History of Java

• Java was initially developed in 1991named as "oak" but was renamed "Java" in 1995.
• Originally designed for small, embedded systems in electronic appliances like set-top boxes.
• The primary motivation was the need for a platform-independent language that could be used to create software to be embedded in various consumer electronic devices.
• 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]}.
• It promised Write Once, Run Anywhere (WORA), providing no-cost run-times on popular platforms.
• Java 2, new versions had multiple configurations built for different types of platforms. J2EE included technologies and APIs for enterprise applications typically run in server environments, while J2ME featured APIs optimized for mobile applications.
• The desktop version was renamed J2SE. In 2006, for marketing purposes, Sun renamed new J2 versions as Java EE, Java ME, and Java SE, respectively.
• On 13 November 2006, Sun released much of Java as free and open-source software (FOSS), 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.

Features/Advantages of Java

• Java promised "Write Once, Run Anywhere", providing no-cost run-time on popular platform.
• Fairly secure and featuring configurable security, it allowed network and file access restriction.

Simple

• It's simple because it contains many features of other languages like C and C++ and java removes complexities because it doesn't use pointers, Storage classes and Go To statement and it also does not support multiple inheritance.

Secure

• When we transfer the code from one machine to another machine, it will first check the code it is affected by the virus or not, it checks the safety of

the code, if it contains virus then it will never execute that code.

Object-Oriented

• We know that all pure object oriented language, in them all of the code is in the form of classes and objects.
• This feature of java is most important and it also supports code reusability and maintainability etc.

Robust

Two main reasons for program failures are:
1. Memory management mistake
2. Mishandled exception or Run time errors
• Java does not support direct pointer manipulation. This resolves the java program to overwrite memory,
• Java manages the memory allocation and de-allocation itself. De-allocation is completely automatic, because Java provides garbage collection for unused objects.
• Java provides object-oriented exception handling. In a well written Java program all runtime errors can be managed by the program.

Multithreaded

• A thread is like a separate program executing concurrently.
• We can write java programs that deal with many tasks at once by defining multiple threads.
• The main advantage of multithreading is that it shares the same memory.
• Threads are important for multi-media, web application etc.

Distributed

• Java is designedfor thedistributed environment of theInternet,becauseit handles TCP/ IP protocols.
• The widely used protocol like HTTP and FTP are developed in Java.
• Internet programmers can call functions on these protocols and can get access the files from any remote machine on the internet rather than writing code on their local system.

Architecture-Neutral

• It means that the programs written in one platform can run on any other platform without rewrite or recompilemthem. In other words it follows "write once, run anywhere, any time, forever" approach,
• Java program are compiled into bytecode format which does not depend on any machine architecture but can be easily translated into a specific machine by a JVM for that machine.
• This will be very helpful when applets or applications are developed which are download by any machine & run anywhere in any system.

Platform Independent

• It means when we compile a program in java,it will create a byte code of that program and that byte code will be executed when we run the program.
• It's not compulsory in java, that in which operating system we create java program, in the same operating system.
• we have to execute the program.

Interpreted

• Most of the programming languages either compiled or interpreted, java is both compiled and interpreted.
• Java compiler translates a java source file to byte code and the java interpreter executes the translated byte codes directly on the system that implements the JVM.

High Performance

• Java programs are complied in to intermediate representation called bytecode, rather than to native machine level instructions and JVM executes Java bytecode on any machine on which JVM is installed.
• Java bytecode then translate directly into native machine code for very high performance by using a Just-In-Time compiler.
• So,Java programs are faster than program or scripts written in purely interpreted languages but slower than C and C++ programs that compiled to native machine languages.

Dynamic

• At the run time, java environment can extend itself by linking in classes that may be located on remote serveron a network,
• At the run time, java interpreter performs name resolution while linking in the necessary classes.
• The java interpreter is responsible for determining the placement of object in the memory.

JDK (Java Development Kit)

• Java Developer Kit contains tools needed to develop the Java programs, and JRE to run the programs.
• The tools include compiler (javac.exe), Java application launcher (java.exe), Appletviewer, etc Compiler converts java code into byte code.
• Java application launcher opens a JRE, loads the class, and invokes its main method.
• For running java programs, JRE is sufficient. JRE is targeted for execution of Java files i.e. JRE = JVM + Java Packages Classes(like util, math, lang, awt,swing etcj+runtime libraries.

JRE (Java Runtime Environment)

• Java Runtime Environment contains JVM, class libraries, and other supporting files.
• It does not contain any development tools such as compiler, debugger, etc.
• Actually, JVM runs the program, and it uses the class libraries, and other supporting files provided in JRE.

JVM (Java Virtual Machine)

• The JVM is called virtual because it provides a machine interface that does not depend on the operating system and machine hardware architecture.
• This independence from hardware and operating system is a cornerstone of the write-once, run-anywhere java programs.
• When we compile a Java file, output is not an '.exe' but it is a '.class' file.
• '.class' file consists of Java byte codes which are understandable by JVM.
• Java Virtual Machine interprets the byte code into the machine code depending upon the operating system and hardware combination.
• It is responsible for all the things like garbage collection, array bounds checking, etc.
• JVM itself is platform dependent
• As of 2014 most JVMs use JIT (Just in Time} compiling, not interpreting, to achieve greater speed.

BYTE Code

• Bytecode is nothing but the intermediate representation of Java source code which is produced by the Java compiler by compiling that source code.
• This byte code is a machine independent code. It is not completely a compiled code but it is an intermediate code somewhere in the middle which is later interpreted and executed by JVM..
• Bytecode is a machine code for JVM. But the machine code is platform specific whereas bytecode is platform independent that is the main difference between them.
• It is stored in .class file which is created after compiling the source code.

JAVA Environment Setup

• Setting up the path for windows: 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, Under 'System variables' alter the ’Path1 variable so that it also contains the path to the Java executable.
• Example, if the path is currently set to 'C:\WINDOWS\SYSTEM321, then

change your path to read 'C:\WINDOWS\SYSTEM32;c:\Program Fi!es\java\jdk\bin'.

Structure of JAVA program

• A Java Program may contain many classes of which only one class defines a main method.
• Classes contain data members and methods that operate on the data members of the class.
• Methods may contain data type declarations and executable statements.

Documentation Section

• The documentation section comprises a set of comment lines giving the name of the program, the author and other details, which the programmers would like to refer to at a later stage.
• Java also uses a comment such as /* .... * / know as documentation comment.
• This form is used for documentation automatically.

Package Statement

• The first statement allowed in Java file is a package statement.
• This statement declares a package name and informs the compiler that the classes defined here belong to this package.
Example: Package Student;
• The main advantage of multithreading is that it shares the same memory.
• Threads are important for multi-media, web application etc.

Import Statements

• The next thing after a package statement may be a number of import statements. This is similar to the ^include statement in C.
Example: Import student.test;

Interface Statement

• An interface is like a class but includes a group of method declarations.
• This is also an optional section and is used only when we wish to implement the multiple inheritance features in the program.

Class Definition

• A Java Program may contain multiple class definitions.
• Classes are primary and essential elements of Java program.
• These classes are used to map the objects of real-world programs.

Main Method Class

• Since, every Java stand-alone program requires a main method as its starting point this is the essential part of Java program.
• A simple Java program may contain only this part.
• The main method creates objects of various classes and establishes

communication between them.

Basic Concepts of OOP

• Various concepts present in OOP to make it more powerful, secure, reliable and easy.

Object

• An object is an instance of a class.
• An object means anything from real world like as person, computer etc.
• Every object has at least one unique identity.
• An object is a component of a program that knows how to interact with other pieces of the program.
• An object is the variable of the type class.

Class

• A class is a template that specifies the attributes and behavior of objects.
• A class is a blueprint or prototype from which objects are created.
• Simply class is collection of objects.
• A class is the implementation of an abstract data type (ADT). it defines attributes and methods which implement the data structure and operations of the ADT, respectively.

Data Abstraction

• Just represent essential features without including the background details.
• Implemented in class to provide data security.
• We use abstract class and interface to achieve abstraction.

Encapsulation

• Wrapping up (Binding) of a data and functions into single unit is known as encapsulation.
• The data is not accessible to the outside world, only those functions can access it which is wrapped together within single unit.

