

String literals in Java are specified like they are in most other languages by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are:
"Hello World"
"two\n lines"
"\"This is in quotes\""

String and char types of literals can contain any Unicode characters. For example:
char a ='\u0001';
String a ="\u0001";
Java language supports few special escape sequences for String and char literals as well. They are:

Notation	**Character represented**
\n	Newline (0x0a)
\r	Carriage return (0x0d)
\f	Formfeed (0x0c)
\b	Backspace (0x08)
\s	Space (0x20)
\t	Tab
\"	Double quote
\'	Single quote
\\	Backslash
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal UNICODE character (xxxx)

5

Variable Types

A variable provides us with named storage that our programs can manipulate. Each variable in Java has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable. You must declare all variables before they can be used. The basic form of a variable declaration is shown here:

data type variable [= value][, variable [= value] ...] ;

Here *data type* is one of Java's datatypes and *variable* is the name of the variable. To declare more than one variable of the specified type, you can use a comma-separated list. Following are valid examples of variable declaration and initialization in Java:

```
int a, b, c;            // Declares three ints, a, b, and c.
int a = 10, b  = 10;          // Example of initialization
 byte B = 22;                 // initializes a byte  type variable B.
double  pi =  3.14159;        // declares and assigns a  value  of  PI.
char a = 'a';           // the char variable a is initialized with value  'a'
```

This chapter will explain various variable types available in Java Language. There are three kinds of variables in Java:

- Local variables
- Instance variables
- Class/static variables

Local variables:

- Local variables are declared in methods, constructors, or blocks.
- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor or block.
- Access modifiers cannot be used for local variables.
- Local variables are visible only within the declared method, constructor or block.
- Local variables are implemented at stack level internally.
- There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.

Example:

Here, *age* is a local variable. This is defined inside *pupAge()* method and its scope is limited to this method only.

```
public class Test{
public void pupAge() {
int age =  0;
```

```
        age = age + 7;
        System.out.println("Puppy age is : " + age);
         }

         public static void main(String args[]) {
        Test test = new Test();
         test.pupAge();
            }
         }
```

This would produce the following result:

```
Puppy age is:  7
```

Example:

Following example uses *age* without initializing it, so it would give an error at the time of compilation.

```
public class Test {

public void pupAge() {

int age;

age = age + 7;

System.out.println("Puppy age is : " + age);

 }

public static void main(String args[]){

Test test = new Test();
test.pupAge();
  }
}
```

This would produce the following error while compiling it:

Test.java:4:variable number might not have been initialized

```
age = age + 7;
1 error
```

Example:

```
//Demonstrate lifetime of a variable.
class LifeTime {
public static void main(String args[]) {
int x;
for(x = 0; x < 3; x++) {
int y = -1; // y is initialized each time block is entered
System.out.println("y is: " + y); // this always prints -1
y = 100;
System.out.println("y is now: " + y);
    }
  }
}
```

Instance variables:

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap, a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
- Instance variables can be declared in class level before or after use.
- Access modifiers can be given for instance variables.
- The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level). However visibility for subclasses can be given for these variables with the use of access modifiers.
- Instance variables have default values. For numbers the default value is 0, for Booleans it is false and for object references it is null. Values can be assigned during the declaration or within the constructor.
- Instance variables can be accessed directly by calling the variable name inside the class. However within static methods and different class (when instance variables are given accessibility) should be called using the fully qualified name . *ObjectReference.VariableName*.

Example:

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

```
// this instance variable is visible for any child class.
 public String name;
// salary variable is visible in Employee class only.
private double salary;
// The name variable is assigned in the constructor.
public Employee (String empName){
 name = empName;
 }
// The salary variable is assigned a value.
public void setSalary(double empSal){
salary = empSal;
}
// This method prints the employee details.
public void printEmp(){
System.out.println("name : " + name );
System.out.println("salary :" + salary);
}
public static void main(String args[]){
Employee empOne = new Employee("Ransika");
empOne.setSalary(1000); empOne.printEmp();
  }
}
```

This would produce the following result:

```
name : Ransika
salary :1000.0
```

Class/static variables:

- Class variables also known as static variables are declared with the *static* keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final and static. Constant variables never change from their initial value.
- Static variables are stored in static memory. It is rare to use static variables other than declared final and used as either public or private constants.
- Static variables are created when the program starts and destroyed when the program stops.
- Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
- Default values are same as instance variables. For numbers, the default value is 0; for Booleans, it is false; and for object references, it is null. Values can be assigned during the declaration or within the constructor. Additionally values can be assigned in special static initializer blocks.
- Static variables can be accessed by calling with the class name .
  ClassName.VariableName.
- When declaring class variables as public static final, then variables names (constants) are all in upper case. If the static variables are not

public and final the naming syntax is the same as instance and local variables.

Example:

```
import java.io.*;
public class Employee{
 // salary variable is a private static variable
private static double  salary;
// DEPARTMENT is a constant
 public static final String DEPARTMENT = "Development ";
 public static void main(String  args[]){
salary = 1000;
System.out.println(DEPARTMENT+"average salary:"+salary);
   }
}
```

This would produce the following result:

```
Development average salary:1000
```

Note: If the variables are access from an outside class the constant should be accessed as Employee.DEPARTMENT

6

Basic Operators

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators
- Misc Operators

The Arithmetic Operators:

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators: Assume integer variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	A + B will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	A - B will give -10
*	Multiplication - Multiplies values on either side of the operator	A * B will give 200
/	Division - Divides left hand operand by right hand operand	B / A will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	B % A will give 0
++	Increment - Increases the value of operand by 1	B++ gives 21
--	Decrement - Decreases the value of operand by 1	B-- gives 19

Example

The following simple example program demonstrates the arithmetic operators. Copy and paste the following Java program in Test.java file and compile and run this program:

```
public class Test{
 public static void main(String args[]){
int a =10;
int b =20;
int  c =25;
 int d =25;
System.out.println("a + b = "+(a + b));
System.out.println("a - b = "+(a - b));
System.out.println("a * b = "+(a * b));
System.out.println("b / a = "+(b / a));
System.out.println("b % a = "+(b % a));
System.out.println("c % a = "+(c % a));
System.out.println("a++ = "+(a++));
System.out.println("b-- = "+(a--));
// Check the difference in d++ and ++d
System.out.println("d++ = "+(d++));
System.out.println("++d = "+(++d));
   }
 }
```

This would produce the following result:

```
a + b =30 a -
b =-10
a * b =200 b
/ a =2
b % a =0 c
% a =5
a++=10 b--
=11
d++=25
++d =27
```

The Relational Operators:

There are following relational operators supported by Java language: Assume variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
==	Checks if the values of two operands areequal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A>=B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Example

The following simple example program demonstrates the relational operators. Copy and paste the following Java program in Test.java file and compile and run this program. :

```
public class Test{
public static void main(String args[]){
int a =10; int b =20;
System.out.println("a == b = "+(a == b));
System.out.println("a != b = "+(a != b));
System.out.println("a > b = "+(a > b));
System.out.println("a < b = "+(a < b));
System.out.println("b >= a = "+(b >= a));
System.out.println("b <= a = "+(b <= a));
  }
}
```

This would produce the following result:

```
a == b =false
a != b =true
a > b =false
a < b  =true
b >= a =true
b <= a =false
```

The Bitwise Operators:

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte. Bitwise operator works on bits and performsbit-by-bit operation.

Assume if a = 60; and b = 13;

now in binary format they will be as follows: a = 0011

```
      1100
      b = 0000 1101
       -----------------
a&b = 0000 1100
   a|b =    0011 1101
   a^b =    0011 0001
    ~a =    1100 0011
```

The following table lists the bitwise operators:

Assume integer variable A holds 60 and variable B holds 13, then:

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -60 which is 1100 0011
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>>	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

Example

The following simple example program demonstrates the bitwise operators. Copy and paste the following Java program in Test.java file and compile and run this program:

```
public class Test  {
```

```
public static void main(String args[]) {
int a =60; /* 60 = 0011 1100 */
int b =13;   /* 13 = 0000 1101  */
 int c =0;
 c = a & b; /* 12 = 0000 1100 */
System.out.println("a & b = "+ c );
c = a | b; /* 61 = 0011 1101  */
System.out.println("a | b = "+ c );
c = a ^ b; /* 49 = 0011 0001 */
System.out.println("a ^ b = "+ c );
c =~a; /*-61 = 1100 0011 */
System.out.println("~a = "+ c );
c = a <<2; /* 240 = 1111 0000 */
System.out.println("a << 2 = "+ c );
c = a >>2; /* 215 = 1111 */
System.out.println("a >> 2 = "+ c );
c = a >>>2; /* 215 = 0000 1111 */
System.out.println("a >>> 2 = "+ c );
        }
}
```

This would produce the following result:

```
a & b =12
a | b =61
a ^ b =49
~a =-61
 a <<2=240
a >>15
a >>>15
```

The Logical Operators:

The following table lists the logical operators:

Assume Boolean variables A holds true and variable B holds false, then:

Operator	**Description**	**Example**
&&	Called Logical AND operator. If both theoperands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

Example

The following simple example program demonstrates the logical operators. Copy and paste the following Java program in Test.java file and compile and run this program:

```
public class Test{
public static void main(String args[]){
 boolean a =true;
 boolean b =false;
 System.out.println("a && b = "+(a&&b));
System.out.println("a || b = "+(a||b));
System.out.println("!(a && b) = "+!(a && b));
     }
}
```

This would produce the following result:

```
a && b  =false
```

a || b =true

!(a && b)=true

The Assignment Operators:

There are following assignment operators supported by Java language:

Operato r	**Description**	**Example**
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A

<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Example:

The following simple example program demonstrates the assignment operators. Copy and paste the following Java program in Test.java file and compile and run this program:

```
public class Test{
public static void main(String args[]){
int a =10;
int b =20;
int c =0;
c = a + b;
System.out.println("c = a + b = "+ c );
c += a  ;
 System.out.println("c += a = "+ c );
 c -= a  ;
System.out.println("c -= a = "+ c );
 c *= a  ;
System.out.println("c *= a = "+ c );
 a =10;
c =15;
c /= a  ;
System.out.println("c /= a = "+ c );
a =10;
c =15;
c %= a  ;
System.out.println("c %= a = "+ c );
c <<=2;
System.out.println("c <<= 2 = "+ c );
 c >>=2;
System.out.println("c >>= 2 = "+ c );
c >>=2;
System.out.println("c >>= a = "+ c );
c &= a ;
System.out.println("c &= 2 = "+ c );
 c ^= a  ;
 System.out.println("c ^= a = "+ c );
 c |= a ;
System.out.println("c |= a = "+ c );
     }
 }
```

This would produce the following result:

```
c = a + b =30
c += a =40
c -= a  =30
c *= a =300
c /= a =1
c %= a =5
c <<=2=20
c >>=2=5
c >>=2=1
```

```
c &= a =0
c ^= a =10
c |= a =10
```

Misc Operators

There are few other operators supported by Java Language. Conditional

Operator (?:):

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide which value should be assigned to the variable. The operator is written as:

variable x =(expression)? value iftrue: value iffalse

Following is the example:

```
public class Test{
public static void main(String args[]){
int a ,  b;
a =10;
b =(a ==1)?20:30;
 System.out.println("Value of b is : "+ b );
 b =(a ==10)?20:30;
System.out.println("Value of b is : "+ b );
    }
 }
```

This would produce the following result:

```
Value of b is:30
Value of b is:20
```

instanceof Operator:

This operator is used only for object reference variables. The operator checks whether the object is of a particular type(class type or interface type). instanceof operator is written as:

(Object reference variable) instanceof (class/interface type)

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true. Following is the example:

String name = —James‖;

boolean result = name instanceof String; // This will return true since name is type of String

This operator will still return true if the object being compared is the assignment compatible with the type on the right. Following is one more **Example:**

```
Class Vehicle{
} public class Car extends Vehicle{
public static void main(String args[]){
Vehicle ab =newCar();
boolean result = ab instanceof Car;
System.out.println(result);
   }
}
```

This would produce the following result:

True

Precedence of Java Operators:

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator: For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7. Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	() [] . (dot operator)	Left to right
Unary	++ - - ! ~	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	>>>>><<	Left to right
Relational	>>= <<=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

7

Loop Control

There may be a situation when we need to execute a block of code several number of times and is often referred to as a loop. Java has very flexible three looping mechanisms. You can use one of the following three loops:

- while Loop
- do...while Loop
- for Loop

As of Java 5, the *enhanced for loop* was introduced. This is mainly used for Arrays.

The while Loop:

A while loop is a control structure that allows you to repeat a task a certain number of times.

Syntax:

The syntax of a while loop is:

```
while(Boolean_expression) {
 //Statements
 }
```

When executing, if the *boolean_expression* result is true, then the actions inside the loop will be executed. This will continue as long as the expression result is true. Here, key point of the *while* loop is that the loop might notever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

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");
    }
  }
}
```

This would produce the following result:

```
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
```

The do...while Loop:

A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax:

The syntax of a do...while loop is:

```
do {
 //Statements }
while(Boolean_expression);
```

Notice that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested. If the Boolean expression is true, the flow of control jumps back up to do, and the statements in the loop execute again. This process repeats until the Boolean expression is false.

Example:

```
public class Test{
 public static void main(String args[]){
int x =10;
 do{
System.out.print("value of x : "+ x );
 x++;
System.out.print("\n");
     }while( x <20);
  }
}
```

This would produce the following result:

```
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
```

The for Loop:

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times. A for loop is useful when you know how many times a task is to berepeated.

Syntax:

The syntax of a for loop is:

```
for(initialization;Boolean_expression; update) {
//Statements
}
```

Here is the flow of control in a for loop:

- The initialization step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.

- Next, the Boolean expression is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement past the for loop.
- After the body of the for loop executes, the flow of control jumps back up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the Boolean expression.
- The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then update step, then Boolean expression). After the Boolean expression is false, the for loop terminates.

Example:

```
public class Test{
public static void main(String args[]){
for(int x =10; x <20; x = x+1){
System.out.print("value of x : "+ x );
System.out.print("\n");
    }
  }
}
```

This would produce the following result:

```
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
```

Enhanced for loop in Java:

As of Java 5, the enhanced for loop was introduced. This is mainly used for Arrays.

Syntax:

The syntax of enhanced for loop is:

```
for(declaration : expression) {
//Statements
}
```

- **Declaration**: The newly declared block variable, which is of a type compatible with the elements of the array you are accessing. The variable will be available within the for block and its value would be the same as the current array element.
- **Expression**: This evaluates to the array you need to loop through. The expression can be an array variable or method call that returns an array.

Example:

```
public class Test{
public static void main(String args[]){
 int[] numbers ={10,20,30,40,50};
 for(int x : numbers ){
 System.out.print(x);
 System.out.print(",");
 }
System.out.print("\n");
String[] names ={"swapna","balu","kiran","ramu"};
for(String name : names ){
 System.out.print( name );
 System.out.print(",");
      }
    }
 }
```

This would produce the following result:

```
10,20,30,40,50,
swapna,balu,kiran,ramu,
```

The break Keyword:

The *break* keyword is used to stop the entire loop. The break keyword must be used inside any loop or a switch statement.

The break keyword will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax:

The syntax of a break is a single statement inside any loop:

```
break;
```

Example:

```
public class Test{
public static void main(String args[]){
int[] numbers ={10,20,30,40,50};
```

```
 for(int x : numbers){
 if(x ==30){ break;
 }
System.out.print( x );
System.out.print("\n");
      }
   }
}
```

This would produce the following result:

```
10
20
```

The continue Keyword:

The *continue* keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.

- In a for loop, the continue keyword causes flow of control to immediately jump to the update statement.
- In a while loop or do/while loop, flow of control immediately jumps to the Boolean expression.

Syntax:

The syntax of a continue is a single statement inside any loop:

continue;

Example:

```
public class Test{
 public static void main(String args[]){
 int[] numbers ={10,20,30,40,50};
 for(int x : numbers){
 if( x ==30){
 continue;
}
System.out.print( x );
System.out.print("\n");
    }
  }
 }
```

This would produce the following result:

```
10
20
40
50
```

8

Decision making

There are two types of decision making statements in Java. They are:

- if statements
- switch statements

The if Statement:

An if statement consists of a Boolean expression followed by one or more statements.

Syntax:

The syntax of an if statement is:

```
if(Boolean_expression) {
//Statements will execute if the Boolean expression is true
 }
```

If the Boolean expression evaluates to true, then the block of code inside the if statement will be executed. If not, the first set of code after the end of the if statement(after the closing curly brace) will be executed.

Example:

```
public class Test{
public static void main(String args[]){
 int x =10;
 if( x <20){
System.out.print("This is if  statement");
    }
  }
}
```

This would produce the following result:

```
This is if statement
```

The if...else Statement:

An if statement can be followed by an optional *else* statement, which executes when the Boolean expression is false.

Syntax:

The syntax of an if...else is:

```
if(Boolean_expression){
 //Executes when the Boolean expression is true
 }else
{
//Executes when the Boolean expression is false
 }
```

Example:

```
public class Test{
 public static void main(String args[]){
 int x =30;
 if(x <20){
System.out.print("This is if  statement");
 }else
{
System.out.print("This is else  statement");
     }
   }
}
```

This would produce the following result:

```
 This is else statement
```

The if...else if...else Statement:

An if statement can be followed by an optional *else if...else* statement, which is very useful to test various conditions using single if...else if statement. When using if, else if , else statements there are few points to keep in mind.

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax:

The syntax of an if...else is

```
if(Boolean_expression1){
 //Executes when the Boolean expression 1 is true
}elseif(Boolean_expression2){
 //Executes when the Boolean expression 2 is true
}elseif(Boolean_expression3){
//Executes when the Boolean expression 3 is true
 }else
{ //Executes when the none of the above condition is true.
 }
```

Example:

```
public class Test{
 public static void main(String args[]){
 int x =30;
if( x ==10){
System.out.print("Value of X is 10");
}elseif( x ==20){
 System.out.print("Value of X is 20");
}elseif( x ==30){
 System.out.print("Value of X is 30");
 }else{ System.out.print("This is else statement");
      }
    }
 }
```

This would produce the following result:

```
Value of X is 30
```

Nested if...else Statement:

It is always legal to nest if-else statements which means you can use one if or else if statement inside another if or else if statement.

Syntax:

The syntax for a nested if...else is as follows:

```
if(Boolean_expression1){
//Executes when the Boolean expression 1 is true
if(Boolean_expression2){
 //Executes when the Boolean expression 2 is true }
 }
```

You can nest *else if...else* in the similar way as we have nested *if* statement. Example:

```
public class Test{
public static void main(String args[]){
 int x =30;
int y =10;
 if( x ==30){
 if( y ==10){
System.out.print("X = 30 and Y = 10");
    }
  }
}
```

This would produce the following result:

```
X =30 and Y =10
```

The switch Statement:

A *switch* statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax:

The syntax of enhanced for loop is:

```
switch(expression){
case value : //Statements
break;
//optional case value  :
 //Statements
break;
//optional
//You can have any number of case statements.
 default://Optional
 //Statements
 }
```

The following rules apply to a switch statement:

- The variable used in a switch statement can only be a byte, short, int, or char.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The value for a case must be the same data type as the variable in the switch and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a *break* statement is reached.

- When a *break* statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.

- Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.

- A *switch* statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Example:

```
public class Test{
 public static void main(String args[]){
 char grade = args[0].charAt(0);
 switch(grade) {
 case'A': System.out.println("Excellent!");
              break;
case'B':
case'C': System.out.println("Well done");
             break;
case'D': System.out.println("You passed");
case'F': System.out.println("Better try again");
              break;
default:  System.out.println("Invalid  grade");
 }
 System.out.println("Your grade is "+   grade);
   }
}
```

Compile and run above program using various command line arguments. This would produce the following result:

```
D>java Test a
Invalid grade Your
grade is a a D> java
Test A Excellent!
Your grade is a A
D>java Test C
Welldone
Your grade is a C D>
```

9

String handling

Strings:

Java string is a sequence of characters. They are objects of type String. Once a String object is created it cannot be changed. Stings are Immutable.

To get changeable strings use the class called StringBuffer.

String and StringBuffer classes are declared final, so there cannot be subclasses of these classes.

The default constructor creates an empty string.

String s = new String();

Creating Strings :

String str = "abc"; is equivalent to: char data[] = {'a', 'b', 'c'};

String str = new String(data);

If data array in the above example is modified after the string object str is created, then str remains unchanged.

Construct a string object by passing another string object. String str2 = new String(str);

String Operations :

The **length()** method returns the length of the string. Eg:

System.out.println(―Hello‖.length()); // prints 5

The + operator is used to concatenate two or more strings.

Eg: String myname = ―Harry‖

String str = ―My name is‖ + myname+ ―.‖;

For string concatenation the Java compiler converts an operand to a String whenever the other operand of the + is a String object.

Characters in a string can be extracted in a number of ways. public char **charAt**(int index)

– Returns the character at the specified index. An index ranges from 0 to length() - 1. The first character of the sequence is at index 0, the next at index 1, and so on, as for array indexing.

char ch;

ch = ―abc‖.charAt(1); //ch = ―b‖

getChars() - Copies characters from this string into the destination character array.

public void **getChars**(int srcBegin, int srcEnd, char[] dst, int dstBegin)

– srcBegin - index of the first character in the string to copy.
– srcEnd - index after the last character in the string to copy.
– dst - the destination array.
– dstBegin - the start offset in the destination array.

equals() - Compares the invoking string to the specified object. The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as the invoking object.

public boolean **equals**(Object anObject)

equalsIgnoreCase()- Compares this String to another String, ignoring case considerations. Two strings are considered equal ignoring case if they are of the same length, and corresponding characters in the two strings are equal ignoring case.

public boolean **equalsIgnoreCase**(String anotherString)

startsWith() – Tests if this string starts with the specified prefix. public

boolean **startsWith**(String prefix)

—Figure‖.startsWith(—Fig‖); // true

endsWith() - Tests if this string ends with the specified suffix. public

boolean **endsWith**(String suffix)

—Figure‖.endsWith(—re‖); // true

startsWith() -Tests if this string starts with the specified prefix beginning at a specified index.

public boolean **startsWith**(String prefix, int toffset) prefix - the

prefix.

toffset - where to begin looking in the string.

—figure‖.startsWith(—gure‖, 2); // true

compareTo() - Compares two strings lexicographically.

- The result is a negative integer if this String object lexicographically precedes the argument string.
- The result is a positive integer if this String object lexicographically follows the argument string.
- The result is zero if the strings are equal.
- compareTo returns 0 exactly when the equals(Object) method would return true.

public int **compareTo**(String anotherString) public int

compareToIgnoreCase(String str)

indexOf() – Searches for the first occurrence of a character or substring. Returns -1 if the character does not occur.

public int indexOf(int ch)- Returns the index within this string of the first occurrence of the specified character.

public int indexOf(String str) - Returns the index within this string of the first occurrence of the specified substring.

String str = ―How was your day today?‖;

str.indexof(‗t‘);

str(―was‖);

public int indexOf(int ch, int fromIndex)- Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.

public int indexOf(String str, int fromIndex) - Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.

String str = ―How was your day today?‖;

str.indexof(‗a‘, 6);

str(―was‖, 2);

lastIndexOf() –Searches for the last occurrence of a character or substring. The methods are similar to indexOf().

substring() - Returns a new string that is a substring of this string. The substring begins with the character at the specified index and extends to the end of this string.

public String substring(int beginIndex)

Eg: "unhappy".substring(2) returns "happy"

- public String substring(int beginIndex, int endIndex) Eg:

"smiles".substring(1, 5) returns"mile―

concat() - Concatenates the specified string to the end of this string.

If the length of the argument string is 0, then this String object is returned.

Otherwise, a new String object is created, containing the invoking string with the contents of the str appended to it.

public String concat(String str) "to".concat("get").concat("her") returns "together"

replace()- Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.

public String replace(char oldChar, char newChar)

"mesquite in your cellar".replace('e', 'o') returns "mosquito in your collar"

trim() - Returns a copy of the string, with leading and trailing whitespace omitted.

public String trim()

String s = ― Hi Mom! ―.trim();

S = ―Hi Mom!‖

valueOf() –Returns the string representation of the char array argument. public static

String valueOf(char[] data)

The contents of the character array are copied; subsequent modification of the character array does not affect the newly created string.

Other forms are:

public static String valueOf(char c) public

static String valueOf(boolean b) public static

String valueOf(int i) public static String

valueOf(long l) public static String

valueOf(float f)

public static String valueOf(double d)

- toLowerCase(): Converts all of the characters in a String to lower case.
- toUpperCase(): Converts all of the characters in this String to upper case.

public String toLowerCase() public

String toUpperCase()

Eg: ―HELLO THERE‖.toLowerCase();

―hello there‖.toUpperCase();

StringBuffer - class

A **StringBuffer** is like a String, but can be modified.

The length and content of the StringBuffer sequence can be changed through certain method calls.

- StringBuffer defines three constructors:
  - StringBuffer()
  - StringBuffer(int size)
  - StringBuffer(String str)

StringBuffer operations:

- The principal operations on a StringBuffer are the append and insert methods, which are overloaded so as to accept data of any type.

Here are few append methods: StringBuffer

append(String str) StringBuffer append(int

num)

- The append method always adds these characters at the end of the buffer.
- The insert method adds the characters at a specified point. Here are few

insert methods:

StringBuffer insert(int index, String str)

StringBuffer append(int index, char ch)

Index specifies at which point the string will be inserted into the invoking StringBuffer object.

delete() - Removes the characters in a substring of this StringBuffer. The substring begins at the specified start and extends to the character at index end - 1 or to the end of the StringBuffer if no such character exists. If start is equal to end, no changes are made.

public StringBuffer delete(int start, int end)

replace() - Replaces the characters in a substring of this StringBuffer with characters in the specified String.
public StringBuffer **replace**(int start, int end, String str)

substring() - Returns a new String that contains a subsequence of characters currently contained in this StringBuffer. The substring begins at the specified index and extends to the end of the StringBuffer.
public String substring(int start)

reverse() - The character sequence contained in this string buffer is replaced by the reverse of the sequence.
public StringBuffer reverse()

length() - Returns the length of this string buffer. public int length()

capacity() - Returns the current capacity of the String buffer. The capacity is the amount of storage available for newly inserted characters.

public int capacity()

charAt() - The specified character of the sequence currently represented by the string buffer, as indicated by the index argument, is returned.

public char charAt(int index)

getChars() - Characters are copied from this string buffer into the destination character array dst. The first character to be copied is at index srcBegin; the last character to be copied is at index srcEnd-1.

public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)

setLength() - Sets the length of the StringBuffer. public

void setLength(int newLength)

Examples: StringBuffer

StringBuffer sb = new StringBuffer(―Hello‖); sb.length(); // 5

sb.capacity(); // 21 (16 characters room is added

if no size is specified)

sb.charAt(1); // e sb.setCharAt(1,‗i‗); //

Hillo sb.setLength(2); // Hi

sb.append(―l‖).append(―l‖); // Hill sb.insert(0,

―Big ―); //Big Hill sb.replace(3, 11, ―‖);

// Big sb.reverse(); // gib

10

Object & Classes

Java is an Object-Oriented Language. As a language that has the Object Oriented feature, Java supports the following fundamental concepts:

- Polymorphism
- Inheritance
- Encapsulation
- Abstraction
- Classes
- Objects
- Instance
- Method
- Message Parsing

In this chapter, we will look into the concepts Classes and Objects.

- **Object -** Objects have states and behaviors. Example: A dog has states- color, name, breed as well as behaviors -wagging, barking, eating. An object is an instance of a class.
- **Class -** A class can be defined as a template/blue print that describes the behaviors/states that object of its type support.

Objects in Java:

Let us now look deep into what are objects. If we consider the real-world we can find many objects around us, Cars, Dogs, Humans, etc. All these objects have a state and behavior. If we consider a dog, then its state is - name, breed, color, and the behavior is - barking, wagging, running If you compare the software object with a real world object, they have very similar characteristics. Software objects also have a state and behavior. A software object's state is stored in fields and behavior is shown via methods. So in software development, methods operate on the internal state of an object and the object- to-object communication is done via methods.

Classes in Java:

A class is a blue print from which individual objects are created. A sample of a class is given below:

```
public class Dog{
 String breed;
int  age;
String color;
void  barking(){
}
void hungry(){
 }
void sleeping(){
   }
}
```

A class can contain any of the following variable types.

- **Local variables:** Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.

- **Instance variables:** Instance variables are variables within a class but outside any method. These variables are instantiated when the class is loaded. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
- **Class variables:** Class variables are variables declared within a class, outside any method, with the static keyword.

A class can have any number of methods to access the value of various kinds of methods. In the above example, barking(), hungry() and sleeping() are methods. Below mentioned are some of the important topics that need to be discussed when looking into classes of the JavaLanguage.

Constructors:

When discussing about classes, one of the most important subtopic would be constructors. Every class has a constructor. If we do not explicitly write a constructor for a class the Java compiler builds a default constructor for that class. Each time a new object is created, at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than oneconstructor.

Example of a constructor is given below:

```
public class Puppy{
public Puppy(){
}
public Puppy(String name){
 // This constructor has one parameter, name.
   }
}
```

Creating an Object:

As mentioned previously, a class provides the blueprints for objects. So basically an object is created from a class. In Java the new keyword is used to create new objects. There are three steps when creating an object from a class:

- **Declaration:** A variable declaration with a variable name with an object type.
- **Instantiation:** The 'new' keyword is used to create the object.
- **Initialization:** The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Example of creating an object is given below:

```
public class Puppy{
public Puppy(String name){
// This constructor has one parameter, name.
System.out.println("Passed Name is :"+ name );
 }
 public static void main(String[]args){
// Following statement would create an object myPuppy
Puppy myPuppy =new Puppy("tommy");
   }
}
```

If we compile and run the above program, then it would produce the following result:

```
Passed Name is:  tommy
```

AccessingInstance Variables and Methods:

Instance variables and methods are accessed via created objects. To access an instance variable the fully qualified path should be as follows:

```
/* First create an object */ ObjectReference=
newConstructor();
/* Now call a variable as follows */
ObjectReference.variableName;
/* Now you can call a class method as follows */
ObjectReference.MethodName();
```

Example:

This example explains how to access instance variables and methods of a class:

```
public class Puppy{
int puppyAge;
public Puppy(String name){
// This constructor has one parameter, name.
System.out.println("Passed Name is :"+ name );
}
public void setAge(int age ){
puppyAge = age;
 }
public int getAge(){
 System.out.println("Puppy's age is :"+ puppyAge );
 return puppyAge;
}
public static void main(String[]args){
/* Object creation  */
Puppy myPuppy =newPuppy("tommy");
/* Call class method to set puppy's age */
myPuppy.setAge(2);
/* Call another class method to get puppy's age */
myPuppy.getAge();
 /* You can access instance variable as follows as well */
 System.out.println("Variable Value :"+ myPuppy.puppyAge );
     }
 }
```

If we compile and run the above program, then it would produce the following result:

```
PassedName is:tommy
Puppy's age is :2
Variable Value :2
```

Source file declaration rules:

As the last part of this section, let‘s now look into the source file declaration rules. These rules are essential when declaring classes, *import* statements and *package* statements in a source file.

- There can be only one public class per source file.
- A source file can have multiple non public classes.
- The public class name should be the name of the source file as well which should be appended by **.java** at the end. For example : The class name is . *public class Employee{}* Then the source file should be as Employee.java.

- If the class is defined inside a package, then the package statement should be the first statement in the source file.
- If import statements are present then they must be written between the package statement and the class declaration. If there are no package statements then the import statement should be the first line in the source file.
- Import and package statements will imply to all the classes present in the source file. It is not possible to declare different import and/or package statements to different classes in the source file.

Classes have several access levels and there are different types of classes; abstract classes, final classes, etc. I will be explaining about all these in the access modifiers chapter.

Apart from the above mentioned types of classes, Java also has some special classes called Inner classes and Anonymous classes.

Java Package:

In simple, it is a way of categorizing the classes and interfaces. When developing applications in Java, hundreds of classes and interfaces will be written, therefore categorizing these classes is a must as well as makes life much easier.

Import statements:

In Java if a fully qualified name, which includes the package and the class name, is given, then the compiler can easily locate the source code or classes. Import statement is a way of giving the proper location for the compiler to find that particular class.

For example, the following line would ask compiler to load all the classes available in directory java_installation/java/io

```
import java.io.*;
```

A Simple Case Study:

For our case study, we will be creating two classes. They are Employee and EmployeeTest.

First open notepad and add the following code. Remember this is the Employee class and the class is a public class. Now, save this source file with the name Employee.java.

The Employee class has four instance variables name, age, designation and salary. The class has one explicitly defined constructor, which takes a parameter.

