MODULE 2: INTRODUCTION TO JAVA

Syllabus:

Java and Java applications; Java Development Kit (JDK); Java is interpreted, Byte Code, JVM; Object-oriented programming; Simple Java programs.

Data types and other tokens: Boolean variables, int, long, char, operators, arrays, white spaces, literals, assigning values; Creating and destroying objects; Access specifiers.

Operators and Expressions: Arithmetic Operators, Bitwise operators, Relational operators, The Assignment Operator, The ? Operator; Operator Precedence; Logical expression; Type casting; Strings.

Control Statements: Selection statements, iteration statements, Jump
Statements.

MODULE 2: Introduction to JAVA

Basic concepts of object oriented programming

Object:

This is the basic unit of object oriented programming. That is both data and method that operate on data are bundled as a unit called as object. **It is a real world entity (Ex:a person, book, tables, chairs etc…)**

Class:

Class is a **collection of objects** or class is a **collection of instance variables and methods**. When you define a class, you define a blueprint for an object. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

Abstraction:

Data abstraction refers to, **providing only essential information** to the outside word and hiding their background details ie. to represent the needed information in program without presenting the details.

For example, a database system hides certain details of how data is stored and created and maintained. Similar way, C++ classes provides different methods to the outside world without giving internal detail about those methods and data.

Encapsulation:

Encapsulation is ***placing the data and the methods/functions*** that work on that data in the same place. While working with procedural languages, it is not always clear which functions work on which variables but object-oriented programming provides you framework to place the data and the relevant functions together in the same object. ***Inheritance:***

One of the most useful aspects of object-oriented programming is **code reusability**. As the name suggests Inheritance is the process of forming a new class from an existing class that is from the existing class called as base class, new class is formed called as derived class.

This is a very important concept of object oriented programming since this feature helps to reduce the code size.

Polymorphism:

The ability to use a method/function in different ways in other words giving different meaning for method/ functions is called polymorphism. Poly refers many. That is a single method/function functioning in many ways different upon the usage is called polymorphism. **Java History:**

Java is a general-purpose object oriented programming language developed by sun Microsystems of USA in the year 1991. The original name of Java is Oak. Java was designed for the development of the software for consumer electronic devices like TVs, VCRs, etc.

Introduction: Java is a general purpose programming language. We can develop two types of Java application. They are:

(1). Stand alone Java application. (2). Web applets.

Stand alone Java application: Stand alone Java application are programs written in Java to carry out certain tasks on a certain stand alone system. Executing a stand-alone Java program contains two phases:

(a) Compiling source coded into bytecode using javac compiler. (b) Executing the bytecodede program using Java interpreter.

Java applet: Applets are small Java program developed for Internet application. An applet located on a distant computer can be downloaded via Internet and execute on local computer.

Java and Internet:

Java is strongly associated with Internet. Internet users can use Java to create applet programs and run them locally using a "Java enabled Browser" such as "hotjave". They can also use a Java enabled browser to download an applet locating on any computer any where in the internet and run them locally.

Internet users can also set their web-sites containing Java applets that could be used by other remote users of Internet. The ability of the Java applets to hitch a ride on the information makes Java a unique programming language for Internet.

Java Environment:

Java environment includes a large number of development tools and hundreds of classes and methods. The Java development tools are part of the systems known as Java development kit (JDK) and the classes and methods are part of the Java standard library known as Java standard Library (JSL) also known as application program interface (API).

Java Features/Buzzwords:

(1) Compiled and Interpreted
(2) Architecture Neutral/Platform independent and portable
(3) Object oriented
(4) Robust and secure. (5)
Distributed.
(6) Familiar, simple and small.
(7) Multithreaded and interactive. (8)
High performance
(9) Dynamic and extendible.

1. Compiled and Interpreted

Usually a computer language is either compiled or interpreted. Java combines both these approaches; first java compiler translates source code into bytecode instructions. Bytecodes are not machine instructions and therefore, in the second stage, java interpreter generates machine code that can be directly executed by the machine that is running the java program.

2. Architecture Neutral/Platform independent and portable

The concept of Write-once-run-anywhere (known as the Platform independent) is one of the important key feature of java language that makes java as the most powerful language. Not even a single language is idle to this feature but java is closer to this feature. The programs written on one platform can run on any platform provided the platform must have the JVM.

3. Object oriented

In java everything is an Object. Java can be easily extended since it is based on the Object model.java is a pure object oriented language.

4. Robust and secure.

Java is a robust language; Java makes an effort to eliminate error situations by emphasizing mainly on compile time error checking and runtime checking. Because of absence of pointers in java we can easily achieve the security.