Inheritance

• Inheritance is the process, by which class can acquire the properties and methods of another class.
• The mechanism of deriving a new class from an old class is called inheritance.
• The new class is called derived class and old class is called base class.
• The derived class may have all the features of the base class and the programmer can add new features to the derived class.

Polymorphism

• Polymorphism means the ability to take more than one form.
• It allows a single name to be used for more than one related purpose.
• It means ability of operators and functions to act differently in different situations.
• We use method overloading and method overriding to achieve polymorphism.

Message Passing

• A program contains set of object that communicates with each other.
• Basic steps to communicate
1. Creating classes that define objects and their behavior.
2. Creating objects from class definition
3. Establishing communication among objects.

Dynamic and Static Binding

First Java program:

FirstProgram.java

```
import java.util;
class FirstProgram
{
String str;
public static void main(String[] args)
{
System.out.println("My First Program In Java...");
}
```

Building Blocks of the Language

Primitive Data Types

• Primitive data types can be classified in four groups:

1) Integers:
• This group includes byte, short, int, and long.
• All of these are signed, positive and negative values.
byte S-bit
short 16-bi
int 32-bit
long 64-bit

2) Floating-point:
• This group includes float and double, which represent numbers with fractional precision.
float 32 bits
double 64 bits

3) Characters:
• This group includes char, which represents symbols in a character set, like letters and numbers.
• Java char is a 16-bit type. The range of a char is 0 to 65,536. There are no negative chars.
For example : char name = 'x';

4) Boolean:
• This group includes boolean, which is a special type for representing true/false values.
• It can have only one of two possible values, true or false.
For example : boolean b = true;

An example for Primitive Data Types:

```
import java.util.*;
class Pdatatype
{
public static void main(String args[])
{
int r=10;
float pi=3.14f,a;
char ch=97,c2='A';
boolean x=true;
a = pi*r*r;
System.out.println("Area of circle is: "+a);
System.out.println("C1 and C2 are: "+cl+" ”+c2);
```

```
System.out.println("Value of X is: "+x);
}
}
```

User Defined Data Types

• User Defined data types can be classified into the following types:

1) Class:
• A class is a template that specifies the attributes and behavior of things or objects.
• A class is a blueprint or prototype from which creates as many desired objects as required.

2) Interface:
• An interface is a collection of abstract methods.
• A class implements an interface,thereby inheriting the abstract methods of the interface.
• An interface is not a class. A class describes the attributes and behaviors of an object. An interface contains behaviors that a class implements.
• Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.
Example:

```
interface Animal
{
public void eat();
public void travel();
}
```

An example for User Defined Data Types:

```
import java.util.*;
class Box
{
double width=l,height=2,depth=3;
void volume()
{
System.out.print("Volume is ");
System.out.println(width * height * depth};
}
}
class demo
{
public static void main(String args[])
{
Box bl = new Box();
```

```
bl.volume();
}
}
```

Identifiers and Literals

1) Identifiers:
• Identifiers are used for class names, method names, and variable names.
• An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore characters and dollar-sign characters.
• There are some rules to define identifiers as given below:
1) Identifiers must start with a letter or underscore { _).
2) Identifiers cannot start with a number.
3) White spacefblank, tab, newline) are not allowed.
4) You can't use a Java keyword as an identifier.
5) Identifiers in Java are case-sensitive; foo and Foo are two different identifiers.

2) Literals (Constants):
• A constant value in a program is denoted by a literal. Literals represent numerical (integer or floating-point), character, boolean or string values.
Integer - 33, 0,-19
Floating-point - 0.3, 3.14
Character - 'R'
Boolean - true, false
String - "language","0.2","r"

Variables

• A variable is defined by the combination of an identifier, a type, and an optional initializer.
• All variables have a scope, which is defines their visibility and lifetime.

Declaring of variable
• All variables must be declared before they can be used.
Syntax:
type identifier [= value][,identifier [= value] ...];
• The type is one of Java's atomic types, or the name of a class or interface.
• The identifier is the name of the variable.
• You can initialize the variable by specifying an equal sign and a value.
• To declare more than one variable of the specified type, use a comma separated list.
Example: int a, b, c = 10;

Dynamic Initialization

• Java allows variables to be initialized dynamically by using any valid expression at the time the variable is declared.

Example:

```
int a=2, b=3;
int c = a+b; // Dynamic initialization
```

Scope of Variables

• Java allows variables to be declared within any block.

• A block is begun with an opening curly brace and ended by a closing curly brace.

• A block defines a scope. Thus, each time you start a new block, you are creating a new scope.

• A scope determines which objects are visible to other parts of your program.

• Java defines two general categories of scopes: global and local.

• Variables declared inside a scope is not visible to code that is defined outside that scope.

• Thus, when you declare a variable within a scope, then you can access it within that scope only and protecting it from unauthorized access.

• Scopes can be nested. So, outer scope encloses the inner scope. This means that objects declared in the outer scope will be visible to code within the inner scope.

• However, the reverse is not true. Objects declared within inner scope will not be visible outside it.

Example:

```
class ScopeOfVari
{
public static void main(String argsf])
{
int x; // visible to all code within the main
x = 10;
if(x == 10) // starting new scope
{
int y = 20; // Visible only to this block
System,out.println("x and y; " + x + " " + y); // x and y both are visible here.
x — y * 2;
}
System.out.println("x is " + x); // y is not visible here but x is.
}
}
```

Default values of variables declared

• If you are not assigning value, then Java runtime assigns default value to variable and when you try to access the variable you get the default value of that variable.

• Following shows variable types and their default values:

boolean - FALSE
char - \u0000
int,short,byte/long - 0 / 0L
float/double - O.Of / O.Od
any reference type - null

• Here, char primitive default value is \u0000, which means blank/space character.

• When you declare any local/block variable,they didn't get the default values.

Type Conversion and Casting

• It assigns a value of one type variable to a variable of another type. If the two types are compatible, then Java will perform the conversion automatically (Implicit conversion).

• For example, it is always possible to assign an int value to a long variable. However, not all types are compatible,and thus, not all type conversions are implicitly allowed.

• There is no automatic conversion defined from double to byte. Fortunately, it is possibleconversion between incompatible types. For that you must perform type casting operation,which performs an explicit conversion between incompatible types.

1) Implicit Type Conversion (Widening Conversion)

• When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:

1. Two types are compatible.
2. Destination type is larger than the source type.

• When these two conditions are met, a widening conversion takes place.

• For example, the int type is always large enough to hold all valid byte values, so no explicit cast statement is required.

• Following shows compatibility of numeric data type:

Integer and Floating-Point - Compatible
Char and Boolean - Incompatible
Also, Boolean and char are not compatible with each other.

1) Explicit Type Conversion (Narrowing Conversion)

• Although the automatic type conversions are helpful, they will not fulfill all

needs. For example, if we want to assign an int value to a byte variable then conversion will not be performed automatically, because a byte is smaller than an int.
• This kind of conversion is sometimes called a narrowing conversion.
• For, this type of conversion we need to make the value narrower explicitly, so that, it will fit into the target data type.
• To create a conversion between two incompatible types, you must use a cast.
• A cast is simply an explicit type conversion.
Synatax:
(target-type) value
• Here, target-type specifies the desired type to convert the specified value to.
• For example, the following casts an int to a byte.
int a;
byte b;
b = (byte) a;
• A different type of conversion will occur when a floating-point value is assigned to an integer type: truncation.
• As we know, integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type,the fractional component is lost.
• For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 will have been truncated.
• When the value 257 is cast into a byte variable, the result is the remainder of the division of 257 by 256 (the range of a byte), which is 1in this case.
• When the d is converted to an int, its fractional component is lost.
• When d is converted to a byte, its fractional component is lost, and the value is reduced modulo 256, which in this case is 67.

An example for Type Conversions:

```
class conversionDemo
{
public static void main(String args[]}
{
byte b;
int i = 257;
double d = 323.142;
System.out.println("\nConversion of int to byte.");
b = (byte) i;
System.out.printlnfi and b 11 + i + n in + b);
System.out.println("\nConversion of double to int.");
i = (int) d;
System.out.printlnf 'd and i " + d + I I I I +i);
```

```
System.out.println(,[\nConversion of double to byte.");
b = (byte) d;
System.out.println("d and b " + d + u H + b;
}
}
```

Wrapper Class

• Wrapper class wraps (encloses) around a data type and gives it an object appearance.
• Wrapper classes are used to convert any data type into an object.
• The primitive data types are not objects and they do not belong to any class.
• So, sometimes it is required to convert data types into objects in java.
• Wrapper classes include methods to unwrap the object and give back the data type.