```
import java.io.*;
public class Employee{
String name;
int age;
String designation;
double salary;
// This is the constructor of the class Employee
public Employee(String name){
this.name = name;
}
// Assign the age of the Employee to the variable age.
public void empAge(int empAge){
age = empAge;
}
/* Assign the designation to the variable designation.*/
public void empDesignation(String empDesig){
designation = empDesig;
}
/* Assign the salary to the variable salary.*/
public void empSalary(double empSalary){
salary = empSalary;
 }
 /* Print the Employee details */
public void printEmployee(){
System.out.println("Name:"+ name );
System.out.println("Age:"+ age );
System.out.println("Designation:"+ designation );
System.out.println("Salary:"+ salary);
      }
 }
```

As mentioned previously in this notes, processing starts from the main method. Therefore in-order for us to run this Employee class there should be main method and objects should be created. We will be creating a separate class for these tasks.

Given below is the *EmployeeTest* class, which creates two instances of the class Employee and invokes the methods for each object to assign values for each variable.

Save the following code in EmployeeTest.java file

```
import java.io.*;
publicclassEmployeeTest{
publicstaticvoid main(String args[]){
/* Create two objects using constructor */
Employee empOne =newEmployee("balu");
Employee empTwo =newEmployee("swapna ");
// Invoking methods for each object created
empOne.empAge(23);
empOne.empDesignation("Senior Software Engineer");
empOne.empSalary(100000);
empOne.printEmployee();
empTwo.empAge(25);
empTwo.empDesignation("Software Engineer");
empTwo.empSalary(50000);
empTwo.printEmployee();
   }
}
```

Now, compile both the classes and then run *EmployeeTest* to see the result as follows:

```
C :> javac Employee.java
C :> vi  EmployeeTest.java
C :> javac EmployeeTest.java
C :> java EmployeeTest
Name:swapna
```

```
Age:23
Designation:SoftwareEngineer
Salary:100000.0
Name:balu
Age:25
Designation:SoftwareEngineer
Salary:50000.0
```

11

Methods

A Java method is a collection of statements that are grouped together to perform an operation. When you call the System.out.println method, for example, the system actually executes several statements in order to display a message on the console. Now you will learn how to create your own methods with or without return values, invoke a method with or without parameters, overload methods using the same names, and apply method abstraction in the program design.

Creating Method:

Considering the following example to explain the syntax of a method:

public static int funcName(int a, int b)

{

// body

}

Here,

- **public static** : modifier.
- **int**: return type
- **funcName**: function name
- **a, b**: formalparameters
- **int a, int b**: list of parameters

Methods are also known as Procedures or Functions:

- **Procedures:** They don't return any value.
- **Functions:** They returnvalue.

Method definition consists of a method header and a method body. The same is shown below:

modifier returnType nameOfMethod (Parameter List)
{
*// **method body***
}

The syntax shown above includes:

- **modifier:** It defines the access type of the method and it is optional to use.
- **returnType:** Method may return a value.
- **nameOfMethod:** This is the method name. The method signature consists of the method name and the parameterlist.
- **Parameter List:** The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
- **method body:** The method body defines what the method does with statements.

Example:

Here is the source code of the above defined method called max(). This method takes two parameters num1 and num2 and returns the maximum between the two:

```
/** the snippet returns the minimum between two numbers */
public static int minFunction(int n1, int n2) {
 int min;
if (n1 > n2) min = n2;
else min  =  n1;
return min;
}
```

Method Calling:

For using a method, it should be called. There are two ways in which a method is called i.e. method returns a value or returning nothing (no return value). The process of method calling is simple. When a program invokes a method, the program control gets transferred to the called method. This called method then returns control to the caller in two conditions, when:

- return statement is executed.
- reaches the method ending closing brace.

The methods returning void is considered as call to a statement. Lets consider an example:

```
System.out.println("This is tutorialspoint.com!");
```

The method returning value can be understood by the following example:

```
int result = sum(6, 9);
```

Example:

Following is the example to demonstrate how to define a method and how to call it:

```
public class ExampleMinNumber{

/** returns the minimum of two numbers */
 public static int minFunction(int n1, int n2) {
 int min;
 if (n1 > n2)
min = n2;
else min = n1;
return min;
 }

public static void main(String[] args) {
int a = 11;
int b =  6;
int c = minFunction(a, b);
 System.out.println("Minimum Value = " + c);
   }
}
```

This would produce the following result:

```
Minimum value =  6
```

The void Keyword:

The void keyword allows us to create methods which do not return a value. Here, in the following example we're considering a void method *methodRankPoints*. This method is a void method which does not return any value. Call to a void method must be a statement i.e. *methodRankPoints(255.7);*. It is a Java statement which ends with a semicolon as shown below.

Example:

```
public class ExampleVoid {
 public static void main(String[] args) {
methodRankPoints(255.7);
 }
public static void methodRankPoints(double points) {
 if (points >=  202.5)
 { System.out.println("Rank:A1");
 }
else if (points >= 122.4) {
System.out.println("Rank:A2");
 } else
 {
System.out.println("Rank:A3");
      }
  }
}
```

This would produce the following result:

```
Rank:A1
```

Passing Parameters by Value:

While working under calling process, arguments is to be passed. These should be in the same order as their respective parameters in the method specification. Parameters can be passed by value or by reference. Passing Parameters by Value means calling a method with a parameter. Through this the argument value is passed to the parameter.

Example:

The following program shows an example of passing parameter by value. The values of the arguments remains the same even after the method invocation.

```
public class swappingExample  {
public static void swapFunction(int a, int b) {
System.out.println("Before swapping(Inside), a = " + a + " b = " + b);
 // Swap n1 with n2
 int c =  a;
a = b;
 b = c;
System.out.println("After swapping(Inside), a = " + a + " b = " + b);
  }
public static void main(String[] args) {
 int a =  30;
int b =  45;
System.out.println("Before swapping, a = " + a + " and b = " + b);
// Invoke the swap method
swapFunction(a, b);
System.out.println("\n**Now, Before and After swapping values will be
same here**:");
```

```
 System.out.println("After swapping, a = " + a + " and b is " + b);
     }
 }
```

This would produce the following result:

```
Before
swapping, a = 30 and b = 45 Before
swapping(Inside), a = 30 b = 45 After
swapping(Inside), a = 45 b =  30
**Now, Before and After swapping values will be same here**:
After swapping, a = 30 and b is 45
```

Method Overloading:

When a class has two or more methods by same name but different parameters, it is known as method overloading. It is different from overriding. In overriding a method has same method name, type, number of parameters etc. if number of parameters are same, then type of parameters should be different.

Lets consider the example shown before for finding minimum numbers of integer type. If, lets say we want to find minimum number of double type. Then the concept of Overloading will be introduced to create two or more methods with the same name but different parameters.

The below example explains the same:

```
public class ExampleOverloading{
// for integer
 public static int minFunction(int n1, int n2) {
 int min;
if (n1 > n2) min = n2;
else min = n1;
return min;
}
// for double
public static double minFunction(double n1, double n2) {
double min;
 if (n1 > n2) min = n2;
else min = n1;
return min;
   }
public static void main(String[] args) {
 int a = 11;
int b = 6;
double c = 7.3;
double d = 9.4;
int result1 = minFunction(a, b);
// same function name with different parameters
double result2 = minFunction(c, d);
System.out.println("Minimum Value = " + result1);
System.out.println("Minimum Value = " + result2);
   }
}
```

This would produce the following result:

```
Minimum Value = 6
Minimum Value = 7.3
```

Overloading methods makes program readable. Here, two methods are given same name but with different parameters.

The minimum number from integer and double types is the result.

Using Command-Line Arguments:

Sometimes you will want to pass information into a program when you run it. This is accomplished by passing command-line arguments to main( ). A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy. they are stored as strings in the String array passed to main( ).

Example:

The following program displays all of the command-line arguments that it is called with:

```
public class CommandLine{
 public static void main(String args[]){
 for(int i=0; i<args.length; i++){
 System.out.println("args["+ i +"]: "+args[i]);
      }
   }
 }
```

Try executing this program as shown here:

```
java CommandLine this is a command line 200-100
```

This would produce the following result:

```
args[0]:this
args[1]:is
args[2]: a
args[3]: command
```

```
args[4]: line
args[5]:200
args[6]:-100
```

The Constructors:

A constructor initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type. Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other startup procedures required to create a fully formed object. All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

Example:

Here is a simple example that uses a constructor:

```
// A simple constructor.
class MyClass{
int x;
// Following is the constructor
MyClass(){
 x =10;
    } }
```

You would call constructor to initialize objects as follows:

```
public class ConsDemo{
public static void main(String args[]){
MyClass t1 =new MyClass();
MyClass t2 =new MyClass();
System.out.println(t1.x +" "+ t2.x);
   }
}
```

Most often, you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method, just declare them inside the parentheses after the constructor's name.

Example:

Here is a simple example that uses a constructor:

```
// A simple constructor.
class MyClass{
 int x;
// Following is the constructor
MyClass(int i ){
 x = i;
    }
 }
```

You would call constructor to initialize objects as follows:

```
public class ConsDemo{
 public static void main(String args[]){
MyClass t1 =new MyClass(10);
MyClass t2 =new MyClass(20);
System.out.println(t1.x +" "+ t2.x);
    }
 }
```

This would produce the following result:

```
10 20
```

Variable Arguments(var-args):

JDK 1.5 enables you to pass a variable number of arguments of the same type to a method. The parameter in the method is declared as follows:
typeName... parameterName

In the method declaration, you specify the type followed by an ellipsis (...) Only one variable-length parameter may be specified in a method, and this parameter must be the last parameter. Any regular parameters must precede it.

Example:

```
public class VarargsDemo{
public static void main(String args[]){
// Call method with variable args
printMax(34,3,3,2,56.5);
printMax(new double[]{1,2,3});
 }
public static void printMax(double... numbers){
 if(numbers.length ==0){
System.out.println("No argument passed");
return;
 }
double result =  numbers[0];
for(int i =1; i < numbers.length; i++)
if(numbers[i]> result) result = numbers[i];
System.out.println("The max value is "+ result);
   }   }
```

This would produce the following result:

```
The max value is 56.5
The max value is 3.0
```

Garbage Collection

Since objects are dynamically allocated by using the new operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator. Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called garbage collection. It works like this: when no references to an object to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++. Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used. Furthermore, different java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs

The finalize () Method

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-java resource such as a file handle or window character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, java provides a mechanism called finalization. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

To add a finalizer to a class, you simply define he **finalize()** method. The java run time calls that method whenever it is about to recycle an object of that class. Inside the **finalize()** method you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the java run time calls the **finalize()**method on the object.

The **finalize()** method has this general form:

```
protected void finalize()
{
// finalization codehere
}
```

Here, the keyword **protected** is a specifier that prevents access to **finalize()** by code defined outside its class.

It is important to understand that finalize() is only called just prior to garbage collection. It is not called when an object goes out-of-scope, for example. This means program should provide other means of releasing system resources, etc., used by the object. It must not rely on **finalize()** for normal program operation.

Example:

```
/** Example shows garbage collector in action Note that the finalize() method of object GC1 runs
without being specifically called and that the id's of garbage collected objects are not always
sequential.
*/



class TestGC {

public static void main(String[] args) {

Runtime rt = Runtime.getRuntime();

System.out.println("Available Free Memory: " + rt.freeMemory());

for(int i=0; i<10000; i++ ) { GC1 x
= new GC1(i);
}

System.out.println("Free Memory before call to gc(): " + rt.freeMemory());
System.runFinalization();
System.gc();
System.out.println(" Free Memory after call to gc(): " + rt.freeMemory());

}
}
```

```
class GC1 {

String str;
int id;

GC1(int i) {
this.str = new String("abcdefghijklmnopqrstuvwxyz"); this.id = i;
}

protected void finalize() {
System.out.println("GC1 object " + id + " has been finalized.");
}

}
```

Some Example programs:

```
class Current_Bill3{
String connection_no;
double previous_reading;
double present_reading;
double bill_amount=0;
double billing_units=0;

Current_Bill3(){
connection_no="123";
previous_reading=300;
```

```
present_reading=450;
}

Current_Bill3(String str, double prv, double prs){
connection_no=str;
previous_reading=prv;
present_reading=prs;
}

void compute_bill(){

billing_units=present_reading-previous_reading;
if (billing_units<=100)
    bill_amount=billing_units*5;
else if (billing_units>100 && billing_units<=500)
    bill_amount=billing_units*7;
else if (billing_units>500)
    bill_amount=billing_units*10;

}

public static void main(String args[]){
double total_amt;
Current_Bill3 b1= new Current_Bill3("125",100,200);
Current_Bill3 b2= new  Current_Bill3("124",500,700);

b1.compute_bill();

b2.compute_bill();
total_amt=b1.bill_amount+b2.bill_amount;
```

```
System.out.println("first connection charges=Rs."+b1.bill_amount);
System.out.println("second connection charges=Rs."+b2.bill_amount);
System.out.print("total charges for 2 connections =");
System.out.println(total_amt );

  }
}
```

Some Example programs:

This program updates opening balance after account holders either deposit or withdraw, minimum transaction amount should be rs.100.

```
import java.util.Scanner;
class Account{
int acc_no;
double open_bal;
// constructor
Account(){
acc_no= 222;
open_bal=1000.00;
}
// overloaded constructor
Account(int n, double d){
acc_no=n;
open_bal=d;
}
// deposit  method
void deposit(double d){
System.out.println("before deposit yr o.b  ="+open_bal);
```

```
open_bal=open_bal+d;

System.out.println("after deposit yr o.b  ="+open_bal);


}
//  deposit  method    for minimum transaction amount.
void deposit(){

System.out.println("before deposit yr o.b ="+open_bal);
open_bal=open_bal+100;

System.out.println("after deposit yr o.b  ="+open_bal);
}
// withdraw method
void withdraw( double w) {
System.out.println("before withdraw yr o.b ="+open_bal);

open_bal=open_bal-w;
System.out.println("after withdraw yr o.b  ="+open_bal);


}
// withdraw method for minimum transaction amount.
void withdraw() {
System.out.println("before withdraw yr o.b ="+ open_bal);

open_bal=open_bal-100;
System.out.println("after withdraw yr o.b  ="+open_bal);


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

```
Scanner in= new Scanner(System.in);
double amt;
Account tr1= new  Account();

Account tr2= new Account(199,5000.00);
System.out.println("enter amount to deposit(min rs.100)");
amt=in.nextDouble();
tr1.deposit(amt);
System.out.println("enter amount to withdraw (min rs.100)");
amt=in.nextDouble();
tr1.withdraw(amt);

System.out.println("enter amount to deposit(min rs.100)");
amt=in.nextDouble();

if (amt<100)
tr2.deposit();
else
tr2.deposit(amt);
System.out.println("enter amount to withdraw (min rs.100)");
amt=in.nextDouble();
if (amt<100)
tr2.withdraw();
else
tr2.withdraw(amt);
 }
}
```

Recursion

Java supports *recursion*. Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be *recursive*. The classic example of recursion is the computation of the factorial of a number. The factorial of a number *N* is the product of all the whole numbers between 1 and *N*. For example, 3 factorial is 1 × 2 × 3, or 6. Here is how a factorial can be computed by use of a recursive method:

Example

```
class Factorial {
// this is a recursive method
int fact(int n)  {
int result;
if(n==1) return 1;
result = fact(n-1) * n;
return result;
}
}
class Recursion {
public static void main(String args[]) {
Factorial f = new  Factorial();
System.out.println("Factorial of 3 is " + f.fact(3));
System.out.println("Factorial of 4 is " + f.fact(4));
System.out.println("Factorial of 5 is " + f.fact(5));
}
}
```

The output :

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

Example:

```
class RecTest {
int values[];
RecTest(int i) {
values = new  int[i];
}
// display array -- recursively
void printArray(int i) {
if(i==0) return;
else printArray(i-1);
System.out.println("[" + (i-1) + "] " + values[i-1]);
}
}
class Recursion2 {
public static void main(String args[]) {
RecTest ob = new RecTest(10);
int i;
for(i=0; i<10; i++) ob.values[i] = i;
ob.printArray(10);
}
}
```

This program generates the following output:

```
[0] 0
[1] 1
[2] 2
[3] 3
[4] 4
[5] 5
[6] 6
[7] 7
[8] 8
[9] 9
```

12

Modifier Types

Modifiers arekeywords that you add to those definitions to change their meanings. The Java language has a wide variety of modifiers, including the following:

1. Java Access Modifiers

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are:

- Visible to the package, the default. No modifiers are needed.
- Visible to the class only (private).
- Visible to the world (public).
- Visible to the package and all subclasses(protected).