5. Distributed.

Java is designed for the distributed environment of the internet.java applications can open and access remote objects on internet as easily as they can do in the local system.

6. Familiar, simple and small.

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

7. Multithreaded and interactive.

With Java's multi-threaded feature it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct smoothly running interactive applications.

8. High performance

Because of the intermediate bytecode java language provides high performance

9. Dynamic and extendible.

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.

Java Development kits(java software:jdk1.6): Java development kit comes with a number of Java development tools. They are:

(1) Appletviewer: Enables to run Java applet. (2) javac: Java compiler.
(3) java : Java interpreter.
(4) javah : Produces header files for use with native methods. (5) javap : Java disassembler.
(6) javadoc : Creates HTML documents for Java source code file.
(7) jdb : Java debugger which helps us to find the error.

Java Building and running Process:

1. Open the notepad and type the below program

Simple Java program:

Example:

```
class Sampleone
{
        public static void main(String args[])
        {
                System.out.println("Welcome to JAVA");
        }
}
```

Description:

(1) **Class declaration:** "class sampleone" declares a class, which is an object- oriented construct. Sampleone is a Java identifier that specifies the name of the class to be defined.

(2) **Opening braces:** Every class definition of Java starts with opening braces and ends with matching one.

(3) **The main line:** the line " public static void main(String args[]) " defines a method name main. Java application program must include this main. This is the starting point of the interpreter from where it starts executing. A Java program can have any number of classes but only one class will have the main method.

(4) **Public:** This key word is an access specifier that declares the main method as unprotected and therefore making it accessible to the all other classes.

(5) **Static:** Static keyword defines the method as one that belongs to the entire class and not for a particular object of the class. The main must always be declared as static.

(6) **Void:** the type modifier void specifies that the method main does not return any value.

(7) **The println:** It is a method of the object out of system class. It is similar to the printf or cout of c or c++.

2. Save the above program with .java extension, here file name and class name should be same, ex: Sampleone.java,
3. Open the command prompt and **Compile** the above program

   javac Sampleone.java

   From the above compilation the java compiler produces a bytecode(.class file)
4. Finally run the program through the **interpreter**

   java Sapleone.java

Output of the program:

Welcome to JAVA

Implementing a Java program: Java program implementation contains three stages. They are:

1. Create the source code.
2. Compile the source code.
3. Execute the program.

(1) Create the source code:

1. Any editor can be used to create the Java source code.
2. After coding the Java program must be saved in a file having the same name of the class containing main() method.
3. Java code file must have .Java extension.

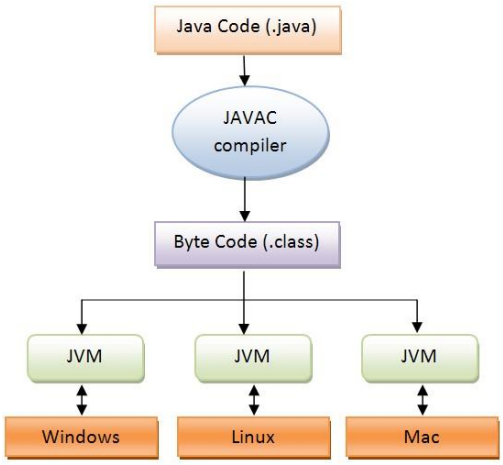
(2) Compile the source code:

1. Compilation of source code will generate the bytecode.
2. JDK must be installed before completion.
3. Java program can be compiled by typing javac <filename>.java
4. It will create a file called <filename>.class containing the bytecode.

(3) Executing the program:

1. Java program once compiled can be run at any system.
2. Java program can be execute by typing Java <filename>

JVM(Java Virtual Machine)



The concept of Write-once-run-anywhere (known as the Platform independent) is one of the important key feature of java language that makes java as the most powerful language. Not even a single language is idle to this feature but java is closer to this feature. The programs written on one platform can run on any platform provided the platform must have the JVM(**Java Virtual Machine**). A Java virtual machine (JVM) is a virtual machine that can execute Java bytecode. It is the code execution component of the Java software platform.

More Examples:

Java program with multiple lines:

Example:

```
import java.lang.math;
class squreroot
{
        public static void main(String args[])
        {
                double x = 5;
                double y;
                y = Math.sqrt(x);
                System.out.println("Y = " + y);
        }
}
```

Java Program structure: Java program structure contains six stages. They are:

(1) Documentation section: The documentation section contains a set of comment lines describing about the program.

(2) Package statement: The first statement allowed in a Java file is a package statement. This statement declares a package name and informs the compiler that the class defined here belong to the package.

```
Package student;
```

(3) Import statements: Import statements instruct the compiler to load the specific class belongs to the mentioned package.

```
Import student.test;
```

(4) Interface statements: An interface is like a class but includes a group of method deceleration. This is an optional statement.