Example:

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

• The int data type k is converted into an object,itl using Integer class.
• The itl object can be used wherever k is required an object,
• To unwrap (getting back int from Integer object) the object itl.

Example:

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

• intValue() is a method of Integer class that returns an int data type.

• Eight wrapper classes exist in java.lang package that represent 8 data types:

Wrapper class - Unwrap methods
Byte - byteValue()
Short - shortVaiue()
Integer - intValue()
Long - longValue()
Float - floatValue()
Double - doubleValue()
Character - charValue()
Boolean - booleanValue()

• There are mainly two uses with wrapper classes.
1) To convert simple data types into objects.
2) To convert strings into data types {known as parsing operations), here methods of type parseXf) are used. (Ex. parselnt())
• The wrapper classes also provide methods which can be used to convert

a String to any of the primitive data types, except character.
• These methods have the format parsex() where x refers to any of the primitive data types except char.
• To convert any of the primitive data type value to a String, we use the valueOf() methods of the String class.

Example:
int x = Integer.parseInt("34");
double y = Double.parseDouble{"34.7"};
String sl= String.valueOf('a'); // sl="a"
String s2=String.valueOf(true); // s2="true"

Comment Syntax

• Comments are the statements which are never execute, (i.e. non-executable statements).
• Comments are often used to add notes between source code. So that it becomes easy to understand & explain the function or operation of the corresponding part of source code.
• Java Compiler doesn't read comments. Comments are simply ignored during compilation.
• There are three types of comments available in Java as follows:

1. Single Line Comment
• This comment is used whenever we need to write anything in single line.
Syntax:
//'write comment here'

2. Multi Line Comment
• These types of comments are used whenever we want to write detailed notes (i.e. more than one line or in multiple lines) related to source code.
Syntax :
/*
.....
'write comment here'
.....
*/

3. Documentation Comment
• The documentation comment is used commonly to produce an HTML file that documents our program.
• This comment begins with /** and end with a */.
• Documentation comments allow us to embed information about our program into the program itself.
• Then, by using the javadoc utility program to extract the information and

put it into an HTML file.
• In the documentation comment, we can add different notations such as author of the project or program, version, parameters required, information on results in return if any, etc.
• To add these notations, we have '@' operator. We just need to write the required notation along with the operator.
• Some javadoc tags are given below:
@author - To describe the author of the project.
@version - To describe the version of the project.
@param - To explain the parameters required to perform respective operation.
@return - To explain the return type of the project.
Syntax:
/**
*'write comment/description here'
*'©author write author name'
*'©version write version here'
*/

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 handies deallocation for you automatically. The technique that accomplishes this is called garbage collection.
• When no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be recovered.
• Garbage collection only occurs occasionally during the execution of your program.
• Different Java run-time implementations will take varying approaches to garbage collection.

Array

• An array is a group of like-typed variables that are referred to by a common name.
• A specific element in an array is accessed by its index.
• Array index start at zero.
• Arrays of any type can be created and may have one or more dimensions.

1. One Dimensional Array

• Steps To create a one dimensional array:

You must declare a variable of the desired array type.
You must allocate the memory that will hold the array, using new operator and assign it to the array variable.
• In java, all arrays are dynamically allocated.
• Here, datatype specifies the type of data being allocated, size specifies the number of elements in the array, and array_var is the array variable that is linked to the array.
• new is a special operator that allocates memory.
• The elements in the array allocated by new will automatically be initialized to zero.
Syntax:
data_type array_var[];
array_var = new data_type[size];
OR
data_type array_var [] = new data_type[sizej;

An example for one dimentional array:

```
import java.util.*;
class oneDarray
{
public static void main (String args[])
{
int a[] = new int[3];
a[0] = 1;
a[1] = 2;
a[2] = 3;
System.out.println("Your array elements are: "+ a[0]+" "+a[1]+" "+a[2]);
}
}
```

2. Multidimensional Array
• Multidimensional arrays are actually arrays of arrays.
• To declare a multidimensional array variable, specify each additional index using another set of square brackets.
• This allocates a 3 by 5 array and assigns it to a. Internally, this matrix is implemented as an array of arrays of int
Syntax :
data_type array_var [][] = new data_type[size][size];

An example for two dimentional array:

```
import java.util.*;
class twoDarray
{
```

```
public static void main (String args[])
{
int a[][] = new int[2][2];
a[0][0] = 1;
a[0][1] = 2;
a[1][0] = 3;
a[1][1] = 4;
System.out.println("Your array elements are :: "+ a[0][0]+" "+a[0][l]+"
"+a[l][0]+" "+a[l][l]);
} }
```

String

• Strings are widely used in JAVA Programming, are not only a sequence of characters but it defines object.
• String Class is defined in java.lang package.
• The String type is used to declare string variables. Also we can declare array of strings.
• A variable of type String can be assign to another variable of type String.
Example:

```
String si = "Welcome To Java String";
System.out.println(sl);
```

• Here, si is an object of type String.
• String objects have many special features and attributes that makes them powerful.
String class has following features:
• It is Final class
• Due to Final,String class cannot be inherited.
• It is immutable.

Example:

```
import java.io.*;
class stringDemo
{
public static void main(String args[])
{
String str = "Hello";
System.out.println(str.!ength(J);
if(str.equals("BYE")}
{
System,out.println("Same");
}
else
{
```

```
System.out.println("Not Same");
}
if ( str,compareTo("YES") > 0)
{
System,out.println("YES is greater than Hello");
}
else
{
System,out.printlnf("Hello is greater than YES");
}
System.out.println( str.substring{1,3});
}
}
```

StringBuffer Class: • Java StringBuffer class is a thread-safe, mutable sequence of characters.
• Every string buffer has a capacity.
• It contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.

Example:

```
import java.io.*;
class stringBufferDemo
{
public static void main(String args[])
{
StringBuffer sb1 = new StringBufferf("DIET");
StringBuffer sb2 = new StringBuffer(lOO);
System.out.println{sb1 : " + strBufl);
System.out.println{"sb1 capacity : " + sb1.capacityO);
System.out.println{"sb2 capacity : " + sb2.capacityO);
System.out.println{"sb1 reverse : " + sb1.reverse());
System.out.println{"sb1 charAt 2 : " +sb1.charAt{ 2});
System.out.println{"sb1 toString() i s : " + sb1.toStringO);
strBufl.append("String 2");
System.out.println{"sb3 when appended with a String: "+ sb1);
}
}
```

Selection Statement

1. If statement

• if statement consists of a condition followed by one or more statements.
Syntax:

```
if (condition)
{
//Statements will execute if the Boolean expression is true
}
```
• If the condition is true then the block of code inside if statement will be executed.

Example:

```
class ifDemo
{
public static void main($tring[] args)
{
int marks = 76;
String grade = null;
if (marks >= 40)
{
grade = "Pass";
}
if (marks < 40)
{
grade = "Fail";
}
System.out.println("Grade = " + grade);
}
}
```

2. If ... else Statement

• The if ...else statement is Java's conditional branch statement.
• Here is the general form of the if..else statement:

Syntax:

```
if (condition)
statementl;
else
statement2;
```

• Here, each statement may be a single statement or a compound statement enclosed in curly braces (that is, a block).
• The condition is any expression that returns a boolean value. The else clause is optional.
• The if..else works as follow: If the condition is true, then statementl is executed. Otherwise, statement2 (if it exists) is executed.

Example:

```
class ifElseDemo
{
public static void main(String[] args)
{
int marks = 76;
String grade;
if (marks >= 40)
{
grade = "Pass";
}
else
{
grade = "Fail";
}
System.out.println("Grade = " + grade);
}
}
```

3. Switch Statement

• The switch statement is Java's multiway branch statement. It provides an easy way to execute different parts of your code based on the value of an expression.
• Here is the general form of a switch statement:
Syntax:

```
switch (expression) {
case valuel: // statement sequence
break;
case value2:
// statement sequence
break;
case valueN:
// statement sequence
break;
default:
// default statement sequence
}
```

• The expression must be of type byte, short, int or char.
• Each case value must be a unique literal (that is, it must be a constant, not a variable). Duplicate case values are not allowed.
• The switch statement works as follow: The value of the expression is compared with each of the literal values In the case statements. If a match is found, the code sequence following that case statement is executed.
• If none of the literal matches the value of the expression, then the default statement is executed. However,the default statement is optional.