Default Access Modifier - Nokeyword:

Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc. A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public static final and the methods in an interface are by default public.

Example:

Variables and methods can be declared without any modifiers, as in the following examples:

```
String version ="1.5.1";
 boolean processOrder()
{
 return true;
 }
```

Private Access Modifier - private:

Methods, Variables and Constructors that are declared private can only be accessed within the declared class itself. Private access modifier is the most restrictive access level. Class and interfaces cannot be private. Variables that are declared private can be accessed outside the class if public getter methods are present in the class. Using the private modifier is the main way that an object encapsulates itself and hide data from the outside world.

Example:

The following class uses private access control:

```
public class Logger{
 private String format;
 public String getFormat(){
 return  this.format;
 }
public void setFormat(String  format)
{
```

```
this.format = format;
   }
}
```

Here, the *format* variable of the Logger class is private, so there's no way for other classes to retrieve or set its value directly. So to make this variable available to the outside world, we defined two public methods: *getFormat()*, which returns the value of format, and *setFormat(String)*, which sets its value.

Public Access Modifier - public:

A class, method, constructor, interface etc declared public can be accessed from any other class. Therefore fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe. However if the public class we are trying to access is in a different package, then the public class still need to be imported. Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.

Example:

The following function uses public access control:

```
public static void main(String[] arguments){
 // ...
}
```

The main() method of an application has to be public. Otherwise, it could not be called by a Java interpreter (such as java) to run the class.

Protected Access Modifier -protected:

Variables, methods and constructors which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.

The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected. Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.

Example:

The following parent class uses protected access control, to allow its child class **override*openSpeaker()* method:**

```
class AudioPlayer{
protected boolean openSpeaker(Speaker  sp){
// implementation details
      }
  }
 class StreamingAudioPlayer{
boolean openSpeaker(Speaker sp){
// implementation details
     }
 }
```

Here, if we define *openSpeaker()* method as private, then it would not be accessible from any other class other than *AudioPlayer*. If we define it as public, then it would become accessible to all the outside world. But our intension is to expose this method to its subclass only, thats why we used *protected* modifier.

Access Control and Inheritance:

The following rules for inherited methods are enforced:

- Methods declared public in a superclass also must be public in all subclasses.
- Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private.
- Methods declared without access control (no modifier was used) can be declared more private in subclasses.
- Methods declared private are not inherited at all, so there is no rule for them.

Non Access Modifiers

To use a modifier, you include its keyword in the definition of a class, method, or variable. The modifier precedes the rest of the statement, as in the following examples (Italic ones):

public class className {

// ...

}

private boolean myFlag;

static final double weeks =9.5;

protected static final int BOXWIDTH =42;

public static void main(String[] arguments)

{

// body of method

}

Access Control Modifiers:

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are:

- Visible to the package. the default. No modifiers are needed.
- Visible to the class only (private).
- Visible to the world (public).
- Visible to the package and all subclasses(protected).

Non Access Modifiers:

Java provides a number of non-access modifiers to achieve many other functionality.

- The *static* modifier for creating class methods and variables
- The *final* modifier for finalizing the implementations of classes, methods, and variables.
- The *abstract* modifier for creating abstract classes and methods.
- The *synchronized* and *volatile* modifiers, which are used for threads.

To use a modifier, you include its keyword in the definition of a class, method, or variable. The modifier precedes the rest of the statement, as in the following examples (Italic ones):

```
Public class className {
// ...
 }
private boolean myFlag;
 static final double weeks =9.5;
protected static final int BOXWIDTH =42;
 public static void main(String[] arguments){
// body of method
 }
```

Access Control Modifiers:

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are:

- Visible to the package. the default. No modifiers are needed.
- Visible to the class only (private).
- Visible to the world (public).
- Visible to the package and all subclasses(protected).

Non Access Modifiers:

Java provides a number of non-access modifiers to achieve many other functionality.

- The *static* modifier for creating class methods and variables
- The *final* modifier for finalizing the implementations of classes, methods, and variables.
- The *abstract* modifier for creating abstract classes and methods.
- The *synchronized* and *volatile* modifiers, which are used for threads.

Understanding static

There will be times when you will want to define a class member that will be used independently of any object of that class. Normally, a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword static. When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object. You can declare both methods and variables to be static. The most common example of a static member is main( ). main( ) is declared as static because it must be called before any objects exist. Instance variables declared as static are, essentially, global variables. When objects of its class are declared, no copy of a static variable is made. Instead, all instances of the class share the same static variable.

Methods declared as **static** have several restrictions:

• They can only call other **static** methods.

• They must only access **static** data.

• They cannot refer to **this** or **super** in any way. (The keyword **super** relates to inheritance and is described in the next chapter.)

If you need to do computation in order to initialize your **static** variables, you can declare a **static** block that gets executed exactly once, when the class is first loaded. The following example shows a class that has a **static** method, some **static** variables, and a **static** initialization block:

Example

```
class UseStatic {
static int a = 3;
static int b;
static void meth(int x) {
System.out.println("x = " + x);
System.out.println("a = " + a);
System.out.println("b = " + b);
}
static {
System.out.println("Static block initialized.");
b = a * 4;
}
public static void main(String args[]) {
meth(42);
  }
```

}

As soon as the **UseStatic** class is loaded, all of the **static** statements are run. First, **a** is set to **3**, then the **static** block executes, which prints a message and then initializes **b** to **a** * **4** or **12**. Then **main( )** is called, which calls **meth( )**, passing **42** to **x**. The three **println( )** statements refer to the two **static** variables **a** and **b**, as well as to the local variable **x**.

Here is the output of the program: Static block initialized.

```
x = 42
a = 3
b = 12
```

Outside of the class in which they are defined, **static** methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by the dot operator. For example, if you wish to call a **static** method from outside its class, you can do so using the following general form:

classname.method( ).

A **static** variable can be accessed in the same way—by use of the dot operator on the name of the class. This is how Java implements a controlled versionof global methods and global variables.

Example

```
class StaticDemo {
static int a = 42;
static int b = 99;
static void callme() {
System.out.println("a = " + a);
}
}
class StaticByName {
public static void main(String args[]) {
StaticDemo.callme();
System.out.println("b = " + StaticDemo.b);
   }
}
```

This would produce the following result:

```
a = 42
b = 99
```

Introducing final key word :

A variable can be declared as **final**. Doing so prevents its contents from being modified. A final method prevents overriding and a final class prevents inheritance, later two are discussed in the following chapters.
This means that you must initialize a **final** variable when it is declared. For

Example:

```
final int FILE_NEW = 1; final
int FILE_OPEN = 2; final int
FILE_SAVE = 3; final int
FILE_SAVEAS = 4; final int
FILE_QUIT = 5;
```

Subsequent parts of your program can now use **FILE_OPEN**, etc., as if they were constants, without fear that a value has been changed.

final - keyword is also used with methods and classes, these two topics are discussed in chapter: **16 (java abstraction)**

13

Inner Classes

Why Use Nested Classes?

There are several compelling reasons for using nested classes, among them:

· It is a way of logically grouping classes that are only used in one place.

· It increases encapsulation.

· Nested classes can lead to more readable and maintainable code.

Logical grouping of classes—If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package morestreamlined.

Increased encapsulation—Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.

More readable, maintainable code—Nesting small classes within top-level classes places the code closer to where it is used.

It is possible to define a class within another class; such classes are known as *nested classes*. The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B does not exist independently of A. A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class. A nested class that is declared directly within its enclosing class scope is a member of its enclosing class. It is also possible to declare a nested class that is local to a block.

There are two types of nested classes: *static* and *non-static*. A static nested class is one that has the **static** modifier applied. Because it is static, it must access the members of its enclosing class through an object. That is, it cannot refer to members of its enclosing class directly. Because of this restriction, static nested classes are seldom used. The most important type of nested class is the *inner* class. An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

The following program illustrates how to define and use an inner class. The class named **Outer** has one instance variable named **outer_x**, one instance method named **test( )**, and defines one inner class called **Inner**.

```
// Demonstrate an inner class.
class Outer {
int outer_x = 100;
void test() {
Inner inner = new Inner();
inner.display();
}
// this is an inner class
class Inner {
void display() {
System.out.println("display: outer_x = " + outer_x);
}
}
}
```

```
class InnerClassDemo {
public static void main(String args[]) {
Outer outer = new Outer();
outer.test();
}
}
```

Output from this application is shown here:

```
display: outer_x =  100
```

In the above program, an inner class named **Inner** is defined within the scope of class **Outer**. Therefore, any code in class **Inner** can directly access the variable **outer_x**. An instance method named **display( )** is defined inside **Inner**. This method displays **outer_x** on the standard output stream. The **main( )** method of **InnerClassDemo** creates an instance of class **Outer** and invokes its **test( )** method. That method creates an instance of class **Inner** and the **display( )** method is called.

```
class OuterClass
{
        private int i = 9;

        public void createInner()
       {
        InnerClass i1 = new InnerClass();
        i1.getValue();
        }
```

```
          class InnerClass
         {
                    public void getValue()
          {
                  System.out.println("value of i =" + i);
                    }
          }
}

class InnerClassDemo2
{
          public static void main(String[] args)
          {
          OuterClass otr = new OuterClass();

          OuterClass.InnerClass inr = otr.new InnerClass();

          inr.getValue();

         otr.createInner();
          }
}
```

Output from this application is shown here:

```
value of i =9
value of i =9
```

In this program the inner class object also can be created and used to call inner class variables and methods.

14

Inheritance

Inheritance can be defined as the process where one object acquires the properties of another. With the use of inheritance, the information is made manageable in a hierarchical order. When we talk about inheritance, the most commonly used keyword would be **extends** and **implements**. These words would determine whether one object IS-A type of another. By using these keywords we can make one object acquire the properties of another object.

Inheritance in java is a mechanism in which one object acquires all the properties and behaviors of parent object.

The idea behind inheritance in java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of parent class, and you can add new methods and fields also.

IS-A Relationship:

IS-A is a way of saying : This object is a type of that object. Let us see how the **extends** keyword is used to achieve inheritance.

Inheritance represents the **IS-A relationship**, also known as *parent-child* relationship.

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
   //methods and fields
}
```

The **extends keyword** indicates that you are making a new class that derives from an existing class.

In the terminology of Java, a class that is inherited is called a super class. The new class is called a subclass.

```
class Employee{
float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary); System.out.println("Bonus of Programmer
   is:"+p.bonus);
}
}
```

Output:

```
Programmer salary is:40000.0
```

```
Bonus of programmer is:10000
```

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

Types of inheritance in java

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.

```
public class Animal{
}
public class Mammal extends Animal{
}
 public class Reptile extends Animal{
 }
public class Dog extends Mammal{
}
```

Now, based on the above example, In Object Oriented terms the following are true:

- Animal is the superclass of Mammal class.
- Animal is the superclass of Reptile class.
- Mammal and Reptile are subclasses of Animal class.
- Dog is the subclass of both Mammal and Animal classes.

Now, if we consider the IS-A relationship, we can say:

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

With use of the extends keyword the subclasses will be able to inherit all the properties of the superclass except for the private properties of the superclass. We can assure that Mammal is actually an Animal with the use of the instance operator.

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);
}
}
```

This would produce the following result:

```
true
true
true
```

Since we have a good understanding of the **extends** keyword, let us look into how the **implements**keyword is used to get the IS-A relationship.

The **implements** keyword is used by classes by inherit from interfaces. Interfaces can never be extended by theclasses.

Example:

```
public interface Animal{
}
 public class Mammal implements Animal{
}
public class Dog extends Mammal{
 }
```

The instanceof Keyword:

Let us use the **instanceof** operator to check determine whether Mammal is actually an Animal, and dog is actually an Animal

```
interfaceAnimal{
}
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);
 }
}
```

This would produce the following result:

```
true
true
true
```

HAS-A relationship:

These relationships are mainly based on the usage. This determines whether a certain class **HAS-A** certain thing. This relationship helps to reduce duplication of code as well as bugs.

Lets us look into an example: public
class Vehicle{
}
public class Speed{
}
public class Van extends Vehicle{ private Speed
sp;
}

This shows that class Van HAS-A Speed. By having a separate class for Speed, we do not have to put the entire code that belongs to speed inside the Van class which makes it possible to reuse the Speed class in multiple applications. In Object-Oriented feature, the users do not need to bother about which object is doing the real work. To achieve this, the Van class hides the implementation details from the users of the Van class. So basically what happens is the users would ask the Van class to do a certain action and the Van class will either do the work by itself or ask another class to perform the action. A very important fact to remember is that Java only supports only single inheritance. This means that a class cannot extend more than one class. Therefore following is illegal:

```
public class extends Animal, Mammal{
}
```

However, a class can implement one or more interfaces. This has made Java get rid of the impossibility of multiple inheritance.

Has-A-Relationship (Aggregation in Java)

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

Consider a situation, Employee object contains many informations such as id, name, emailId etc. It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc. as given below.

```
class Employee{
int id;
String name;
Address address;//Address is a class
...
}
```

In such case, Employee has an entity reference address, so relationship is Employee HAS-A address.

Why use Aggregation?

- For Code Reusability.

In this example, we have created the reference of Operation class in the Circle class.

```
class Operation{
 int square(int n){
  return n*n;
 }
}

class Circle{
 Operation op;//aggregation
double pi=3.14;

 double area(int radius){ op=new
   Operation();
   int rsquare=op.square(radius);//code reusability (i.e. delegates the method c all).
   return pi*rsquare;
 }



 public static void main(String args[]){ Circle
   c=new Circle();
   double result=c.area(5);
   System.out.println(result);
 }
}
```

Understanding meaningful example of Aggregation

In this example, Employee has an object of Address, address object contains its own informations such as city, state, country etc. In such case relationship is Employee HAS-A address.

Address.java

```
public class Address {
```

```
String city,state,country;

    public Address(String city, String state, String country) {
    this.city = city; this.state
    = state; this.country=
    country;
}

}
```

Emp.java

```
public class Emp {
int id;
String name;
Address address;

public Emp(int id, String name, Address address) {
    this.id = id; this.name =
    name;
    this.address=address;
}

void display(){ System.out.println(id+"
"+name);
System.out.println(address.city+" "+address.state+""+address.country);
}

public static void main(String[] args) {
Address address1=new Address("gzb","UP","india"); Address
address2=new Address("gno","UP","india");
Emp   e=new   Emp(111,"varun",address1);   Emp
e2=new Emp(112,"arun",address2); e.display();
e2.display();


}
}
```

```
Output:111 varun gzb
        UP india 112
        arun
          gno UP india
```

15

Overriding

In the previous chapter, we talked about superclasses and subclasses. If a class inherits a method from its superclass, then there is a chance to override the method provided that it is not marked final. The benefit of overriding is: ability to define a behavior that's specific to the subclass type which means a subclass can implement a parent class method based on its requirement. In object-oriented terms, overriding means to override the functionality of an existing method.