(5) Class definition: A Java program may contain multiple class definition The class are used to map the real world object.

(6) Main method class: The main method creates objects of various classes and establish communication between them. On reaching to the end of main the program terminates and the control goes back to operating system.

Java command line arguments: Command line arguments are the parameters that are supplied to the application program at the time when they are invoked. The main() method of Java program will take the command line arguments as the parameter of the args[ ] variable which is a string array.

Example:

```
Class Comlinetest
{
        public static void main(String args[ ] )
        {
                int count, n = 0;
                string str;
                count = args.length;
                System.out.println ( " Number of arguments :" + count);
                While ( n < count )
                {
                        str = args[ n ];
                       n = n + 1;
                       System.out.println( n + " : " + str);
                }
        }
}
```

Run/Calling the program:

javac Comlinetest.java
java Comlinetest Java c cpp fortran

Output:

1 : Java
2 : c
3 : cpp
4 : fortran

Java API:

Java standard library includes hundreds of classes and methods grouped into several functional packages. Most commonly used packages are:

(a) Language support Package. (b)
Utilities packages.
(c) Input/output packages (d)
Networking packages (e)
AWT packages.
(f) Applet packages.

Java Tokens

Constants: Constants in Java refers to fixed value that do not change during the execution of program. Java supported constants are given below:

(1) **Integer constants:** An integer constant refers to a sequence of digits. There are three types of integer namely decimal integer, octal integer and hexadecimal integer. For example: 123 -321

(2) **Real constants:** Any real world number with a decimal point is known as real constants. For example : 0.0064 12e-2 etc.

(3) **Single character constants:** A single character constant contains a single character enclosed with in a pair of single quotes. For ex: 'm' '5'

(4) **String constants :** A string constant is a sequence of character enclosed with double quotes. For ex: "hello" "java" etc.

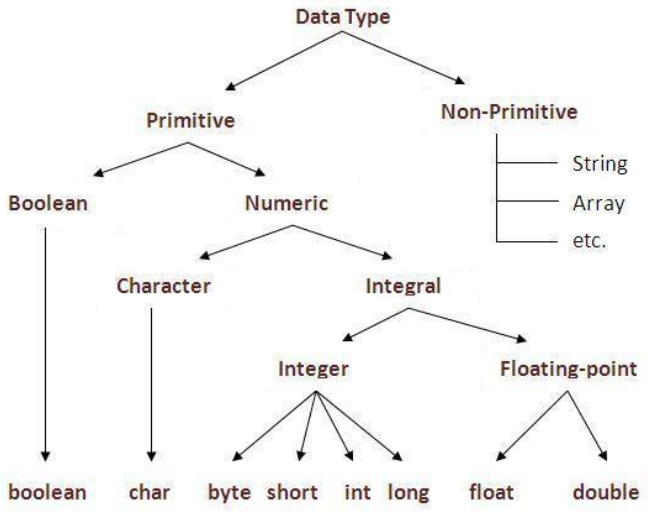
(5) **Backslash character constants:** Java supports some backslash constants those are used in output methods. They are :

1. \b Backspace
2. \f Form feed
3. \n New Line
4. \r Carriage return.
5. \t Horizontal tab.
6. \' Single quotes.
7. \" Double quotes
8. \\ Back slash

Data Types in Java:

In java, data types are classified into two catagories :

1. Primitive Data type
2. Non-Primitive Data type



Data Type	Default Value	Default size
boolean	False	1
bit char	'\u0000'	2
byte byte	0	
1 byte short	0	
2 byte int	0	
4 byte long	0L	
8 byte float	0.0f	
4 byte double	0.0d	

8 byte

Integers Type: Java provides four types of Integers. They are byte, sort, Int, long. All these are sign, positive or negative.

Byte: The smallest integer type is byte. This is a signed 8-bit type that has a range from –128 to 127. Bytes are useful for working with stream or data from a network or file. They are also useful for working with raw binary data. A byte variable is declared with the keyword "byte".

```
byte b, c;
```

Short: Short is a signed 16-bit type. It has a range from –32767 to 32767. This data type is most rarely used specially used in 16 bit computers. Short variables are declared using the keyword short.

```
short a, b;
```

int: The most commonly used Integer type is int. It is signed 32 bit type has a range from –2147483648 to 2147483648.

```
int a, b, c;
```

long: Long is a 64 bit type and useful in all those occasions where Int is not enough. The range of long is very large. long a, b;

Floating point types: Floating point numbers are also known as real numbers are useful when evaluating a expression that requires fractional precision. The two floating-point data types are float and double.

float: The float type specifies a single precision value that uses 32-bit storage. Float keyword is used to declare a floating point variable. float a, b;

double: Double DataTips is declared with **double** keyword and uses 64-bit value.