• The break statement is used inside the switch to terminate a statement sequence.

Example:

```
class switchDemo
{
public static void main(String[] args)
{
int day = 8;
switch (day)
{
case 1: System.out.printlnf("Monday"); break;
case 2: System.out.println("Tuesday"); break;
case 3: System.out.println("Wednesday"); break;
case 4: System.out.printlnf("Thursday"); break;
case 5: System.out.println("Friday"); break;
case 6: System.out.printlnf("Saturday"); break;
case 7: System.out.printlnf("Sunday"); break;
default: System.out.printlnpnvalid Day");break;
}
}
}
```

Iteration Statement

• Java's iteration statements are for, while, and do-while.
• These statements commonly call loops.
• Loop repeatedly executes the same set of instructions until a termination condition is met.

1. While loop

• It repeats a statement or block until its controlling expression is true.
Syntax:

```
while(condition)
{
// body of loop
}
```

• The condition can be any boolean expression. The body of the loop will be executed as long as the conditional expression is true.
• When condition becomes false, control passes to the next line of code immediately following the loop.

Example:

```
class whileDemo
{
public static void main($tring[] args)
{
int x =1; while( x < 5 ) { System.out.print(x+" " ); x++;
}
}
}
```

2. Do-while loop

• Sometimes it is desirable to execute the body of a loop at least once, even if the conditional expression is false.
• The do-while loop always executes its body at least once, because its conditional expression is at the bottom of the loop.
Syntax:

```
do
{
// body of loop
}
while (condition);
```

• Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates. Condition must be a boolean expression.

Example:

```
class dowhileDemo
{
public static void main($tring[] args)
{
int x = 1;
do
{
System.out.print(x+" ");
x++;
}
while( x < 5);
}
}
```

3. For loop

Syntax:

```
for(initialization; condition; iteration)
```

```
{
// body
}
```
• When the loop first starts, the initialization portion of the loop is executed. Generally, this is an expression that sets the value of the loop control variable, which acts as a counter that controls the loop. The initialization expression is only executed once.
• Next, condition is evaluated. This must be a boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates.
• Next, the iteration portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable.
• The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

Example:

```
class forDemo
{
public static void main($tring[] args)
{
for{int x = 1; x < 5; x++ )
{
System.out.print(x+" ");
}
}
}
```

4. Jump Statement

• These statements transfer control to another part of your program.
break
• 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.

Example:

```
class breakDemo
{
public static void main($tring[] args)
{
```

```
int [] numbers = {10, 20, 30, 40,50};
for(int x=0;x<=5;x++)
if( numbers[x] == 30 )
{
break;
}
System.out.print( numbers[x] +" ");
}
}
}
```

5. Continue statement

• Continue statement is used when we want to skip the rest of the statement in the body of the loop and continue with the next iteration of the loop.

Example:

```
class conDemo
{
public static void main($tring[] args)
{
int [] numbers = {10, 20, 30, 40, 50};
for(int x=0;x<=5;x++)
{
if( numbers[x] == 30 )
{
continue;
}
System.out.printf numbersfx] + ");
}
}
}
```

6. Return statement

• The return statement is used to explicitly return from a method. It transfers control of program back to the caller of the method.
• The return statement immediately terminates the method in which it is executed.

Example:

```
class test
{
```

```
public static void main($tring[] args)
{
int day = 8;
switch (day)
{
int a=10, b=20, c;
c=add(a,b);
System.out.println("Addition="+c);
}
public static int add(int x, int y)
{
return x+y;
}
}
```

Object Oriented Programming Concepts

Class

• A class is a template that specifies the attributes and behavior of things or objects.
• A class is a blueprint or prototype from which objects are created.
• A class is the implementation of an abstract data type (ADT). It defines attributes and methods which implement the data structure and operations of the ADT, respectively.

Syntax:

```
class classname
{
Datatype variable1;
Datatype variable2;
Datatype variableN;
return_type methodnamel(parameter-list)
{
// body of method
}
return_type methodname2(parameter-list)
{
// body of method
}
return_type methodnamelM(parameter-list)
{
// body of method
}
}
```

• A class is declared by use of the class keyword.
• The data, or variables, defined within a class are called instance variables because eachinstance of the class (that is, each object of the class) contains its own copy of these variables. Thus,the data of one object is separate and unique from the data of another.
• The actual code contained within methods.
• The methods and variables defined within a class are called members of the class.

Return value

• Method that have a return type other than void return a value to the calling method using the following form of the return statement:
Syntax:
return value;
• Here, value is the value returned.

Example:

```
class Box
{
double width =1;
double height = 2;
double depth = 3;
double volume()
{
return (width * height * depth);
}
}
```

Method Call

Syntax:
var name object_name.method_name(parameter-list};
Example:
vol =bl.volumeQ;
• In above example, bl is an object and when volume{ } is called, it is put on the right side of an assignment statement. On the left is a variable, in this case vol, that will receive the vaiue returned by volume( ).

Declaring Object

• To obtain an object of class, first you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object.
• Second, you must obtain an actual copy of the object and assign it to that variable. You can do this using the new operator.
• The new operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference Is an address in memory of the object allocated by new.
Syntax:
Step - 1: class_name class_var;
Step - 2: classvar = new class_name();
• Here, class_var is a variable of the class type being created. The class_name is the name of the class that is being instantiated.

Assigning Object Reference Variables

• Object reference variables acts differently than you might expect when an assignment takes place.
Example:
Box bl = new Box();
Box b2 = bl;
Here, bl and b2 will both refer to the same object. The assignment of bl to b2 did not allocate any memory of the original object.
It simply makes b2 refer to the same object as bl does. Thus, any changes made to the object through b2 will affect the object to which bl is referring, since they are the same object.

Example:

```
class Box
{
double width = 1.0;
double height = 2.0;
double depth = 3.0;
void volume()
{
System.out.print("Volume is ");
System.out.println(width * height * depth);
}
}
class BoxDemo
{
public static void main{String args[]) {
Box bl;
bl = new Box()
//Box bl = new Box();
bl.volume();
// declare reference to object, allocate a Box objectt
}
}
```

Visibility controls of JAVA

• Java provides a number of access modifiers to set access levels for classes, variables, methodsand constructors. The four access levels are:
1) Package/friendly (default) -Visible to the package. No modifiers are needed.
2) Private - Visible to the class only.
3) Public- Visible to the class as well as outside the class.
4) Protected- Visible to the package and all subclasses.
Default Access Modifier - No Keyword • Default access modifier means no

need to 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 default modifier cannot be used for methods in an interface because the methods in an interface are by default public.

Example:

```
String str = "Hi";
void a()
{
System.out.println(str);
}
```

Private Access Modifier

• 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, an object encapsulates itself and hides data from the outside world.

Example:

```
class A
{
private String si = "Hello";
public String getName()
{
return this.si;
}
}
```

Here,si variable of A class is private, so there's no way for other classes to retrieve. So, to make this variable available to the outside world, we defined public methods: getName{), which returns the value of si.

Public Access Modifier

• The public keyword is an access specifier, which allows the programmer to control the visibility of class members.
• When a class member is preceded by public, then that member may be accessed by code outside the class.
• A class, method, constructor, interface etc declared public can be accessed from any other class.
Therefore, methods or blocks declared inside a public class can be

accessed from any class belonging to the Java world.
However, if the public class we are trying to access is in a different package, and 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:

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

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

Protected Access Modifier

Variables, methods and constructors which are declared protected in a super class 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 can be declared protected, however methods in a interface cannot be declared protected.
Protected access gives chance to the subclass to use the helper method or variable, while prevents a non-related class from trying to use it.