Why Overridden Methods?

As stated earlier, overridden methods allow Java to support run-time polymorphism. Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.

Example:

Let us look at an example.

```
Class Animal{
public void move(){
```

```
System.out.println("Animals can move");
}
}
class Dog extends Animal{
 public void move(){
System.out.println("Dogs can walk and run");
}
}
public class TestDog{
public static void main(String args[]){
Animal a =new Animal(); // Animal reference and object
Animal b =new Dog(); // Animal reference but Dog object
a.move();// runs the method in Animal class
b.move();//Runs the method in Dog class
 }
 }
```

This would produce the following result:

```
Animals can move
Dogs can walk and run
```

In the above example, you can see that the even though **b** is a type of Animal it runs the move method in the Dog class. The reason for this is: In compile time, the check is made on the reference type. However, in the runtime, JVM figures out the object type and would run the method that belongs to that particular object. Therefore, in the above example, the program will compile properly since Animal class has the method move. Then, at the runtime, it runs the method specific for that object. Consider the following example:

```
class Animal{ public void move(){
System.out.println("Animals can move");
 }
}
class Dog extendsAnimal{
 public void move(){
System.out.println("Dogs can walk and run");
 }
 public void bark(){
System.out.println("Dogs can bark");
}
}
public class TestDog{
public static void main(String args[]){
Animal a =new Animal();// Animal reference and object
Animal b =new Dog();// Animal reference but Dog object
a.move();// runs the method in Animal class
b.move();//Runs the method in Dog class
b.bark();
 }
}
```

This would produce the following result:

```
TestDog.java:30: cannot find symbol
symbol : method bark()
location:class Animal
b.bark();
^
```

This program will throw a compile time error since b's reference type Animal doesn't have a method by the name of bark.

Rules for method overriding:

- The argument list should be exactly the same as that of the overridden method.
- The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
-
- The access level cannot be more restrictive than the overridden method's access level. For example, if the superclass method is declared public, then the overriding method in the subclass cannot be either private or protected.
- Instance methods can be overridden only if they are inherited by the subclass.
- A method declared final cannot be overridden.
- A method declared static cannot be overridden but can be re-declared.
- If a method cannot be inherited, then it cannot be overridden.
- A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
- A subclass in a different package can only override the non-final methods declared public or protected.
- An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
- Constructors cannot be overridden. Using the super keyword:

When invoking a superclass version of an overridden method the **super** keyword is used.

```
class Animal{
 public void move(){
 System.out.println("Animals can move");
 }
} class Dog extends Animal{
 public void move(){
super.move();// invokes the super class method
System.out.println("Dogs can walk and run");
 }
}
public class TestDog{
public static void main(String args[]){
Animal b =new Dog();// Animal reference but Dog object
b.move();//Runs the method in Dog class
 }
}
```

This would produce the following result:

```
Animals can move
Dogs can walk and run
```

Dynamic Method Dispatch :

While the examples in the preceding section demonstrate the mechanics of method overriding, they do not show its power. Indeed, if there were nothing more to method overriding than a name space convention, then it would be, at best, an interesting curiosity, but of little real value. However, this is not the case. Method overriding forms the basis for one of Java‘s most powerful concepts: *dynamic method dispatch.* Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

Dynamic method dispatch is important because this is how Java implements run-time polymorphism.
Let‘s begin by restating an important principle: a superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. Here is how. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called.

In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed. Here is an example that illustrates dynamic method dispatch:

```
//Dynamic Method Dispatch
class A {
void callme() {
System.out.println("Inside A's callme  method");
}
}
```

```
class B extends A {
// override callme()
void callme() {
System.out.println("Inside B's callme  method");
}
}
class C extends A {
// override callme()
void callme() {
System.out.println("Inside C's callme  method");
}
}
class Dispatch {
public static void main(String args[]) {
A a = new A(); // object of type A
B b = new B(); // object of type B
C c = new C(); // object of type C
A r; // obtain a reference of type A
r = a; // r refers to an A object
        allme(); // calls A's version of
        callme r = b; // r refers to  a  B
        object r.callme(); // calls B's
        version of callme r = c; // r refers
        to  a  C  object r.callme(); // calls
        C's version of callme
}
}
```

The output from the program is shown here:

```
Inside A’s callme method
Inside B’s callme method
Inside C’s callme method
```

This program creates one superclass called **A** and two subclasses of it, called **B** and **C**. Subclasses **B** and **C** override **callme( )** declared in **A**. Inside the **main( )** method, objects of type **A**, **B**, and **C** are declared. Also, a reference of type **A**, called **r**, is declared. The program then in turn assigns a reference to each type of object to **r** and uses that reference to invoke **callme( )**. As the output shows, the version of **callme( )** executed is determined by the type of object being referred to at the time of the call. Had it been determined by the type of the reference variable, **r**, you would see three calls to **A**‘s **callme( )** method.

16

Polymorphism

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP, occurs when a parent class reference is used to refer to a child class object. Any Java object that can pass more than one IS-A test is considered to be polymorphic. In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object. It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed. The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object. A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

Example:

Let us look at an example.

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

Now, the Deer class is considered to be polymorphic since this has multiple inheritance.
Following are true for the above example:

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

When we apply the reference variable facts to a Deer object reference, the following declarations are legal:

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

All the reference variables d,a,v,o refer to the same Deer object in the heap.

Virtual Methods:

In this section, I will show you how the behavior of overridden methods in Java allows you to take advantage of polymorphism when designing your classes. We already have discussed method overriding, where a child class can override a method in its parent. An overridden method is essentially hidden in the parent class, and is not invoked unless the child class uses the super keyword within the overriding method.

```
/* 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;
}
publicString getName() {
return name;
}
public String getAddress() {
return address;
 }
public void setAddress(String newAddress) {
address = newAddress;
}
public int getNumber() {
 return number;
   }
 }
```

Now suppose we extend Employee class as follows:

```
/* File name : Salary.java */
public class Salaryextends 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, you study the following program carefully and try to determine its output:

/* File name : VirtualDemo.java */

```
public class VirtualDemo  {
public static void main(String[] args) {
Salarys =newSalary("MohdMohtashim","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 usingEmployee reference--");
e.mailCheck();
   }
}
```

Thiswouldproducethefollowingresult:

```
Constructing an Employee
Constructing an Employee
Call mailCheck using Salary reference –
Within mailCheck of Salary class
Mailing check to MohdMohtashim with salary 3600.0
Call mailCheck using Employee reference—
Within mailCheck of Salary class
Mailing check to JohnAdams with salary 2400.0
```

Here, we instantiate two Salary objects, one using a Salary reference s, and the other using an Employee reference e. While invoking *s.mailCheck()* the compiler sees mailCheck() in the Salary class at compile time, and the JVM invokes mailCheck() in the Salary class at run time. Invoking mailCheck() on e is quite different because e is an Employee reference. When the compiler sees*e.mailCheck()*, the compiler sees the mailCheck() method in the Employee class. Here, at compile time, the compiler used mailCheck() in Employee to validate this statement. At run time, however, the JVM invokes mailCheck() in the Salary class. This behavior is referred to as virtual method invocation, and the methods are referred to as virtual methods. All methods in Java behave in this manner, whereby an overridden method is invoked at run time, no matter what data type the reference is that was used in the source code at compile time.

17

Abstraction

Abstraction refers to the ability to make a class abstract in OOP. An abstract class is one that cannot be instantiated. All other functionality of the class still exists, and its fields, methods, and constructors are all accessed in the same manner. You just cannot create an instance of the abstract class. If a class is abstract and cannot be instantiated, the class does not have much use unless it is subclass. This is typically how abstract classes come about during the design phase. A parent class contains the common functionality of a collection of child classes, but the parent class itself is too abstract to be used on its own. **Abstract Class:**

Use the **abstract** keyword to declare a class abstract. The keyword appears in the class declaration somewhere before the class keyword.

Abstract Methods:

If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class asabstract.

The abstract keyword is also used to declare a method as abstract. An abstract method consists of a method signature, but no method body. Abstract method would have no definition, and its signature is followed by a semicolon, not curly braces as follows:

```
/* File name : Employee.java */
public abstract classEmployee {
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 double computePay() {
 System.out.println("Inside Employee computePay");
 return0.0;
 }
 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;
 }
}
```

Notice that nothing is different in this Employee class. The class is now abstract, but it still has three fields, seven methods, and one constructor. Now if you would try as follows:

```
/* File name : AbstractDemo.java */
 public class AbstractDemo {
public static void main(String[] args) {
/* Following is not allowed and would raise error */
Employee e =new Employee("George W.","Houston, TX",43);
 System.out.println("\n Call mailCheck usingEmployee reference--");
 e.mailCheck();
  }
}
```

When you would compile above class, then you would get the following error:

```
Employee.java:46:Employee is abstract; cannot be instantiated
Employee e =new Employee("George W.","Houston, TX",43);
^
1 error
```

Extending Abstract Class:

We can extend Employee class in normal way as follows:

```
/* 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;
 }
}
```

Here, we cannot instantiate a new Employee, but if we instantiate a new Salary object, the Salary object will inherit the three fields and seven methods from Employee.

```
/* File name : AbstractDemo.java */
 public class AbstractDemo {
public static void main(String[] args) {
 Salary s =new Salary("Mohd Mohtashim","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 usingEmployee reference--");
e.mailCheck();
```

```
 }
}
```

This would produce the following result:

```
Constructing an Employee
Constructing  an Employee
Call mailCheck using Salary reference –

Within mailCheck of Salary class
Mailing check to MohdMohtashim with salary 3600.0
Call mailCheck using Employee reference—
Within mailCheck of Salary class
Mailing check to JohnAdams with salary 2400.
```

Another example on abstract methods and classes.

```
public abstract class Employee {
private String name;
 private String address;
private int number;
public abstract double  computePay();
 //Remainder of class  definition
}
```

Declaring a method as abstract has two results:

- The class must also be declared abstract. If a class contains an abstract method, the class must be abstract as well.
- Any child class must either override the abstract method or declare itself abstract.

A child class that inherits an abstract method must override it. If they do not, they must be abstract and any of their children must override it. Eventually, a descendant class has to implement the abstract method; otherwise, you would have a hierarchy of abstract classes that cannot be instantiated. If Salary is extending Employee class, then it is required to implement computePay() method as follows:

```
/* File name : Salary.java */
 public class Salary extends Employee {
privatedouble salary;// Annual salary
public double computePay() {
System.out.println("Computing salary pay for "+ getName());
return  salary/52;
}
// Remainder of class  definition
 }
```

Here is a simple example of a class with an abstract method, followed by a class which implements that method:

```
//A Simple demonstration of abstract.
abstract class A {
abstract void callme();
// concrete methods are still allowed in abstract classes
void callmetoo() {
System.out.println("This is a concrete method.");
   }
}
class B extends A {
void callme() {
```

```
System.out.println("B's implementation of  callme.");
   }
}
class AbstractDemo {
public static void main(String args[]) {
B b = new B();
b.callme();
b.callmetoo();
  }
}
```

Notice that no objects of class A are declared in the program. As mentioned, it is not possible to instantiate an abstract class. One other point: class A implements a concrete method called callmetoo( ). This is perfectly acceptable. Abstract classes can include as much implementation as they see fit. Although abstract classes cannot be used to instantiate objects, they can be used to create.

Using an abstract class, you can improve the Figure class shown earlier. Since there is no meaningful concept of area for an undefined two-dimensional figure, the following version of the program declares area( ) as abstract inside Figure. This, of course, means that all classes derived from Figure must override area( ).

```
//Using abstract methods and classes.
abstract class Figure {
double dim1;
double dim2;
Figure(double a, double b) {
dim1 = a;
```

```
dim2 = b;
}
// area is now an abstract method
abstract double area();
}
class Rectangle extends Figure {
Rectangle(double a, double b) {
super(a, b);
}
// override area for rectangle
double area() {
System.out.println("Inside Area for Rectangle.");
return dim1 *  dim2;
}
}
class Triangle extends Figure {
Triangle(double a, double b) {
super(a, b);
}
// override area for right triangle
double area() {
System.out.println("Inside Area for Triangle.");
return dim1 * dim2 / 2;
}
}
class AbstractAreas {
public static void main(String args[]) {
// Figure f = new Figure(10, 10); // illegal now
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
```

```
Figure figref; // this is OK, no object is created
figref = r;
System.out.println("Area is " + figref.area());
figref = t;
System.out.println("Area is " + figref.area());
   }
}
```

As the comment inside main( ) indicates, it is no longer possible to declare objects of type Figure, since it is now abstract. And, all subclasses of Figure must override area( ). To prove this to yourself, try creating a subclass that does not override area( ). You will receive a compile-time error. Although it is not possible to create an object of type Figure, you can create a reference variable of type Figure. The variable figref is declared as a reference to Figure, which means that it can be used to refer to an object of any class derived from Figure. As explained, it is through superclass reference variables that overridden methods are resolved at run time.

final – key word from chapter no.11

final methods and final classes

Using final with Inheritance

The keyword **final** has three uses. First, it can be used to create the equivalent of a named constant. This use was described in the preceding chapter. The other two uses of **final** apply to inheritance. Both are examined here.

Using final to Prevent Overriding

While method overriding is one of Java‘s most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden. The following fragment illustrates **final**:

```
class A {
final void meth()  {
System.out.println("This is a final method.");
}
}
class B extends A {
void meth() { // ERROR! Can't override.
System.out.println("Illegal!");
}
}
```

Because **meth( )** is declared as **final**, it cannot be overridden in **B**. If you attempt to do so, a compile-time error will result. Methods declared as **final** can sometimes provide a performance enhancement: The compiler is free to *inline* calls to them because it ―knows‖ they will not be overridden by a subclass. When a small **final** method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call. Inlining is only an option with **final** methods. Normally, Java resolves calls to methods dynamically, at run time. This is called *late binding*. However, since **final** methods cannot be overridden, a call to one can be resolved at compile time. This is called *early binding*.

Using final to Prevent Inheritance

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with **final**. Declaring a class as **final** implicitly declares all of its methods as **final**, too. As you might expect, it is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

Here is an example of a **final** class:

```
final class A {
// ...
}
// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
// ...
}
```

As the comments imply, it is illegal for **B** to inherit **A** since **A** is declared as **final**.

18

Encapsulation

Encapsulation is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction. Encapsulation is the technique of making the fields in a class private and providing access to the fields via public methods. If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class. For this reason, encapsulation is also referred to as data hiding. Encapsulation can be described as a protective barrier that prevents the code and data being randomly accessed by other code defined outside the class. Access to the data and code is tightly controlled by an interface. The main benefit of encapsulation is the ability to modify our implemented code without breaking the code of others who use our code. With this feature Encapsulation gives maintainability, flexibility and extensibility to our code.

Example:

Let us look at an example that depicts encapsulation:

```
/* File name : EncapTest.java */
public class EncapTest{
private String name;
private String idNum;
private int  age;
public int getAge(){
```

```
return age;
}
Public String getName(){
return name;
}
Public String getIdNum(){
return idNum;
}
Public void setAge(int newAge){
age =  newAge;
 }
Public void setName(String newName){
name = newName;
}
public void setIdNum(String newId){
 idNum = newId;
  }
 }
```

The public methods are the access points to this class' fields from the outside java world. Normally, these methods are referred as getters and setters. Therefore any class that wants to access the variables should access them through these getters and setters. The variables of the EncapTest class can be access as below:

```
/* File name : RunEncap.java */
public class RunEncap{
public static void main(String args[]){
EncapTest encap =new EncapTest();
encap.setName("James");
encap.setAge(20);
encap.setIdNum("12343ms");
System.out.print("Name : "+ encap.getName()+" Age : "+ encap.getAge());
 }
}
```

This would produce the following result:

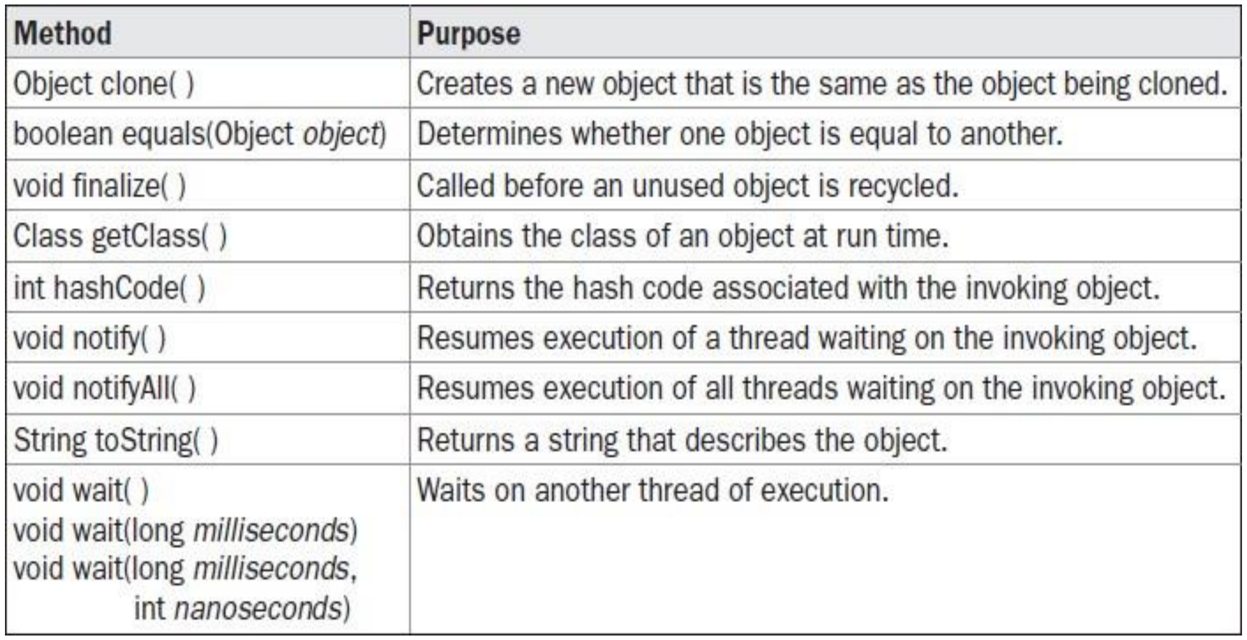
```
Name:JamesAge:20
```

Benefits of Encapsulation:

The fields of a class can be made read-only or write-only. A class can have total control over what is stored in its fields. The users of a class do not know how the class stores its data. A class can change the data type of a field and users of the class do not need to change any of their code.

The Object Class

There is one special class, Object, defined by Java. All other classes are subclasses of Object. That is, Object is a superclass of all other classes. This means that a reference variable of type Object can refer to an object of any other class. Also, since arrays are implemented as classes, a variable of type Object can also refer to any array. Object defines the following methods, which means that they are available in every object.



Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object *object*)	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled.
Class getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.

The methods getClass( ), notify( ), notifyAll( ), and wait( ) are declared as final. You may override the others. These methods are described elsewhere in this book. However, notice two methods now: equals( ) and toString( ). The equals( ) method compares the contents of two objects. It returns true if the objects are equivalent, and false otherwise.

19

Packages

A **java package** is a group of similar types of classes, interfaces and sub- packages. Package in java can be categorized in two form, built-in package and user-defined package. There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc. Here, we will have the detailed learning of creating and using user-defined packages.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.

Simple example of java package

The **package keyword** is used to create a package in java.

```
//save as Simple.java
package mypack; public
class Simple{
 public static void main(String args[]){
    System.out.println("Welcome to package");
    }
}
```

How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

```
> javac -d directory javafilename
```

For **example**

```
>javac -d . Simple.java
```

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

```
Output:Welcome to package
```

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

*1) Using packagename.**

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}


//save by B.java
package mypack;
import pack.*;

class B{
 public static void main(String args[]){ A obj
  = new A();
   obj.msg();
  }
}
```

Output:Hello

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

```
//save by A.java

package pack;
public class A{
     public void msg(){System.out.println("Hello");}
}



//save by B.java
package mypack;
import pack.A;

class B{
  public static void main(String args[]){ A obj
   = new A();
   obj.msg();
  }
}
```

Output:Hello

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
class B{
  public static void main(String args[]){
   pack.A obj = new pack.A();//using fully qualified name obj.msg();
  }
}
```

Output:Hello

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

Subpackage in java

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

Let's take an example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

Example of Subpackage

```
package com.javatpoint.core;
class Simple{
  public static void main(String args[]){ System.out.println("Hello
   subpackage");
  }
}
```

To Compile: javac -d . Simple.java

To Run: java com.javatpoint.core.Simple

Output:Hello subpackage

20

Interfaces

Interface inJava

An **interface in java** is a blueprint of a class. It has static constants and abstract methods only.

The interface in java is **a mechanism to achieve fully abstraction**. There can be only abstract methods in the java interface not method body. It is used to achieve fully abstraction and multiple inheritance in Java.

Java Interface also **represents IS-A relationship**. It

cannot be instantiated just like abstract class. **Why use**

Java interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve fully abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.

In other words, Interface fields are public, static and final bydefault, and methods are public and abstract.

Understanding relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface but a **class implements an interface**.

Class extends class

Class implements interface Interface

extends interface

Simple example of Java interface

In this example, Printable interface have only one method, its implementation is provided in the A class.

```
interface printable{
void print();
}

class A6 implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){ A6 obj
= new A6();
obj.print();
```

```
 }
}
```

Output:Hello

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.

```
interface Printable{
void print();
}

interface Showable{
void show();
}

class A7 implements Printable,Showable{

public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){ A7 obj
= new A7();
obj.print();
obj.show();
 }
}
```

```
Output:Hello
       Welcome
```

Multiple inheritance is not supported through class in java but it is possible by interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in case of class. But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class. For example:

```
interface Printable{
void print();
}
interface Showable{
void print();
}

class TestTnterface1 implements Printable,Showable{
public void print(){System.out.println("Hello");} public
static void main(String args[]){ TestTnterface1 obj =
new TestTnterface1(); obj.print();
 }
}
```

Output: Hello

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

Interface inheritance

A class implements interface but one interface extends another interface .

```
interface Printable{
void print();
}
interface Showable extends Printable{
void show();
}
class Testinterface2 implements Showable{

public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
Testinterface2 obj = new Testinterface2();
obj.print();
obj.show();
 }
}
```

Output:

```
Hello
Welcome
```

What is marker or tagged interface?

An interface that have no member is known as marker or tagged interface. For example: Serializable, Cloneable, Remote etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

```
//How Serializable interface is written?
public interface Serializable{
}
```

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

Abstract class	**Interface**
1) Abstract class can **have abstract and non-abstract**methods.	Interface can have **only abstract** methods.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final,**	Interface has **only static and final variables**.

static and non-static variables.	
4) Abstract class **can have static methods, main method and constructor**.	Interface **can't have static methods, main method or constructor**.
5) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
6) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.

7) **Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Example of abstract class and interface in Java

Let's see a simple example where we are using interface and abstract class both.

```
//Creating interface that has 4 methods
interface A{
void a();//bydefault, public and abstract
void b();
void c();
void d();
}
```

```
//Creating abstract class that provides the implementation of one method of A interface
abstract class B implements A{
public void c(){System.out.println("I am C");}
}

//Creating subclass of abstract class, now we need to provide the implementati on of rest of the
methods
class M extends B{
public void a(){System.out.println("I am a");} public
void b(){System.out.println("I am b");} public void
d(){System.out.println("I am d");}
}

//Creating a test class that calls the methods of A interface
class Test5{
public static void main(String args[]){ A
a=new M();
a.a();
a.b();
a.c();
a.d();
  }
}
```

Output:

```
I am a
I am b I
am c I
am d
```

21

Exception Handling

The **exception handling in java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.

In this page, we will learn about java exception, its type and the difference between checked and unchecked exceptions.

What is exception

Dictionary Meaning: Exception is an abnormal condition.

In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

What is exception handling

Exception Handling is a mechanism to handle runtime errors such as ClassNotFound, IO, SQL, Remote etc.

Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application**. Exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

Suppose there is 10 statements in your program and there occurs an exception at statement 5, rest of the code will not be executed i.e. statement 6 to 10 will not run. If we perform exception handling, rest of the statement will be executed. That is why we use exception handling in java.

Types of Exception

There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception. The sun microsystem says there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between checked and unchecked exceptions

1) Checked Exception

The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g.IOException, SQLException etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that extend RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

3) Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Common scenarios where exceptions may occur

There are given some scenarios where unchecked exceptions can occur. They are as follows:

1) Scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
int a=50/0;//ArithmeticException
```

2) Scenario where NullPointerException occurs

If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.

```
String s=null; System.out.println(s.length());//NullPointerException
```

3) Scenario where NumberFormatException occurs

The wrong formatting of any value, may occur NumberFormatException. Suppose I have a string variable that have characters, converting this variable into digit will occur NumberFormatException.

```
String s="abc";
int i=Integer.parseInt(s);//NumberFormatException
```

4) Scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:

```
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```

Java Exception Handling Keywords

There are 5 keywords used in java exception handling.

1. try
2. catch
3. finally
4. throw
5. throws

Java try-catch

Java try block

Java try block is used to enclose the code that might throw an exception. It must be used within the method.

Java try block must be followed by either catch or finally block.

Syntax of java try-catch

```
try{
//code that may throw exception
}catch(Exception_class_Name ref){}
```

Syntax of try-finally block

```
try{
//code that may throw exception
}finally{}
```

Java catch block

Java catch block is used to handle the Exception. It must be used after the try block only.

You can use multiple catch block with a single try.

Problem without exception handling

Let's try to understand the problem if we don't use try-catch block.

```
public class Testtrycatch1{
  public static void main(String args[]){
      int data=50/0;//may throw exception System.out.println("rest of the
      code...");
}
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
```

As displayed in the above example, rest of the code is not executed (in such case, rest of the code... statement is not printed).

There can be 100 lines of code after exception. So all the code after exception will not be executed.

Solution by exception handling

Let's see the solution of above problem by java try-catch block.

```
public class Testtrycatch2{
  public static void main(String args[]){
   try{
      int data=50/0;
   }catch(ArithmeticException e){System.out.println(e);}
   System.out.println("rest of the code...");
}
}
```

Output:

```
Exceptioninthreadmainjava.lang.ArithmeticException:/byzero rest of the
code...
```

Now, as displayed in the above example, rest of the code is executed i.e. rest of the code... statement is printed.

Java catch multiple exceptions

Java Multi catch block

If you have to perform different tasks at the occurrence of different Exceptions, use java multi catch block.

Let's see a simple example of java multi-catch block.

```
public class TestMultipleCatchBlock{
  public static void main(String args[]){ try{
    int a[]=new int[5];
    a[5]=30/0;
```

```
   }
   catch(ArithmeticException e){System.out.println("task1 is completed");}
   catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 compl eted");}
   catch(Exception e){System.out.println("common taskcompleted");}

   System.out.println("rest of the code...");
 }
}
```

```
Output:task1 completed rest of
       the code...
```

Rule: At a time only one Exception is occured and at a time only one catch block is executed.

Rule: All catch blocks must be ordered from most specific to most general i.e. catch for ArithmeticException must come before catch for Exception .

```
class TestMultipleCatchBlock1{
  public static void main(String args[]){
   try{
    int a[]=new int[5];
    a[5]=30/0;
   }
   catch(Exception e){System.out.println("common task completed");} catch(ArithmeticException
   e){System.out.println("task1 is completed");} catch(ArrayIndexOutOfBoundsException
   e){System.out.println("task 2  compl
eted");}
   System.out.println("rest of the code...");
 }
}
```

Output:

Compile-time error

Java Nested try block

The try block within a try block is known as nested try block in java.

Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Syntax:

```
....
try
{
    statement 1;
    statement 2;
    try
    {
        statement 1;
        statement 2;
    }
    catch(Exception e)
    {
    }
```

```
   }
catch(Exception e)
{
}
....
```

Java nested try example

Let's see a simple example of java nested try block.

```
class Excep6{
 public static void main(String args[]){
  try{
    try{
     System.out.println("going to divide");
     int b =39/0;
    }catch(ArithmeticException e){System.out.println(e);}

    try{
    int a[]=new int[5];
    a[5]=4;
    }catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}

    System.out.println("other statement);
  }catch(Exception e){System.out.println("handeled");}

  System.out.println("normal flow..");
 }
}
```

Java finally block

Java finally block is a block that is used *to execute important code* such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not. Java finally block

must be followed by try or catch block.

Note: If you don't handle exception, before terminating the program, JVM executes finally block(if any).

Why use java finally

- Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

Usage of Java finally

Let's see the different cases where java finally block can be used.

Case 1

Let's see the java finally example where **exception doesn't occur**. **class**

```
TestFinallyBlock{
  public static void main(String args[]){
  try{
   int data=25/5; System.out.println(data);
  }
  catch(NullPointerException e){System.out.println(e);}
  finally{System.out.println("finally block is always executed");}
  System.out.println("rest of the code...");
  }
}
```

```
Output:5
       finallyblockisalwaysexecuted rest of
       the code...
```

Case 2

Let's see the java finally example where **exception occurs and not handled**. **class**

```
  TestFinallyBlock1{
  public static void main(String args[]){
  try{
   int data=25/0; System.out.println(data);
  }
  catch(NullPointerException e){System.out.println(e);}
  finally{System.out.println("finally block is always executed");}
  System.out.println("rest of the code...");
  }
}
```

```
        Output:finally block is always executed
               Exception in thread main java.lang.ArithmeticException:/ by zero
```

Case 3

Let's see the java finally example where **exception occurs and handled**. **public**

```
class TestFinallyBlock2{
  public static void main(String args[]){
  try{
   int data=25/0; System.out.println(data);
  }
  catch(ArithmeticException e){System.out.println(e);}
  finally{System.out.println("finally block is always executed");}
  System.out.println("rest of the code...");
  }
}
```

```
Output:Exception in thread main java.lang.ArithmeticException:/ by zero finally block is
       always executed
       rest of the code...
```

Rule: For each try block there can be zero or more catch blocks, but only one finally block.

Note: The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

Java throw exception

Java throw keyword

The Java throw keyword is used to explicitly throw an exception.

We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

The syntax of java throw keyword is given below.

```
throw exception;
```

Let's see the example of throw IOException.

```
throw new IOException("sorry device error);
```

java throw keyword example

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
public class TestThrow1{
   static void validate(int age){
     if(age<18)
      throw new ArithmeticException("not valid");
     else
      System.out.println("welcome to vote");
   }
   public static void main(String args[]){
      validate(13);
      System.out.println("rest of the code...");
  }
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:not valid
```

Java Exception propagation

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method,If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack.This is called exception propagation.

Program of Exception Propagation

```
class TestExceptionPropagation1{
  void m(){
    int data=50/0;
  }
  void n(){
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handled");}
  }
  public static void main(String args[]){
   TestExceptionPropagation1obj=new TestExceptionPropagation1(); obj.p();
   System.out.println("normal flow...");
  }
}
        Output:exception handled normal
              flow...
```

Program which describes that checked exceptions are not propagated

```
class TestExceptionPropagation2{
  void m(){
    throw new java.io.IOException("device error");//checked exception
  }
  void n(){
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handeled");}
  }
  public static void main(String args[]){
   TestExceptionPropagation2obj=new TestExceptionPropagation2(); obj.p();
   System.out.println("normal flow");
  }
}
        Output:Compile Time Error
```

Java throws keyword

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

Syntax of java throws

```
return_type method_name() throws exception_class_name{
//method code
}
```

Which exception should be declared

Ans) checked exception only, because:

- **unchecked Exception:** under your control so correct your code.
- **error:** beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Advantage of Java throws keyword

Now Checked Exception can be propagated (forwarded in call stack).