Characters: The Java data type to store characters is char. **char data type** of Java uses Unicode to represent characters. Unicode defines a fully international character set that can have all the characters of human language. Java char is 16-bit type. The range is 0 to 65536.

Boolean: Java has a simple type called **boolean** for logical values. It can have only one of two possible values. They are true or false.

Key Words: Java program is basically a collection of classes. A class is defined by a set of declaration statements and methods containing executable statements. Most statement contains an expression that contains the action carried out on data. The compiler recognizes the tokens for building up the expression and statements. Smallest individual units of programs are known as tokens. Java language includes five types of tokens. They are

(a) Reserved Keyword
(b) Identifiers
(c) Literals. (d) Operators (e) Separators.

(1) **Reserved keyword:** Java language has 60 words as reserved keywords. They implement specific feature of the language. The keywords combined with operators and separators according to syntax build the Java language.

(2) **Identifiers:** Identifiers are programmer-designed token used for naming classes methods variable, objects, labels etc. The rules for identifiers are

1. They can have alphabets, digits, dollar sign and underscores.
2. They must not begin with digit.
3. Uppercase and lower case letters are distinct.
4. They can be any lengths.
5. Name of all public method starts with lowercase.

6. In case of more than one word starts with uppercase in next word.
7. All private and local variables use only lowercase and underscore.
8. All classes and interfaces start with leading uppercases.
9. Constant identifier uses uppercase letters only.

(3) **Literals:** Literals in Java are sequence of characters that represents constant values to be stored in variables. Java language specifies five major types of Literals. They are:

1. Integer Literals.
2. Floating-point Literals.
3. Character Literals.
4. String Literals.
5. Boolean Literals.

(4) **Operators:** An operator is a symbol that takes one or more arguments and operates on them to produce an result.

(5) **Separators:** Separators are the symbols that indicates where group of code are divided and arranged. Some of the operators are:

1. Parenthases()
2. Braces{ }
3. Brackets [ ]
4. Semicolon ;
5. Comma ,
6. Period .

Java character set: The smallest unit of Java language are its character set used to write Java tokens. This character are defined by unicode character set that tries to create character for a large number of character worldwide.

The Unicode is a 16-bit character coding system and currently supports 34,000 defined characters derived from 24 languages of worldwide.

Variables: A variable is an identifier that denotes a storage location used to store a data value. A variable may have different value in the different phase of the program. To declare one identifier as a variable there are certain rules. They are:

1. They must not begin with a digit.
2. Uppercase and lowercase are distinct.

3. It should not be a keyword.
4. White space is not allowed.

Declaring Variable: One variable should be declared before using. The syntax is

Data-type variblaname1, variablename2,. . . . . variablenameN;

Initializing a variable: A variable can be initialize in two ways. They are

(a) Initializing by Assignment statements. (b) Initializing by Read statements.

Initializing by assignment statements: One variable can be initialize using

assignment statements. The syntax is :

Variable-name = Value;

Initialization of this type can be done while declaration.

Initializing by read statements: Using read statements we can get the values in the variable.

Scope of Variable: Java variable is classified into three types. They are

(a) Instance Variable
(b) Local Variable
(c) Class Variable

Instance Variable: Instance variable is created when objects are instantiated and therefore they are associated with the object. They take different values for each object.

Class Variable: Class variable is global to the class and belongs to the entire set of object that class creates. Only one memory location is created for each class variable.

Local Variable: Variable declared inside the method are known as local variables. Local variables are also can be declared with in program blocks. Program blocks can

be nested. But the inner blocks cannot have same variable that the outer blocks are having.

Arrays in Java

Array which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Declaring Array Variables:

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable:

dataType[] arrayRefVar; or dataType arrayRefVar[];

Example:

The following code snippets are examples of this syntax:

int[] myList; or int myList[];

Creating Arrays:

You can create an array by using the new operator with the following syntax:

arrayRefVar = new dataType[arraySize];

The above statement does two things:

It creates an array using new **dataType[arraySize];**
It assigns the reference of the newly created array to the variable **arrayRefVar**.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below:

dataType[] arrayRefVar = new dataType[arraySize];

Alternatively you can create arrays as follows:

dataType[] arrayRefVar = {value0, value1, ..., valuek};

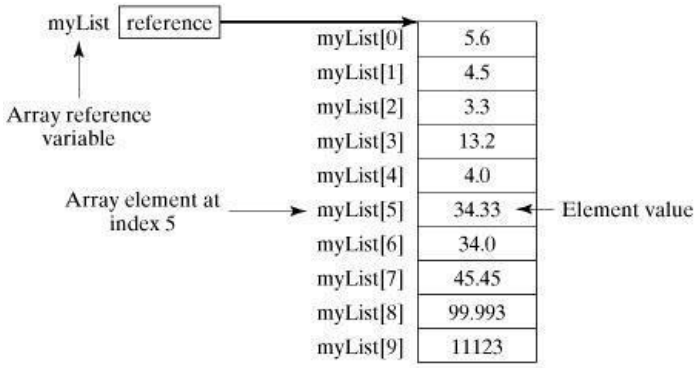
The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.