Example:

```
package pi;
class A
{
float fl;
protected intil;
} //class B belongs to the same package as class A
package pi;
class B
{
public void getData()
{ // create an instance of class A
A al= new A();
al.fl= 19;
al.il = 12;
}
}
```

this Keyword

• Sometimes a method will need to refer to the object that invoked it.
• To allow this, Java defines the this keyword. Keyword this can be used inside any method or constructor of class to refer to the current object.
• It means,this is always a reference to the object on which the method was invoked.
• this keyword can be very useful in case of Variable Hiding.
• You can use this anywhere a reference to an object of the current class' type is permitted.
• We cannot create two Instance/Local variables with same name. But it is legal to create one instance variable & one local variable or method parameter with same name.
• Local Variable will hide the instance variable which is called Variable Hiding.

Example:

```
class A
{
int v = 5;
public static void main(String args[])
{
A al = new A();
al.method(20);
al.method();
}
void methodfint variable()
{
int v = 10; System.out.println("Value of Instance variable + this.v);
System.out.println("Value of Local variable + v);
}
void method()
{
int v = 40;
System.out.println('[Value of Instance variable + this.v);
System.out.printlnC'Value of Local variable + v);
) }
```

static Keyword

• 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.

• 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:
1) They can only call other static methods.
2) They must only access static data.
3) They cannot refer to this or super in any way.
• If you wish to call a static method from outside its class, you can do so using the following general form:
A fi A classname.method( );
• Here, classname is the name of the class in which the static method is declared. No need to call static method through object of that class.

Example:

```
class staticDemo
{
static int count=0; //will get memory only once and retain its value
staticDemo()
{
count++;
System.out.println(count);
}
static
{
System.out.println("Static block initialized...");
}
static void display()
{
System.out.println("Static method call...");
}
public static void main(String args[j])
{
staticDemo sl=new staticDemoO;
staticDemo s2=new staticDemoO;
staticDemo s3-new staticDemoO;
display();
}
}
```

final Keyword

• A variable can be declared as final. You cannot change the value of final variable. It means, final variable act as constant and value of that variable can never be changed.
• If you declared any method as final then you cannot override it.

• If you declared any class as final then you cannot inherit it.

Example:

```
class finalDemo
{
final int b = 100;
void ml()
{
b = 200; /* Error generate because we cannot change the value of final
variable*/
}
public static void main(String args[])
{
finalDemo fl = new finalDemo();
fl.mlQ;
}
}
```

Method Overloading

• If class have multiple methods with same name but different parameters is known as Method Overloading.
• Method overloading is also known as compile time (static) polymorphism.
• The same method name will be used with different number of parameters and parameters of different type.
• Overloading of methods with different return types is not allowed.
• Compiler identifies which method should be called among all the methods have same name using the type and number of arguments.
• However, the two functions with the same name must differ in at least one of the following.
1) The number of parameters
2) The data type of parameters
3) The order of parameter

Example:

```
class overloadingDemo
{
void sum(int a,int b)
{
System.out.println("Sum of (a+b) is:: "+(a+b));
}
void sumfint a,int b,int c)
{
System.out.println("Sum of (a+b+c) is:: ”+(a+b+c));
```

```
}
void sum{double a,double b)
{
System.out.println(,rSum of double (a+b) is:: "+(a+b});
}
public static void main(String args[])
{
overloadingDemo ol = new overloadingDemo();
ol.sum{10,10);
ol.sum(10,10,10);
ol.sum(10.5,10.5);
// call methodl
// call method2
// call method3
}
}
```

Constructor

• Constructor is special type of method that is used to initialize the object.
• It is invoked at the time of object creation.
• There are two rules to define constructor as given below:
1) Constructor name must be same as its class name.
2) Constructor must not have return type.
• Return type of class constructor is the class type itself.
• There are two type of constructor :
1) Default Constructor
2) Parameterized constructor

Example:

```
class A
{
A() //Default constructor
{
System.out.println("Default constructor called..");
}
public static void main(String args[])
{
A a = new A();
}
}
```

Parameterized Constructor:

• A constructor that has parameters is known as parameterized constructor.

• It is used to provide different values to the distinct objects.
• It is required to pass parameters on creation of objects.
• If we define only parameterized constructors, then we cannot create an object with default constructor. This is because compiler will not create default constructor. You need to create default constructor explicitly.

Example:

```
class A
{
int a;
Stringsi;
A(int b,String s2) //Parameterized constructor
{
b = a;
s2 - si;
}
void display()
{
System.out.println("Value of parameterized constructor is: "fa^1 and "+bj;
}
public static void main(String args[])
{
A a = new A(10,"Hello");
a.displayO;
}
}
```

Copy Constructor

• A copy constructor is a constructor that takes only one parameter which is the same type as the class in which the copy constructor is defined.
• A copy constructor is used to create another object that is a copy of the object that it takes as a parameter, But, the newly created copy is totally independent of the original object.
• It is independent in the sense that the copy is located at different address in memory than the original.

Overloading Constructor

• Constructor overloading in java allows to more than one constructor inside one Class.
• It is not much different than method overloading. In Constructor overloading you have multiple constructors with different signature with only difference that constructor doesn't have return type.

• These types of constructor known as overloaded constructor.

Passing object as a parameter

• If you want to construct a new object, thatis initially the same as some existing object. To do this, you must define a constructor that takes an object of its class as a parameter.

Example:

```
class Box
{
double width;
double height;
double depth;
// It takes an object of type Box. Copy constructor
BoxfBox obj;
{
// pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
} //Parameterized constructor
Box(double w, double h, double d)
{
width = w;
height = h;
depth = d;
}// Default constructor
Box()
{
width = -1; // use -1 to indicate
height =1;// an uninitialized
depth = -1;// box
}
//constructor used when cube is created
Box(double len)
{
width = height = depth = len;
}
compute and return volume
double volume()
{
return width * height * depth;
}
```

```
}
class DemoAIICons
{
public static void main{$tring args[])
{
Box myboxl = new Box(10,20,15);
Box mybox2 = new Box();
Box mycube = new Box(7);
Box myclone = new Box(myboxl); // create copy of myboxl
double vol;
// get volume of first box
vol = myboxl.volumeO;
System.out.println("Volume of myboxl is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
// get volume of cube
vol = mycube.volume();
System.out.println("Volume of cube is " + vol);
// get volume of clone
vol = myclone.volumeQ;
System.out.println("Volume of clone is 11 + vol);
}
}
```

Inheritance, Packages & Interfaces

Inheritance

• Inheritance is the process, by which class can acquire the properties and methods of its parent class.
• The mechanism of deriving a new child class from an old parent class is called inheritance.
• The new class is called derived class and old class is called base class.
• 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.
• All the properties of superclass except private properties can be inherit in its subclass using extends keyword.

Byte Streams

Types of Inheritance

Single Inheritance
• If a class is derived from a single class then it is called single inheritance.
• Class B is derived from class A.

Multilevel Inheritance
• A class is derived from a class which is derived from another class then it is called multilevel inheritance
• Here, class C is derived from class B and class B is derived from class A, so it is called multilevel Inheritance.

Multiple Inheritance
• If one class is derived from more than one class then it is called multiple inheritance.
• It is not supported in java through class.

Hierarchical Inheritance
• If one or more classes are derived from one class then it is called hierarchical inheritance.
• Here, class B, class C and class D are derived from class A.

Hybrid Inheritance
• Hybrid inheritance is combination of single and multiple inheritance.
• But java doesn't support multiple inheritance, so the hybrid inheritance is also not possible.

Example of Hybrid Inheritance:

```
class A
{
public void displayA()
{
System.out.println("class A method");
}
}
class B extends A //Single Inheritance - class B is derived from class A
{
public void displayB( )
{
System.out.println("class B method");
}
}
class C extends B // Multilevel Inheritance - class C is derived from class B
{
public void display()
{
System.out.println("rclass C method");
}
}
class D extends A //Hierarchical Inheritance - Class B and Class D are
derived from Class A
{
public void displayD()
{
System.out.println("class D method");
}
}
class Trial
{
public static void main(String []args)
{
B b=new B();
C c=new C();
D d=new D();
b.displayB();
c.displayC();
d.dispiayD();
} }
```

Method Overriding

• When a method in a subclass has the same name and type signature as method in its superclass, then the method in the subclass is said to

override the method of the super class.
• The benefit of overriding is: Ability to define a behavior that's specific to the subclass type. Which means a subclass can implement a superclass method based on its requirement.
• Method overriding is used for runtime polymorphism.
• In object oriented terms, overriding means to override the functionality of any existing method.
• Because, at compile time reference type of object is checked. However, at the runtime JVM determine the object type and execute the method that belongs to that particular object.