It provides information to the caller of the method about the exception.

Java throws example

Let's see the example of java throws clause which describes that checked exceptions can be propagated by throws keyword.

```
import java.io.IOException;
class Testthrows1{
  void m()throws IOException{
    throw new IOException("device error");//checked exception
  }
  void n()throws IOException{
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handled");}
  }
  public static void main(String args[]){
   Testthrows1 obj=new Testthrows1(); obj.p();
   System.out.println("normal flow...");
  }
}
```

Output:

```
exception handled
normal flow...
```

There are two cases:

1. **Case1:**You caught the exception i.e. handle the exception using try/catch.
2. **Case2:**You declare the exception i.e. specifying throws with the method.

Case1: You handle the exception

In case you handle the exception, the code will be executed fine whether exception occurs during the program or not.

```
 import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }
}
public class Testthrows2{
public static void main(String args[]){
    try{
     M m=new M();
     m.method();
    }catch(Exception e){System.out.println("exception handled");}

    System.out.println("normal flow...");
  }
}
```

```
Output:exception handled normal
       flow...
```

Case2: You declare the exception

- A)In case you declare the exception, if exception does not occur, the code will be executed fine.
- B)In case you declare the exception if exception occures, an exception will be thrown at runtime because throws does not handle the exception.

A)Program if exception does not occur

```
import java.io.*;
class M{
 void method()throws IOException{
  System.out.println("device operation performed");
 }
}
class Testthrows3{
   public static void main(String args[])throws IOException{//declare exceptio
n
     M m=new M();
    m.method();

    System.out.println("normal flow...");
  }
}
```

```
     Output:device operation performed normal
            flow...
```

B)Program if exception occurs

```
import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }
}
class Testthrows4{
   public static void main(String args[])throws IOException{//declare exceptio
n
     M m=new M();
```

```
    m.method();

   System.out.println("normal flow...");
 }}
        Output:Runtime Exception
```

Difference between throw and throws in Java

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g.public void method()throws IOException, SQLException.

Java throw example

```
void m(){
throw new ArithmeticException("sorry");
}
```

Java throws example

```
void m()throws ArithmeticException{
//method code
}
```

Java throw and throws example

```
void m()throws ArithmeticException{
throw new ArithmeticException("sorry");
}
```

Difference between final, finally and finalize

There are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

No.	final	finally	finalize
1)	Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
2)	Final is a keyword.	Finally is a block.	Finalize is a method.

Java final example

```
class FinalExample{
public static void main(String[] args){
final int x=100;
x=200;//Compile Time Error
}
}
```

Java finally example

```
class FinallyExample{
public static void main(String[] args){
try{
int x=300;
}catch(Exception e){System.out.println(e);}
finally{System.out.println("finally block is executed");}
}
}
```

Java finalize example

```
class FinalizeExample{
public void finalize(){System.out.println("finalize called");}
public static void main(String[] args){
FinalizeExamplef1=new FinalizeExample();
FinalizeExamplef2=new FinalizeExample(); f1=null;
f2=null;
System.gc();
}
}
```

22

Input and Output

I/O Basics :

- Not much use has been made of I/O in the example programs.
- In fact, aside from print( ) and println( ), none of the I/O methods have been used significantly.
- The reason is simple: most real applications of Java are not text-based, console programs.
- Text-based console I/O is just not very important to Java programming.
- The preceding paragraph notwithstanding, Java does provide strong, flexible support for I/O as it relates to files and networks.
- Java‘s I/O system is cohesive and consistent.
- In fact, once you understand its fundamentals, the rest of the I/O system is easy to master.
- Rather, they are graphically oriented programs that rely upon Java‘s Abstract Window Toolkit (AWT) or Swing for interaction with the user.
- Although text-based programs are excellent as teaching examples, they do not constitute an important use for Java in the real world.
- Also, Java‘s support for console I/O is limited and somewhat awkward to use—even in simple example programs.

I/O Streams :

- Java programs perform I/O through streams.
- A *stream* is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system.
- All streams behave in the same manner, even if the actual physical devices to which they are linked differ.

- Java programs perform I/O through streams.
- A *stream* is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system.
- All streams behave in the same manner, even if the actual physical devices to which they are linked differ.

Byte Streams and Character Streams:

- **Java defines two types of streams: byte and character.**

- Byte streams provide a convenient means for handling input and output of bytes.
- Byte streams are used, for example, when reading or writing binary data.
- Character streams provide a convenient means for handling input and output of characters.
- They use Unicode and, therefore, can be internationalized.
- Also, in some cases, character streams are more efficient than byte streams.

- The original version of Java (Java 1.0) did not include character streams and, thus, all I/O was byte-oriented.
- Character streams were added by Java 1.1, and certain byte-oriented classes and methods were deprecated.
- This is why older code that doesn‘t use character streams should be updated to take advantage of them, where appropriate.
- One other point: at the lowest level, all I/O is still byte-oriented.
- The character-based streams simply provide a convenient and efficient means for handling characters.
- An overview of both byte-oriented streams and character-oriented streams is presented
  in the following sections.

The Byte Stream Classes:

- Byte streams are defined by using two class hierarchies.
- At the top are two abstract classes: InputStream and OutputStream.
- Each of these abstract classes has several concrete subclasses that handle the differences between various devices, such as disk files, network connections, and even memory buffers.
- The byte stream classes are shown in Table 13-1.
- Remember, to use the stream classes, you must import java.io
- The abstract classes InputStream and OutputStream define several key methods that the other stream classes implement.
- Two of the most important are read( ) and write( ), which, respectively, read and write bytes of data.
- Both methods are declared as abstract inside InputStream and OutputStream.
- They are overridden by derived stream classes.

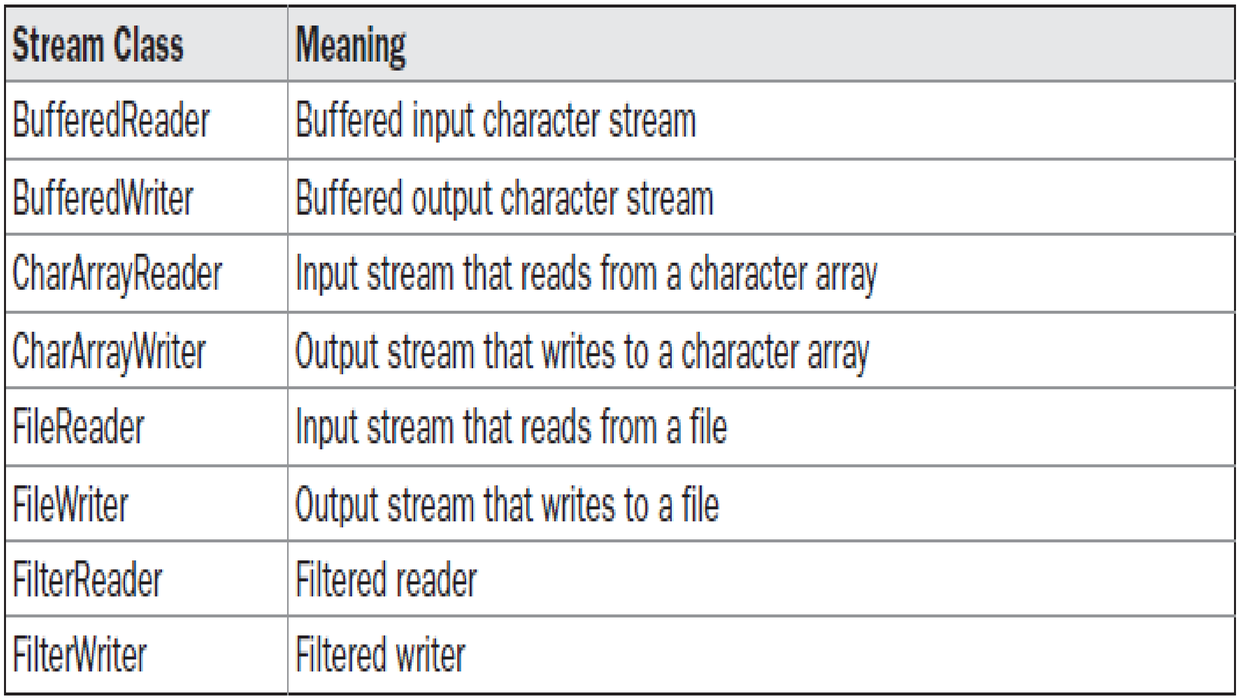
TABLE 13-1 The Byte Stream Classes

Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements **InputStream**
FilterOutputStream	Implements **OutputStream**
InputStream	Abstract class that describes stream input
ObjectInputStream	Input stream for objects
ObjectOutputStream	Output stream for objects
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains **print( )** and **println( )**
PushbackInputStream	Input stream that supports one-byte "unget," which returns a byte to the input stream
RandomAccessFile	Supports random access file I/O
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

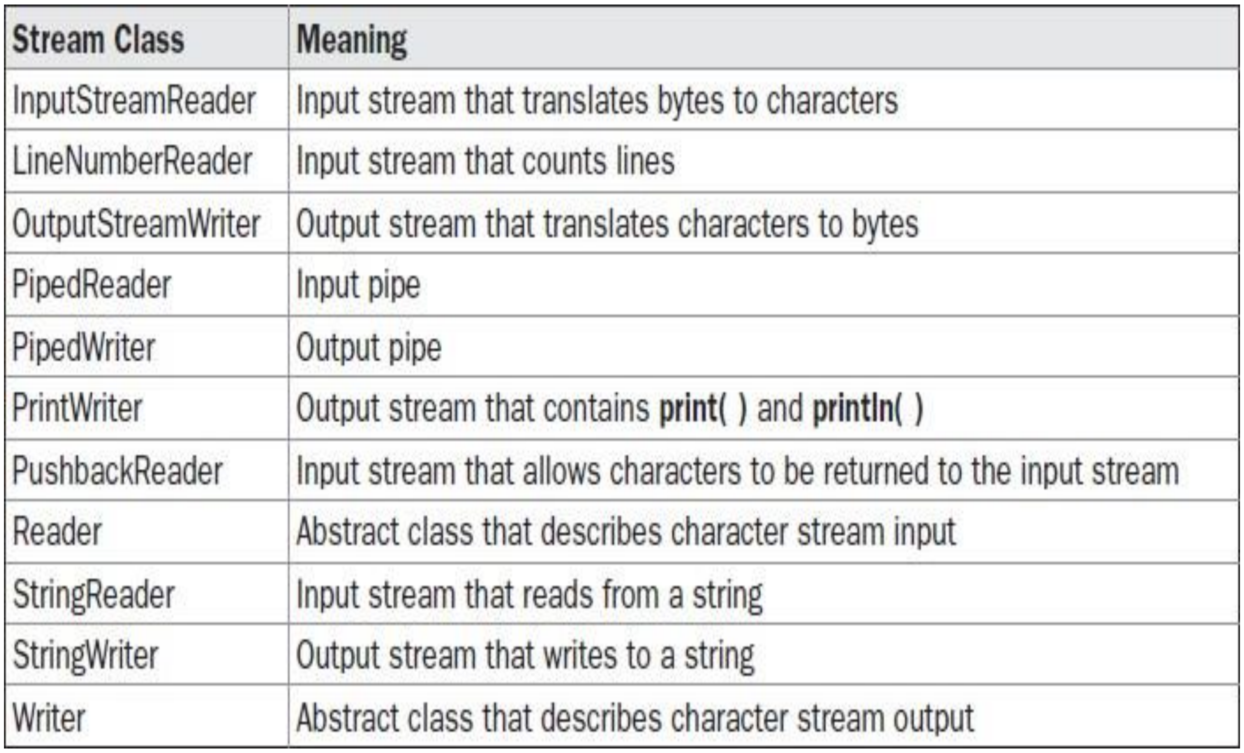
The Character Stream Classes:

- Character streams are defined by using two class hierarchies.
- At the top are two abstract classes, Reader and Writer.
- These abstract classes handle Unicode character streams.
- Java has several concrete subclasses of each of these.
- The character stream classes are shown in Table
- The abstract classes Reader and Writer define several key methods that the other stream classes implement.
- Two of the most important methods are read( ) and write( ), which read and write characters of data, respectively.
- These methods are overridden by derived stream classes.

Table :13.2 The Character Stream I/O Classes



Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer



Stream Class	Meaning
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains **print( )** and **println( )**
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

The Predefined Streams:

- As you know, all Java programs automatically import the java.lang package.
- This package defines a class called System, which encapsulates several aspects of the run-time environment.
- For example, using some of its methods, you can obtain the current time and the settings of various properties associated with the system.
- System also contains three predefined stream variables: in, out, and err.
- These fields are declared as public, static, and final within System.
- This means that they can be used by any other part of your program and without reference to a specific System object.
- System.out refers to the standard output stream. By default, this is the console.
- System.in refers to standard input, which is the keyboard by default.
- System.err refers to the standard error stream, which also is the console by default.

However, these streams may be redirected to any compatible I/O device.

- System.in is an object of type InputStream; System.out and System.err are objects of type PrintStream.
- These are byte streams, even though they typically are used to read and write characters from and to the console.
- As you will see, you can wrap these within characterbased streams, if desired.

Reading Console Input:

- In Java, console input is accomplished by reading from System.in.

- To obtain a characterbased stream that is attached to the console, wrap System.in in a BufferedReader object.
- BufferedReader supports a buffered input stream.
- Its most commonly used constructor is shown here:

  BufferedReader(Reader *inputReader)*
- Here, *inputReader is the stream that is linked to the instance of BufferedReader that is being* created.
- Reader is an abstract class.
- One of its concrete subclasses is InputStreamReader, <u>which converts bytes to characters.</u>
- To obtain an InputStreamReader object that is linked to System.in, use the following constructor:

  InputStreamReader(InputStream *inputStream)*
- Because System.in refers to an object of type InputStream, it can be used for *inputStream*.
- Putting it all together, the following line of code creates a BufferedReader that is connected to the keyboard:

```
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
```

- After this statement executes, br is a character-based stream that is linked to the console through System.in.

Reading Characters:

- To read a character from a BufferedReader, use read( ).
- The version of read( ) that we will be using is

  int read( ) throws IOException Each time that read( ) is called, it reads a character from the input stream and returns it as an integer value. It returns – 1 when the end of the stream is encountered.
- As you can see, it can throw an IOException
- The following program demonstrates read( ) by reading characters from the console until the user types a "q." Notice that any I/O exceptions that might be generated are simply thrown out of main( ).
- Such an approach is common when reading from the console, but you can handle these types of errors yourself, if you chose.

Reading Characters program:

```
// Use a BufferedReader to read characters import
java.io.*;
class BRRead {
public static void main(String args[]) throws
IOException
{
char c;
BufferedReader br = new BufferedReader(new InputStreamReader(System.in)); System.out.println("Enter
characters, 'q' to quit.");
//read characters do {
c = (char) br.read();
System.out.println(c);
} while(c != 'q');
}
}
```

Here is a sample run:

```
Enter characters, 'q' to quit.
123abcq
1
2
3
a
b
c
q
```

- This output may look a little different from what you expected, because System.in is linebuffered, by default.
- This means that no input is actually passed to the program until you press ENTER.
- As you can guess, this does not make read( ) particularly valuable for interactive console input.

Reading Streams:

- To read a string from the keyboard, use the version of readLine( ) that is a member of the BufferedReader class.
- Its general form is shown here:
  String readLine( ) throws IOException
- As you can see, it returns a String object.