Example:

Following statement declares an array variable, myList, creates an array of 10 elements of double type and assigns its reference to myList:

double[] myList = new double[10];

Following picture represents array myList. Here, myList holds ten double values and the indices are from 0 to 9.



Processing Arrays:

When processing array elements, we often use either for loop or foreach loop because all of the elements in an array are of the same type and the size of the array is known.

Example:

Here is a complete example of showing how to create, initialize and process arrays:

```
public class TestArray
 {
        public static void main(String[] args)
        {
                double[] myList = {1.9, 2.9, 3.4, 3.5};

                // Print all the array elements
                for (int i = 0; i < myList.length; i++) {
                System.out.println(myList[i] + " ");
        }
```

```
      // adding all elements
            double total = 0;
            for (int i = 0; i < myList.length; i++)
            {
                  total += myList[i];
            }
     System.out.println("Total is " + total);
     // Finding the largest element
     double max = myList[0];
     for (int i = 1; i < myList.length; i++)
    {
       if (myList[i] > max)
       max = myList[i];
     }
     System.out.println("Max is " + max);
  }
}
```

This would produce the following result:

```
1.9
2.9
3.4
3.5
Total is 11.7
Max is 3.5
```

The foreach Loop

JDK 1.5 introduced a new for loop known as for-each loop or enhanced for loop, which enables you to traverse the complete array sequentially without using an index variable.

Example:

The following code displays all the elements in the array myList:

```
public class TestArray {

      public static void main(String[] args) {
     double[] myList = {1.9, 2.9, 3.4, 3.5};

     // Print all the array elements
```

```
      for (double  element: myList)
      {
         System.out.println(element);
      }
   }
}
```

This would produce the following result:

```
1.9
2.9
3.4
3.5
```

Type Casting: It is often necessary to store a value of one type into the variable of another type.

In these situations the value that to be stored should be casted to destination type. Type casting can be done in two ways.

Type Casting

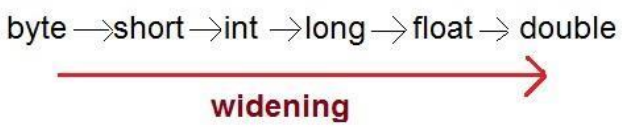
Assigning a value of one type to a variable of another type is known as **Type Casting**.

Example :

```
int x = 10;
byte y = (byte)x;
```

In Java, type casting is classified into two types,

Widening Casting(Implicit)



Narrowing Casting(Explicitly done)

Widening or Automatic type converion

Automatic Type casting take place when,

the two types are compatible
the target type is larger than the source type

Example :

```
public class Test
{
        public static void main(String[] args)
        {
            int i = 100;
            long l = i;      //no explicit type casting required
            float f = l;     //no explicit type casting required System.out.println("Int value
            "+i); System.out.println("Long value "+l); System.out.println("Float value
            "+f);
        }

}
```

Output :

```
Int value 100
Long value 100
Float value 100.0
```

Narrowing or Explicit type conversion

When you are assigning a larger type value to a variable of smaller type, then you need to perform explicit type casting.

Example :

```
public class Test
{
        public static void main(String[] args)
        {
            double d = 100.04;
```

```
        long l = (long)d;  //explicit type casting required
        int i = (int)l;  //explicit type casting required

        System.out.println("Double value "+d);
        System.out.println("Long value "+l);
        System.out.println("Int value "+i);

    }

}
```

Output :

```
Double value 100.04
Long value 100
Int value 100
```

Java operators:

Java operators can be categorized into following ways:

(1) Arithmetic operator
(2) Relational operator
(3) Logical operator
(4) Assignment operator
(5) Increment and decrement operator
(6) Conditional operator
(7) Bitwise operator
(8) Special operator.