Example:

```
class A
{
public void display()
{
System.out.println("Class A");
}
}
class B extends A
{
public void display()
{
System.out.println("Class B");
}
}
class Trial
{
public static void main(String args[])
{
A a = new A();// A reference and object
A b = new B();// A reference but B object
a.displayO;// Runs the method in A class
b.displayO;// Runs the method in B class
}
}
```

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 as the return type declared in the original overridden method in the superclass.
• 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.
• Constructors cannot be overridden.

super keyword

• The super keyword in java is a reference variable that is used to refer immediate parent class object.

Example:

```
class A
{
A()
{
System.out.println("Super class default constructor called.,");
}
A(String si)
{
System.out.println("Super class parameterized constructor called: "+sl);
}
}
class B extends A
{
/* Implicitly default constructor of superclass(A) will be called. Whether you define
super or not in subclass(B) constructor * /
B()
{
System.out.println("Sub class default constructor called..1");
}
/* To call a parameterized constructor of superclass(A) you must write super() with
same number of arguments*/
B(String si)
{
super("Class A 1");
System.out.println("Sub class parameterized constructor called: " + si);
}
```

```
}
class SuperConDemo
{
public static void main{String args[])
{
B bl = new B();
B b2 = new B("Class B");
}
}
```

super To Call super-class Constructor

• Every time a parameterized or non-parameterized constructor of a subclass is created, by default a default constructor of superclass is called implicitly.
• The syntax for calling a superclass constructor is:
super();
OR
super(parameter list);
• The following example shows how to use the super keyword to invoke a superclass's constructor.

Example:

```
class A
{
A()
{
System.out.println("Super class default constructor called.,");
}
A(String si)
{
System.out.println("Super class parameterized constructor called: "+sl);
}
}
class B extends A
{
/* Implicitly default constructor of superclass(A) will be called. Whether you define
super or not in subclass(B) constructor * /
B()
{
System.out.println("Sub class default constructor called..1");
}
```

```
/* To call a parameterized constructor of superclass(A) you must write
super() with
same number of arguments*/
B String si()
{
super("Class A 1");
System.out.println("Sub class parameterized constructor called: " + si);
}
}
class SuperConDemo
{
public static void main(String args[])
{
B bl = new B();
B b2 = new B("Class B");
}
}
```

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.
• It is also known as run-time polymorphism.
• A superclass reference variable can refer to a subclass object. It is known as Upcasting.
• When an overridden method is called through a superclass reference, the determination of the method to be called is based on the object being referred to by the reference variable
• This determination is made at run time.

Example:

```
class A
{
void callme()
{
System.out.println("Inside A's callme method");
}
}
class B extends A
{
void callme()
{
System.out.println("Inside B's callme method");
}
```

```
}
class C extends A
{
void callme()
{
System.out.println("Inside C's callme method");
}
}
class DispatchDemo
{
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 r a reference of type A
r = a;// r refers to an A object
r.callme();// 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()
}
}
```

Object Class

• The Object class is the parent class (java.lang.Object) of all the classes in java by default.
• The Object class provides some common behaviors to all the objects must have, such as object can be compared, object can be converted to a string, object can be notified etc..
• Some Java object class methods are given below:

Method Description

equals() -- To compare two objects for equality.
getClass() -- Returns a runtime representation of the class of this object. By using this class object we can get information about class such as its name, its superclass etc.
toString() -- Returns a string representation of the object.
notify() -- Wakes up single thread, waiting on this object's monitor.
notifyAll() -- Wakes up all the threads, waiting on this object's monitor.
wait() -- Causes the current thread to wait, until another thread notifies.

Example:

```
class parent
{
int i = 10;
Integer il= new Integer(i);
void PrintClassName(Object obj) // Pass object of class as an argument
{
System.out.println("The Object's class name is +
obj.getClass().getName());
}
}
class ObjectCiassDemo
{
public static void main(String args[])
{
parent al = new parent();
al.PrintClassName(al);
System.out.println("String representation of object il is:: "+al.il.toString());
}
}
```

Packages

• A java package is a group of similar types of classes, interfaces and sub-packages.
• Packages are used to prevent naming conflicts and provides access protection.
• It is also used to categorize the classes and interfaces, so that, they can be easily maintained.
• We can also categorize the package further by using concept of subpackage. Package inside the package is called the subpackage.

Package can be categorized in two form:
• built-in packages:
Existing Java package such as java.lang, java.util, java.io, java.net, java.awt.
• User-defined-package:
Java package created by user to categorized classes and interface.
• Programmers can define their own packages to bundle group of classes, interfaces etc.

Creating a package

• To create a package, package statement followed by the name of the

package.
• The package statement should be the first line in the source file. There can be only one package statement in each source file.
• If a package statement is not used then the class, interfaces etc. will be put into an unnamed.
package.

Example:

```
package mypack;
class Book
{
String bookname;
String author;
Book()
{
bookname = "Complete Reference";
author = "Herbert";
}
void show()
{
System.out.println("Book name is: "+bookname+,"nand author name is:
"+author);
}
}
class DemoPackage
{
public static void main(String[] args)
{
Book bl = new Book();
bl.show();
}
}
```

Import Package

• Import keyword is used to import built-in and user-defined packages into your java source file.
• If a class wants to use another class in the same package, no need to import the package.
• But, if a class wants to use another class that is not exist in same package then import keyword is used to import that package into your java source file.
• A class file can contain any number of import statements.
• There are three ways to access the package from outside the package:

1) Using packagename.* :
• If you use packagename.* then all the classes and interfaces of this package will be accessible but not subpackages.
Syntax: import packagename.*;

2) Using packagename.classname:
• If you use packagename.classname then only declared class of this package will be accessible.
Syntax:
import packagename.classname;

3) Using fully qualified name:
• If you use fully qualified name then only declared class of this package will be accessible.
• So, no need to import the package.
• 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.

Interfaces

• An interface is a collection of abstract methods. An interface is not a class.
• When you create an interface it defines what a class can do without saying anything about how the class will do it.
• Interface contains only 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.
• By default (Implicitly), an interface is abstract, interface fields(data members) are public, static and final and methods are public and abstract.
• It is used to achieve fully abstraction and multiple inheritance in Java.
• Similarity between class and interface are given below:
• An interface can contain any number of methods.
• An interface is written in a file with a .java extension, with the name of the interface matching the name of the file.
• The bytecode of an interface appears in a .class file

Declaring Interfaces

• The interface keyword is used to declare an interface.

Syntax:
NameOfInterface.java
import java.lang.*;

```
public interface NameOfInterface
{
//Any number of final, static fields
//Any number of abstract method declarations
}
```

Example:
DemoInterface.java

```
interface DemoInterface
{
inti = 10;
void demo();
}
```

• In above example, name of interface is DemoInterface and it contains a variable i of integer type and an abstract method named demo(). Implementing Interfaces • A class uses the implements keyword to implement an interface.
• A class implements an interface means, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface.
• If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

Inheritance on Interfaces

• We all knows a class can extend another class. Same way an interface can extend another interface.
• The extends keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

Example:

```
public interface A
{
void getdata(String name);
}
public interface B extends A
{
void setdata();
}
class InheritInterface implements B
{
String display;
public void getdata{String name)
```

```
{
display = name;
}
public void setdata()
{
System,out.println(display);
}
public static void main(String args[])
{
InheritInterface obj = new InheritInterface();
obj.getdata("Welcome TO Heaven");
obj.setdata();
}
}
```

Multiple Inheritance using Interface

• If a class implements multiple interfaces, or an interface extends multiple interfaces known as multiple inheritance.
• A java class can only extend one parent class. Multiple inheritances are not allowed. However, an interface can extend more than one parent interface.
• The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

Example:

```
public interface A
{
void getdata(String name);
}
public interface B
/* Here we can also extends multiple interface like interface B extends A,C
*/
{
void setdata();
}
class InheritInterface implements A, B
{
String display;
public void getdata(String name)
{
display = name;
}
public void setdata()
```

```
{
System.out.println(display);
}
public static void main(String args[])
{
InheritInterface obj = new InheritInterface();
obj.getdata("Welcome TO Heaven");
obj.setdata();
}
}
```

Abstract Class

• A class that is declared with abstract keyword, is known as an abstract class in java.It can have abstract and non-abstract methods (method with body).
• It needs to be extended and its method implemented. It cannot be instantiated means we can't create object of it.
• Any class that extends an abstract class must implement all the abstract methods declared by the super class.