The following program demonstrates BufferedReader and the readLine( ) method; the program reads and displays lines of text until you enter the word ―stop‖: class BRReadLines

```
{
public static void main(String args[]) throws
IOException
{
// create a BufferedReader using System.in
BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
String str;
System.out.println("Enter lines of text.");
System.out.println("Enter 'stop' to quit."); do {
str = br.readLine(); System.out.println(str);
} while(!str.equals("stop"));
}
}
```

- The next example creates a Tiny text editor.
- It creates an array of String objects and then reads in lines of text, storing each line in the array.
- It will read up to 100 lines or until you enter ―stop.‖
- It uses a BufferedReader to read from the console.

```
// A tiny editor.
import java.io.*;
class TinyEdit {
public static void main(String args[])
throws IOException
{
BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
String str[] = new String[100];
System.out.println("Enter lines of text.");
System.out.println("Enter 'stop' to quit."); for(int
i=0; i<100; i++){
str[i] = br.readLine();
if(str[i].equals("stop"))  break;
}
System.out.println("\nHere is your file:");
// display the lines for(int
i=0; i<100; i++) {
if(str[i].equals("stop")) break;
System.out.println(str[i]);
}
}
}
```

Here is a sample run: Enter

```
lines of text.
Enter 'stop' to quit.
This is line one.
This is line two.
Java makes working with strings easy. Just
create String objects.
stop
Here is your file:
This is line one.
This is line two.
```

Java makes working with strings easy. Just
create String objects.

Reading Console Input Scanner:

- Scanner is the complement of Formatter.
- Added by JDK 5, Scanner reads formatted input and converts it into its binary form.
- Although it has always been possible to read formatted input, it required more effort than most programmers would prefer.
- Because of the addition of Scanner, it is now easy to read all types of numeric values, strings, and other types of data, whether it comes from a disk file, the keyboard, or another source.
- Scanner can be used to read input from the console, a file, a string, or any source that implements the Readable interface or ReadableByteChannel.
- For example, you can use Scanner to read a number from the keyboard and assign its value to a variable.
- As you will see, given its power, Scanner is surprisingly easy to use.

- The addition of Scanner to Java makes what was formerly a tedious task into an easy one.
- To understand why, let‘s look at some examples.
- // Use Scanner to compute an average of the values in a file.

```
import java.util.*;

import java.io.*; class

AvgFile {
```

```
public static void main(String args[]) throws

IOException {

int count = 0; double

sum=0.0;

//Write output to a file.

FileWriter fout = new FileWriter("test.txt"); fout.write("2

3.4 5 6 7.4 9.1 10.5 done"); fout.close();

FileReaderfin=newFileReader("Test.txt"); Scanner

src = newScanner(fin);

// Read and sumnumbers.

while(src.hasNext()) {

if(src.hasNextDouble()) { sum +=

src.nextDouble(); count++;

}

else {

String str = src.next();

if(str.equals("done")) break; else {

System.out.println("File format error.");
return;
}
}
}
fin.close();
System.out.println("Average is " + sum / count);
}
}
```

- Here is the output:

```
Average is 6.2
```

Writing Console Output :

- Console output is most easily accomplished with print( ) and println( ), described earlier, which are used in most of the examples in this book.
- These methods are defined by the class PrintStream (which is the type of object referenced by System.out).
- Even though System.out is a byte stream, using it for simple program output is still acceptable.
- Because PrintStream is an output stream derived from OutputStream, it also implements the low-level method write( ).
- Thus, write( ) can be used to write to theconsole.
- The simplest form of write( ) defined by PrintStream is shown here:
- void write(int *byteval)*
- This method writes to the stream the byte specified by *byteval.*

- *Although byteval is declared* as an integer, only the low-order eight bits are written.
- Here is a short example that uses write( ) to output the character —A‖ followed by a newline to the screen:

```
//Demonstrate System.out.write(). class
WriteDemo {
public static void main(String args[]) { int b;
b = 'A';
System.out.write(b); System.out.write('\n');
}
}
```

You will not often use write( ) to perform console output (although doing so might be useful in some situations), because print( ) and println( ) are substantially easier to use.

The PrintWriter Class:

- Although using System.out to write to the console is acceptable, its use is recommended mostly for debugging purposes or for sample programs.
- For real-world programs, the recommended method of writing to the console when using Java is through a PrintWriter stream.
- PrintWriter is one of the character-based classes.
- Using a character-based class for console output makes it easier to internationalize your program.
- PrintWriter defines several constructors. The one we will use is shown here:

PrintWriter(OutputStream *outputStream, boolean flushOnNewline)*

- Here, outputStream is an object of type OutputStream, and flushOnNewline controls whether Java flushes the output stream every time a println( ) method is called.
- If flushOnNewline is true, flushing automatically takes place.
- If false, flushing is not automatic.
- PrintWriter supports the print( ) and println( ) methods for all types including Object.
- Thus, you can use these methods in the same way as they have been used with System.out.
- If an argument is not a simple type, the PrintWriter methods call the object's toString( ) method and then print the result.
- To write to the console by using a PrintWriter, specify System.out for the output stream and flush the stream after each newline.
- For example, this line of code creates a PrintWriter that is connected to console output:
- PrintWriter pw = new PrintWriter(System.out, true);

```
// Demonstrate PrintWriter import
java.io.*;
public class PrintWriterDemo {
public static void main(String args[]) {
PrintWriter pw = new PrintWriter(System.out, true);
pw.println("This is a string");
int i = -7;
pw.println(i); double d
=4.5e-7;
pw.println(d);
}
}
```

The output from this program is shown here: This is a

string

-7

4.5E-7

- Remember, there is nothing wrong with using System.out to write simple text output to the console when you are learning Java or debugging your programs.
- However, using a PrintWriter will make your real-world applications easier to internationalize.
- Because no advantage is gained by using a PrintWriter in the sample programs shown in this book, we will continue to use System.out to write to the console.

Writing data into a file :

The following program to enter data into a new file or appends into an existing file. An employee database with three fields name, number and basic salary for number of employees can be entered into a file ―scores.txt‖.

```
import java.io.*;
import java.util.Scanner; class
WriteData{
public static void main(String args[])throws Exception{

BufferedWriter output=null; File fil=
new File("scores.txt");

if(!fil.exists()){
   fil.createNewFile();
 }

FileWriter file= new FileWriter("scores.txt", true);

Scanner in=new Scanner(System.in);

int i,n;
String nm,num,bas;

output=new BufferedWriter(file); System.out.print("Enter
How many records:"); n=in.nextInt();
for (i=0;i<n;i++){

System.out.println("Enter name:");
nm=in.next(); System.out.println("Enter
number:"); num=in.next();
System.out.println("Enter basic:");
bas=in.next();

output.write(nm+" ");
output.write(num+" ");
output.write(bas+" ");
output.newLine();
}
output.close();
 }
}
```

Reading data from a Text file:

The following program reads the data from ―scores.txt‖ and displays the content.

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

class  ReadData{
public static void main(String args[])throws Exception{ try{
File file=new File("scores.txt"); String
name,n2,n3;
int num;
double sal;

Scanner input=newScanner(file);
while(input.hasNext()) {
name=input.next(); n2=input.next();
n3=input.next();

num=Integer.parseInt(n2); sal
=Double.parseDouble(n3);
System.out.println(name+ " "+num+ " "+sal );

}

}catch(FileNotFoundException fnf){
  System.out.println(fnf);
        System.out.println(" pl. provide me the file");
}
}
}
```

Program (reads for instance variables in a class):

When employee class file contains name, number and basic are the instance fields and to process salary for number of employees, number of objects aerated and salary is processed for each employee and display the output.
Note: This program is so useful while doing lab based projects.

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

class Employee{
String name; int
num; double basic;

Employee(String m, int n, double b){
name=m;
num=n;
basic=b;
}

void salary(){
double sal=basic+1000; System.out.println(name+" "
+"salary="+sal);
}
public static void main(String args[])throws Exception{ Employee
emp[] = newEmployee[10];

File file=newFile("scores.txt"); String
name,n2,n3;
int num,i=0;
double bas;

Scanner input=newScanner(file);
while(input.hasNext()) {
name=input.next(); n2=input.next();
n3=input.next();

num=Integer.parseInt(n2); bas
=Double.parseDouble(n3);
emp[i]=new Employee(name,num,bas); emp[i].salary();

i++;
}
}
}
```

23

Multithreading

Multithreading in java is a process of executing multiple threads simultaneously.

Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation etc.

Advantage of Java Multithreading

1) It doesn't block the user because threads are independent and you can perform multiple operations at same time.

2) You can perform many operations together so it saves time.

3) Threads are independent so it doesn't affect other threads if exception occur in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved by two ways:

- Process-based Multitasking(Multiprocessing)
- Thread-based Multitasking(Multithreading)

1) Process-based Multitasking (Multiprocessing)

- Each process have its own address in memory i.e. each process allocates separate memory area.
- Process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

2) Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- Thread is lightweight.
- Cost of communication between the thread is low.

What is Thread in java

A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution.

Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memoryarea.

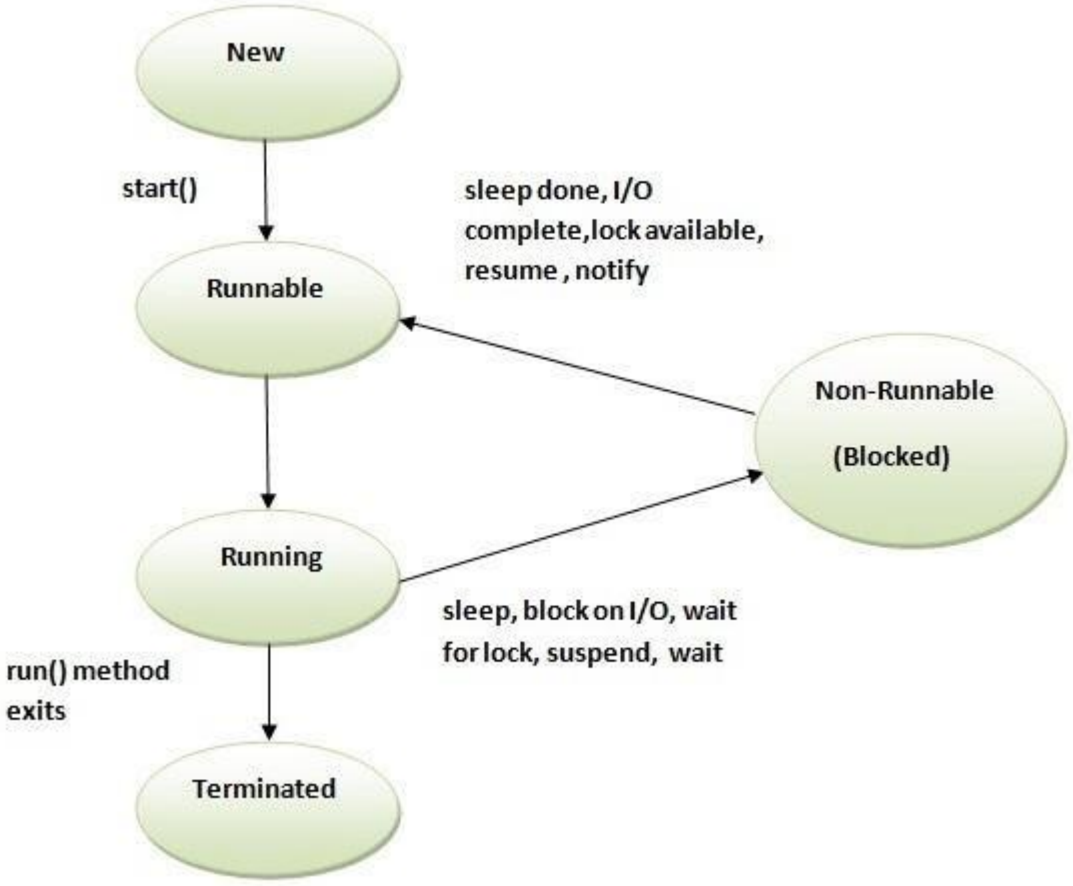
Life cycle of a Thread (Thread States)

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New : The thread is in new state if you create an instance of Thread class but before the invocation of start() method.
2. Runnable : The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.
3. Running: The thread is in running state if the thread scheduler has selected it.
4. Non-Runnable (Blocked): This is the state when the thread is still alive, but is currently not eligible to run.
5. Terminated: A thread is in terminated or dead state when its run() method exits.


New
start()
Runnable
sleep done, I/O complete, lock available, resume , notify
Non-Runnable (Blocked)
Running
sleep, block on I/O, wait for lock, suspend, wait
run() method exits
Terminated

How to create thread

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.

Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

1. **public void run():** is used to perform action for a thread.

Starting a thread:

start() method of Thread class is used to start a newly created thread. It performs following tasks:

- A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

1) By extending Thread class:

```
class Multi extends Thread{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi t1=new Multi();
t1.start();
 }
}
```

Output:thread is running...

2) By implementing the Runnable interface:

```
class Multi3 implements Runnable{ public void
run(){ System.out.println("thread is running...");
}

public static void main(String args[]){ Multi3
m1=new Multi3();
Thread t1 =new Thread(m1);
t1.start();
 }
}
```

Output:thread is running...

Thread Scheduler in Java

Thread scheduler in java is the part of the JVM that decides which thread should run.

There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.

Only one thread at a time can run in a single process.

The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

Sleep method in java

The sleep() method of Thread class is used to sleep a thread for thespecified amount of time.

Syntax of sleep() method in java

The Thread class provides two methods for sleeping a thread:

- public static void sleep(long miliseconds)throws InterruptedException
- public static void sleep(long miliseconds, int nanos)throws InterruptedException

Example of sleep method in java

```
class TestSleepMethod1 extends Thread{
 public void run(){
  for(int i=1;i<5;i++){
    try{Thread.sleep(500);}catch(InterruptedException e){System.out.printl
n(e);}
    System.out.println(i);
  }
 }
 public static void main(String args[]){
  TestSleepMethod1 t1=new TestSleepMethod1();
  TestSleepMethod1 t2=new TestSleepMethod1();
  t1.start();
  t2.start();
 }
}
```

Output:

```
  1
        1
        2
        2
        3
        3
        4
        4
```

As you know well that at a time only one thread is executed. If you sleep a thread for the specified time,the thread shedular picks up another thread and so on.

```
class TestCallRun2 extends Thread{
 public void run(){
  for(int i=1;i<5;i++){
    try{Thread.sleep(500);}catch(InterruptedException e){System.out.printl
n(e);}
    System.out.println(i);
  }
 }
 public static void main(String args[]){
  TestCallRun2 t1=new TestCallRun2();
  TestCallRun2 t2=new TestCallRun2();

  t1.run();
  t2.run();
 }
}
```

```
Output:1 2
       3
       4
       5
       1
       2
       3
       4
       5
```

The join() method:

The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

Syntax:

```
public void join()throws InterruptedException

public void join(long milliseconds)throws InterruptedException
```

Example of join() method

```
class TestJoinMethod1 extends Thread{
public void run(){
 for(int i=1;i<=5;i++){
  try{
  Thread.sleep(500);
 }catch(Exception e){System.out.println(e);}
 System.out.println(i);
 }
 }
public static void main(String args[]){
TestJoinMethod1 t1=new TestJoinMethod1();
TestJoinMethod1 t2=new TestJoinMethod1();
TestJoinMethod1 t3=new TestJoinMethod1();
t1.start();
try{
  t1.join();
 }catch(Exception e){System.out.println(e);}

 t2.start();
 t3.start();
 }
}
```

Priority of a Thread (Thread Priority):

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defiend in Thread class:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example of priority of a Thread:

```
class TestMultiPriority1 extends Thread{
 public void run(){
   System.out.println("running thread name is:"+Thread.currentThread().ge
tName());
   System.out.println("running thread priority is:"+Thread.currentThread().
getPriority());

  }
 public static void main(String args[]){
  TestMultiPriority1 m1=new TestMultiPriority1();
  TestMultiPriority1 m2=new TestMultiPriority1();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
  m1.start();
  m2.start();

 }
```

}

```
Output:running thread name is:Thread-0 running
       thread priority is:10 running thread name
       is:Thread-1 running thread priority is:1
```

--End of Lecture Notes –