Arithmetic operator: The Java arithmetic operators are:

+	: Addition
-	: Subtraction
	: Multiplication
/	: Division
%	: Remainder

Relational Operator:

<	: Is less then
<=	: Is less then or equals to
	: Is greater then
>=	: Is grater then or equals to
==	: Is equals to
!=	: Is not equal to

Logical Operators:

&&	: Logical AND
||	: Logical OR

! : Logical NOT

Assignment Operator:

+= : Plus and assign to
-= : Minus and assign to
*= : Multiply and assign to.
/= : Divide and assign to.
%= : Mod and assign to.
= : Simple assign to.

Increment and decrement operator:

++ : Increment by One {Pre/Post)
-- : Decrement by one (pre/post)

Conditional Operator: Conditional operator is also known as ternary operator.

The conditional operator is :

Exp1 ? exp2 : exp3

Bitwise Operator: Bit wise operator manipulates the data at Bit level. These operators are used for tasting the bits. The bit wise operators are:

& : Bitwise AND
! : Bitwise OR
^ : Bitwise exclusive OR
~ : One's Complement.
<< : Shift left.
>> : Shift Right.
>>> : Shift right with zero fill

Example:

Bitwise operator works on bits and performs bit-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

<<	**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

Special Operator:

Instanceof operator: The instanceof operator is a object refrence operator that returns true if the object on the right hand side is an instance of the class given in the left hand side. This operator allows us to determine whether the object belongs to the particular class or not.

Person instanceof student

The expression is true if the person is a instance of class student.

Dot operator: The dot(.) operator is used to access the instance variable or method of class object.

Example Programs

```
class arithmeticop
{
	public static void main(String args[])
	{
		float a=20.5f; float b=6.4f;
		System.out.println("a = " + a);
		System.out.println("b = " + b );
		System.out.println("a + b = " + (a+b));
	}
}
```

```
class Bitlogic
{
	public static void main(String args[])
	{
String binary[] = {"0000","0001","0010","0011","0100","0101","0110","0111","1000","1001","1010",
                   "1011","1100","1101","1110","1111"};
		int a = 3;
		int b = 6;
		int c = a | b;
		int d = a & b;
		int e = a ^ b;
		int f = (~a&b)|(a & ~b); System.out.println("a
		or b :"+binary[c]);
		System.out.println("a and b : "+binary[d]);
		System.out.println("a xor b :  "+binary[e]);
		System.out.println("(~a&b)|(a & ~b) :  "+binary[f]);
	}
}
```

Control Statements

Decision making statements:

1.Simple If statement:

The general form of single if statement is :

```
If ( test expression)
{
        statement-Block;
}
statement-Blocks;
```

2. If- Else statement:

The general form of if-else statement is

```
If ( test expression)
{
        statement-block1;
}
else
{
        statement-block2
}
```

3. Else-if statement:

The general form of else-if statement is:

```
If ( test condition)
{
        statement-block1;
}
else if(test expression2)
{
        statement-block2;
}
 else
{
        statement block3;
}
```

4. Nested if – else statement:

The general form of nested if-else statement is:

```
If ( test condition)
{
        if ( test condition)
        {
                statement block1;
        }
         else
        {
```

```
                    statement block2;
                }
        }
         else
        {
                statement block 3
        }
```

5. The switch statements:

The general form of switch statement is: Switch

```
( expression)
{
        case value-1:
                block-1;
                break;
         case value-2:
                block-2;
                break;
        . . . . . .
        default:
                default block;
                break;
}
```

Loops In Java: In looping a sequence of statements are executed until a number of time or until some condition for the loop is being satisfied. Any looping process includes following four steps:

(1) Setting an initialization for the counter.
(2) Execution of the statement in the loop
(3) Test the specified condition for the loop. (4)
Incrementing the counter.

Java includes three loops. They are:

(1) While loop:

The general structure of a while loop is:

```
Initialization
While (test condition)
{
        body of the loop
}
```

(2) Do loop:

The general structure of a do loop is :

```
Initialization
do
{
        Body of the loop;
}
while ( test condition);
```

(3) For loop :

The general structure of for loop is:

```
For ( Initialization ; Test condition ; Increment)
{
        body of the loop;
}
```

More about Loops:

Break Statement: Using "break" statement we can jump out from a loop. The "break" statement will cause termination of the loop.

Continue statement: The "continue" statement will cause skipping some part of the loop.