Syntax:

```
abstract classdass name
{
//Number of abstract as well as non-abstract methods.
}
```

• A method that is declared as abstract and does not have implementation is known as abstract method.
• The method body will be defined by its subclass. Abstract method can never be final and static.

Syntax:

```
abstract return_typefunction_name (); // No definition
```

Example:

```
abstract class A
{
abstract void abs_method();
public void display()
{
System.out.println("This is concrete method..");
}
}
```

```
class DemoAbstract extends A
{
void abs_method()
{
System.out.println("This is an abstract method..");
}
public static void main(String[j] args)
{
DemoAbstract abs = new DemoAbstract();
abs.abs_method();
abs.display();
}
}
```

Final Keyword

• The final keyword in java is used to restrict the user. The java final keyword can be used with:
1) variable
2) method
3) class

• Final Variable:
If you make any variable as final, you cannot change the value of that final variable (It will be constant).
A variable that is declared as final and not initialized is called a blank final variable. A blank final variable forces the constructors to initialize it.

• Final Method:
Methods declared as final cannot be overridden.

• Final Class:
Java classes declared as final cannot be extended means cannot inherit.
If you declare any parameter as final, you cannot change the value of it.

Example:

```
class DemoBase
{
final int i =1; //final variable must be initialize.
final void display()
{
System.out.println("Value of i is "+i);
}
```

```
}
class DemoFinal extends DemoBase
{
/'“void display() // Compilation error final method cannot override.
{
System.out.println("Value of i is v- > mm "+i);
}
public static void main(String args[])
{
DemoFinal obj = new DemoFinal();
obj.display();
}
}
```

Exceptions Handling

• An exception (or exceptional event) is a problem that arises during the execution of a program.
• When an Exception occurs the normal flow of the program is disrupted and thenprogram/Appiication terminates abnormally,therefore these exceptions are needs to be handled.
• A Java Exception is an object that describes the exception that occurs in a program. When annexceptional event occurs in java, an exception is said to be thrown.
• An exception can occur for many different reasons, some of them are as given below:
• A user has entered invalid data.
• A file that needs to be opened cannot be found.
• A network connection has been lost in the middle of communications, or the JVM has run out of memory.
• Exceptions are caused by users, programmers or when some physical resources get failed.
• The Exception Handling in java is one of the powerful mechanisms to handle the exceptionn(runtime errors), so that normal flow of the application can be maintained.

• In Java there are three categories of Exceptions:

1) Checked exceptions: A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions. Example, lOException, SQLException etc.

2) Runtime exceptions: An Unchecked exception is an exception that occurs during the execution, these are also called as Runtime Exceptions. These include programming bugs, such as logic errors or improper use of an API.
Runtime exceptions are ignored at the time of compilation.
Example :
ArithmeticExceptlon, NullPointerException, Array Index out of Bound exception.

3) Errors: These are not exceptions at all, but problems that arise beyond the control of the user or the programmer.
Example:
OutOfMemoryError, VirtuaiMachineErrorException.
• All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from

the Throwable class.
• Errors are not normally trapped form the Java programs. These conditions normally happen in case of severe failures, which are not handled by the java programs. Errors are generated to indicate errors generated by the runtime environment.
Example:
JVM is out of Memory. Normally programs cannot recover from errors.
• The Exception class has two main subclasses: lOException class and RuntimeException class.

Exception Handling Mechanism

• Exception handling is done using five keywords:

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

1) 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.

2) catch block :
• Java catch block is used to handle the Exception. It must be used after the try block only.
• The catch block that follows the try is checked, if the type of exception that occurred is listed in the catch block then the exception is handed over to the catch block that handles it.
• You can use multiple catch block with a single try.

Syntax:
```
try
{
//Protected code
}
catch(ExceptionNamel el)
{
//Catch block 1
}
catch(ExceptionName2 e2)
{
//Catch block 2
}
```

• In above syntax, there are two catch blocks. In try block, we write code that might generate exception. If the exception generated by protected code then exception thrown to the first catch block.
• If the data type of the exception thrown matches ExceptionNamel, it gets caught there and execute the catch block.
• If not, the exception passes down to the second catch block.
• This continues until the exception either is caught or falls through all catches, in that case the current method stops execution.

Nested Try-Catch Blocks

• In java, the try block within a try block is known as nested try block.
• Nested try block is used when a part of a block may cause one error while entire block may cause another error.
• In that case, if inner try block does not have a catch handler for a particular exception then the outer try is checked for match.
• This continues until one of the catch statements succeeds, or until the entire nested try statements are done in. If no one catch statements match ,then the Java run-time system will handle the exception.

Syntax:

```
try
{
Statement 1;
try
{
//Protected code
}
catch(ExceptionName el)
{
//Catch blockl
}
}
catch(ExceptionName e2)
{
//Catch block 2
}
```

Key points to keep in mind:
• A catch clause cannot exist without a try statement.
• It is not compulsory to have finally clause for every try/catch.
• The try block cannot be present without either catch clause or finally clause.
• Any code cannot be present in between the try, catch, finally block.

4) throw :

• The throw keyword is used to explicitly throw an exception.
• We can throw either checked or uncheked exception using throw keyword.
• Only object of Throwable class or its sub classes can be thrown.
• Program execution stops on encountering throw statement, and the closest catch statement is checked for matching type of exception.

Syntax:
throw ThrowableInstance;

5) throws:

• The throws keyword is used to declare an exception.
• if a method does not handle a checked exception, the method must declare it using the throws keyword. The throws keyword appears at the end of a method's signature.
• You can declare multiple exceptions.

Syntax:
return_type method_name() throws exception_class_name_list
{
//method code
}

User Defined Exception

• In java, we can create our own exception that is known as custom exception or user-defined exception.
• We can have our own exception and message.
Key points to keep in mind:
• All exceptions must be a child of Throwable.
• If you want to write a checked exception that is automatically enforced by the Declare Rule, you need to extend the Exception class.
• If you want to write a runtime exception, you need to extend the RuntimeException class.

Example:

```
class demoUserException extends Exception
{
private int ex;
demoUserException(int a )
{
ex=a;
}
public String toString()
```

```
{
return irMyException[" + ex +"] is less than zero";
}
}
class demoException
{
static void sumfint a,int b) throws demoUserException
{
if (a<=0)
{
throw new demoUserException (a);
}
else
{
System.out.println(a+b);
}
}
public static void main(String[] args)
{
try
{
sum(-10,10); }
catch(demollserException e)
{
System.out.println(e);
}
}
}
```

Multithreading

• Java is a multithreaded programming language which means we can develop multi threaded program using Java.
• Multithreaded programs contain two or more threads that can run concurrently. This means that a single program can perform two or more tasks 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 which saves memory, and context-switching between the threads takes less time than process.
• Threads are independent. So, it doesn't affect other threads if exception occurs in a single thread.

• Java Multithreading is mostly used in games, animation etc.
• For example, one thread is writing content on a file at the same time another thread is performing spelling check.
• In Multiprocessing, Each process has its own address in memory. So, each process allocates separate memory area.
• Switching from one process to another require some time for saving and loading registers, memory maps,updating lists etc. So that cost of communication between the processes is high.
• Disadvantage: If you create too many threads, you can actually degrade the performance of your program rather than enhance it.
• Remember, some overhead is associated with context switching. If you create too many threads, more CPU time will be spent changing contexts than executing your program.

Life Cycle of Thread

• A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies.

• New:
A new thread begins its life cycle in the New state. It remains in this state until the program starts the thread by invoking Start() method.It is also referred to as a born thread.

• 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.
The thread is in running state if the thread scheduler has selected it.

• Waiting:
Sometimes a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

• Timed waiting:
A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transition back to the runnable state when that time interval expires or when the event It is waiting for occurs.