Labeled loops: We can put a label for the loop. The label can be any Java recognized keyword. The process of giving label is

```
Label-name : while (condition)
                {
                        Body ;
                }
```

Example Program for label break statement

```
class BreakTest
{
    public static void main(String args[])
    {
        boolean t= true;
        first:
        {
            second:
            {
                third:
                {
                    System.out.println("Third stage");
                    if(t)
                        break second;
                        System.out.println("Third stage complete");
                }
                System.out.println("Second stage");
            }
            System.out.println("First stage");
        }
    }
```

```
}
```

Example Program for continue statement

```
class ContinueTest
{
        public static void main(String args[])
        {
                outer:  for(int i=0;i<10;i++)
                        {
                                for(int j = 0; j<10;j++)
                                {
                                        if(j>i)
                                        {
                                                System.out.println("\n");
                                                continue outer;
                                        }
                                        System.out.print("  "+(i*j));
                                 }
                         }

        //System.out.println(" ");
        }
}
```

Example Program for return statement

```
class ReturnTest
{
        public static void main(String args[])
        {
                boolean t = true; System.out.println("Before the
                return"); if(t) return;
                System.out.println("After return");
        }
}
```

Java Access Specifiers/Modifiers

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.

Using the private modifier is the main way that an object encapsulates itself and hides data from the outside world.

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.

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.

Default Access Modifier - No keyword:

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.

Advantages of JAVA:

• It is an open source, so users do not have to struggle with heavy license fees each year.

• Platform independent.

• Java API's can easily be accessed by developers.

• Java perform supports garbage collection, so memory management is automatic.

• Java always allocates objects on the stack.

• Java embraced the concept of exception specification.

• Multi-platform support language and support for web-services.

• Using JAVA we can develop dynamic web applications.

• It allows you to create modular programs and reusable codes.

UNIT-3: CLASSES, INHERITANCE, EXCEPTIONS, PACKAGES AND INTERFACES

Syllabus:

Classes, Inheritance, Exceptions, Packages and Interfaces:

Classes: Classes fundamentals; Declaring objects; Constructors, this keyword, garbage collection.

Inheritance: inheritance basics, using super, creating multi level hierarchy, method overriding.

Exception handling: Exception handling in Java. Packages, Access Protection, Importing Packages, Interfaces. .

1. CLASSES:

Definition

A class is a template for an object, and defines the data fields and methods of the object. The class methods provide access to manipulate the data fields. The "data fields" of an object are often called "instance variables."

Example Program:

```
Program to calculate Area of Rectangle
class Rectangle
{
        int length;             //Data Member or instance Variables
        int width;
        void getdata(int x,int y)       //Method
        {
                length=x;
                width=y;
        }
        int rectArea()                  //Method
        {
                return(length*width);
        }
}

class RectangleArea
{
    public static void main(String args[])
    {
          Rectangle rect1 =new Rectangle(); //object creation
              rect1.getdata(10,20); //calling methods using object with dot(.)
         int area1=rect1.rectArea();
        System.out.println("Area1="+area1);
    }
}
```

- ✓ After defining a class, it can be used to create objects by instantiating the class. Each object occupies some memory to hold its instance variables (i.e. its state).
- ✓ After an object is created, it can be used to get the desired functionality together with its class.

Creating instance of a class/Declaring objects:

```
Rectangle rect1 =new Rectangle()
Rectangle rect2=new Rectangle()
```

- ✓ The above two statements declares an **object rect1 and rect2 is of type Rectangle class using new operator** , this operator dynamically allocates memory for an object and returns a refernce to it.in java all class objects must be dynamically allocated.

We can also declare the object like this:

```
Rectangle rect1;          // declare reference to object.
rect1 =new Rectangle()    // allocate memory in the Rectangle object.
```

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:

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

Parameterized Constructor:

- ✓ 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 Parameterized constructor
   MyClass(int i )
   {
      x = 10;
   }
}
```

You would call constructor to initialize objects as follows:

```
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 following result:

```
10 20
```

static keyword

The **static keyword** is used in java mainly for memory management. We may apply static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class.

The static can be:

1. variable (also known as class variable)
2. method (also known as class method)
3. block
4. nested class

static variable

Example Program without static variable

In this example, we have created an instance variable named count which is incremented in the constructor. Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable, if it is incremented, it won't reflect to other objects. So each objects will have the value 1 in the count variable.

```
class Counter
{
        int count=0;//will get memory when instance is created
        Counter()
        {
                count++;
                System.out.println(count);
        }
}

Class MyPgm
{
        public static void main(String args[])
        {
                        Counter c1=new Counter();
```

```
                Counter c2=new Counter();
                Counter c3=new Counter();
            }
        }
Output: 1
          1
          1
```

Example Program with static variable

As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

```
class Counter
{
        static int count=0;//will get memory only once and retain its value

        Counter()
        {
                count++;
                System.out.println(count);
        }
 }
Class MyPgm
{
        public static void main(String args[])
        {
                Counter c1=new Counter();
                Counter c2=new Counter();
                Counter c3=new Counter();
        }
}
Output:1
        2
        3
```

static method

If you apply static keyword with any method, it is known as static method

- ✓ A static method belongs to the class rather than object of a class.
- ✓ A static method can be invoked without the need for creating an instance of a class.
- ✓ static method can access static data member and can change the value of it.