• Terminated:
A runnable thread enters the terminated state when it (run() method exits) completes its task.

Thread Priorities

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.
• Java priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10).
By default, every thread is given priority NORM_PRIORITY (a constant of 5).
• Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads.
• The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

Creating a Thread
• Thread class provide constructors and methods to create and perform operations on a thread.

1) Constructor of Thread class:
Thread ()
Thread (String name)
Thread (Runnable r)
Thread (Runnable r, String name)

2) Methods of Thread Class:
public void run() -- Entry point for a thread
public void start() -- Start a thread by calling run() method.
public String getName() -- Return thread's name.
public void setName(String name) -- To give thread a name
public int getPriority() -- Return thread's priority.
public int setPriority(int priority) -- Sets the priority of this Thread object. The possible values are between1and 10.
public final boolean isAlive() -- Checks whether thread is still running or not.
public static void sleep(long millisec) -- Suspend thread for a specified time.
public final void join(long millisec) -- Wait for a thread to end.

• Java defines two ways by which a thread can be created.

1) By implementingthe Runnable interface:
• The easiest way to create a thread is to create a class that implements the runnable interface.
• After implementing runnable interface, the class needs to implement the run{} method, which has following form:
public void run( )
• This method provides entry point for the thread and you will put you complete business logic inside this method.
• After that, you will instantiate a Thread object using the following constructor:

• Thread (Runnable threadObj,String threadName);
• Where, threadObj is an instance of a class that implements the Runnable interface and threadName is the name given to the new thread.
• Once Thread object is created, you can start it by calling start( ) method, which executes a call to run( ) method.
void start ( );

2) By extending the Thread class.
• Second way to create a thread is to create a new class that extends Thread class and then create an instance of that class.
• The extending class must override the run( ) method, which is the entry point for the new thread.
• Once Thread object is created, you can start it by calling startf) method, which executes a call to run( ) method.

Thread Synchronization

• When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.
• The process by which this synchronization achieved is called thread synchronization.
• The synchronized keyword in Java creates a block of code referred to as a critical section.
• Every Java object with a critical section of code gets a lock associated with the object.
• To enter a critical section, a thread needs to obtain the corresponding object's lock.

Syntax:
```
synchronized(object)
{
// statements to be synchronized
}
```
Here, object is a reference to the object being synchronized.
A synchronized block ensures that a call to a method that is a member of object occurs only after the current thread has successfully entered object's critical section.

Inter-thread Communication

• Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.
• Inter-thread communication is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock)

in the same critical section to be executed.
• To avoid polling(It is usually implemented by loop), Inter-thread communication is implemented by following methods of Object class:
• wait( ): This method tells the calling thread to give up the critical section and go to sleep until some other thread enters the same critical section and calls notify( ).
• notify( ): This method wakes up the first thread that called wait( ) on the same object.
• notifyAll( ): This method wakes up all the threads that called wait( ) on the same object. The highest priority thread will run first.
• Above all methods are implemented as final in Object class.
• All three methods can be called only from within a synchronized context.

FILE HANDLING

Streams

• A stream is a sequence of data. In Java a stream is composed of bytes.
• In java, 3 streams are created for us automatically.
1. System.out:standard output stream
2. System.in:standard input stream
3. System.err:standard error stream

Byte Streams

• Java byte streams are used to perform input and output of 8-bits / 1byte at a time from binary file.
• InputStream and OutpufStream class are most common used classes of byte stream.

Method Call

Syntax:
var name object_name.method_name(parameter-list};
Example:
vol =bl.volumeQ;
• In above example, bl is an object and when volume{ } is called, it is put on the right side of an assignment statement. On the left is a variable, in this case vol, that will receive the vaiue returned by volume( ).

Byte Streams classes
BufferedInputStream -- Used for Buffered input stream.
BufferedOutputStream -- Used for Buffered output stream.
DataInputStream -- Contains method for reading java standard data type.
DataOutputStream --An output stream that contain method for writing java standard data type.
FileInputStream -- Input stream that reads from a file.
FileOutputStream -- Output stream that write to a file.
InputStream -- Abstract class that describe stream input.
OutputStream -- Abstract class that describe stream output.
PrintStream -- Output stream that contain print() and println() method.

FileOutputStream

• FileOutputStream is used to create a file and write data into it.
• The OutputStream will only create a file,if it doesn't already exist, before opening it for output.
• There are two way for using file with FileOutputStream.

1. FileOutputStream f = new FileOutputStream("C:/java/hello.txt");
Here, directly File path is specified while creating the FileOutputStream object.
2. File f = new File("C:/java/hello.txt");
FileOutputStream fl = new FileOutputStream(f);
In the second method,first object of File is created and then FileOutputStream object is created and File object passed to the FileOutputStream as a argument.

FileInputStrearn

• FileInputStream is used for reading data from the files.
• While using FileInputStream class, file must already exist if file does not exist it will generate error.
• There are two way for using file with FileInputStream.
1. FileInputStream fl = new FileInputStream{"C:/java/hello.txt");
Here,directly File path is specified while creatingthe FileInputStream object.
2. File f = new File(MC:/java/hello.txtM);
FileInputStream fl = new FileInputStream(f);
In the second method, first object of File is created and then FileInputStream object is created and File object passed to the FileInputStream as a argument.

Example of FileInputStream and FileOutputStream:

```
import java.io.*;
public class File_l
{
public static void main(String[] args)
{
try
{
FileInputStream in = new FileInputStream("C:/temp/Hello.txt ");
FileOutputStream out- new FileOutputStream{"C:/temp/Flellol.txt ");
int c = 0;
while ((c = in.read() ) !=-l)
{
out.write(c);
}
in.close();
out.close();
System.out.println("Successfully Write");
}
catch (FileNotFoundException e)
{
```

```
e,printStackTrace();
}
catch (IOException e)
{
e.printStackTrace();
}
}
}
```

Character Streams

• Character streams are used to perform input and output of 16-bits / 2 bytes at a time from Character file.
• Reader and Writer are most common used classes of character stream.

Character Stream classes

BufferedReader -- Handles buffered input stream.
BufferedWriter -- Handles buffered output stream.
FileReader -- Input stream that reads from file.
FileWriter -- Output stream that writes to file.
InputStreamReader -- Input stream that translate byte to character.
OutputStreamReader -- Output stream that translate character to byte.
Reader -- Abstract class that define character stream input.
Writer -- Abstract class that define character stream output.

FileWriter

• Java FileWriter class is used to write character data to the file.
• Two ways to create FileWriter Object.
1. FileWriter f = new FileWriter(String fname); //File path is specified in argument.
2. FileWriter f = new FileWriter(File fl); //1st file is created and it is passed as argument.

Method of FileWriter Class
void write(int ch) -- Writing single character to the file.
void write(String s) -- Writes the string into FileWriter.
void write(char[] c) -- Writes char array into FileWriter.
void close() -- Closes FileWriter.

Example of FileReader and FileWriter:

```
import java.io.*;
public class File_2
{
public static void main(String[] args) throws IOException
{
File f = new File("d:/temp/Hello.txt");
```

```
File fl= new File("d:/temp/Hellol.txt");
FileReader fr = new FileReader(f);
FileWriter fw = new FileWriter(fl);
int c = 0;
while ((c = fr.read())!=-l)
{
fw.write(c);
}
fr.close();
fw.close();
System.out.println("Successfully Write");
}
}
```

Buffered Classes

BufferedReader

• BufferedReader class can be used to read data line by line using of readLine() method.

BufferedOutputStream

• BufferedOutputStream class uses an internal buffer to store data. •It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.

BufferedInputStream

• BufferedInputStream class is used to read information from stream. It internally uses buffer mechanism to make the performance fast.

Example:

```
public class File_3
{ public static void main(String[] args) { InputStreamReader in = new
InputStreamReader(System.in);
BufferedReader br = new BufferedReader(in);
File f = new File("d:/temp/Flello.txt"); try
{
OutputStream out = new FileOutputStream(f);
System.out.print("Type your text to write in file==- ");
String str;
str = br.readUne();
byte b[] = str.getBytes();
out.write(b);
System.out.println("File Successfully Write,.");
br.close();
in.close();
```

```
outclose();
}
catch MOException (e)
{
e.printStackTrace();
}
}
}
```