//Program to get cube of a given number by static method

```
 class Calculate
{
        static int cube(int x)
        {
                return x*x*x;
        }
Class MyPgm
{
        public static void main(String args[])
        {
                //calling a method directly with class (without creation of object)
                int result=Calculate.cube(5);
                System.out.println(result);
        }
}
```

Output:125

this keyword

- ✓ this keyword can be used to refer current class instance variable.
- ✓ If there is ambiguity between the instance variable and parameter, this keyword resolves the problem of ambiguity.

Understanding the problem without this keyword

Let's understand the problem if we don't use this keyword by the example given below:

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

      student(int id,String name)
      {
            id = id;
            name = name;
      }
      void display()
      {
            System.out.println(id+" "+name);
      }
}
Class MyPgm
{
      public static void main(String args[])
      {
            student s1 = new student(111,"Anoop");
            student s2 = new student(321,"Arayan");
            s1.display();
            s2.display();
      }
}
```

Output: 0 null
0 null

In the above example, parameter (formal arguments) and instance variables are same that is why we are using this keyword to distinguish between local variable and instance variable.

Solution of the above problem by this keyword

```
//example of this keyword
class Student
{
        int id;
        String  name;

        student(int id,String name)
        {
                this.id = id;
                this.name = name;
        }
         void display()
        {
                System.out.println(id+" "+name);
        }
}
   Class MyPgm
   {
        public static void main(String args[])
        {
                Student s1 = new Student(111,"Anoop");
                Student s2 = new Student(222,"Aryan");
                s1.display();
                s2.display();
        }
   }
```

Output111 Anoop
222 Aryan

Inner class

- ✓ It has access to all variables and methods of **Outer** class and may refer to them directly. But the reverse is not true, that is, **Outer** class cannot directly access members of **Inner** class.
- ✓ One more important thing to notice about an **Inner** class is that it can be created only within the scope of **Outer** class. Java compiler generates an error if any code outside **Outer** class attempts to instantiate **Inner** class.

Example of Inner class

```
class Outer
{
        public void display()
        {
                Inner in=new Inner();
                in.show();
        }

        class Inner
        {
                public void show()
                {
                        System.out.println("Inside inner");
                }
        }
}

class Test
{
        public static void main(String[] args)
        {
                Outer ot=new Outer();
                ot.display();
        }
}
```

Output:

Inside inner

Garbage Collection

In Java destruction of object from memory is done automatically by the JVM. When there is no reference to an object, then that object is assumed to be no longer needed and the memories occupied by the object are released. This technique is called **Garbage Collection**. This is accomplished by the JVM.

Can the Garbage Collection be forced explicitly?

No, the Garbage Collection cannot be forced explicitly. We may request JVM for **garbage collection** by calling **System.gc()** method. But this does not guarantee that JVM will perform the garbage collection.

Advantages of Garbage Collection

1. Programmer doesn't need to worry about dereferencing an object.
2. It is done automatically by JVM.
3. Increases memory efficiency and decreases the chances for memory leak.

---

finalize() method

Sometime an object will need to perform some specific task before it is destroyed such as closing an open connection or releasing any resources held. To handle such situation **finalize()** method is used. **finalize()** method is called by garbage collection thread before collecting object. It's the last chance for any object to perform cleanup utility.

Signature of **finalize()** method

```
protected void finalize()
{
 //finalize-code
}
```

gc() Method

gc() method is used to call garbage collector explicitly. However **gc()** method does not guarantee that JVM will perform the garbage collection. It only requests the JVM for garbage collection. This method is present in **System** and **Runtime** class.

Example for gc() method

```
public class Test
{

   public static void main(String[] args)
   {
      Test t = new Test();
      t=null;
      System.gc();
   }
   public void finalize()
   {
      System.out.println("Garbage Collected");
   }
}
```

Output :

Garbage Collected

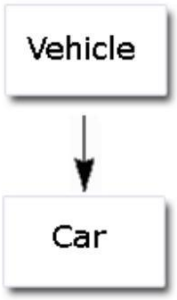
Inheritance:

- ✓ As the name suggests, inheritance means to take something that is already made. It is one of the most important features of Object Oriented Programming. It is the concept that is used for **reusability** purpose.
- ✓ Inheritance is the mechanism through which we can derive classes from other classes.
- ✓ The derived class is called as child class or the subclass or we can say the extended class and the class from which we are deriving the subclass is called the base class or the parent class.
- ✓ To derive a class in java the keyword **extends** is used. The following kinds of inheritance are there in java.

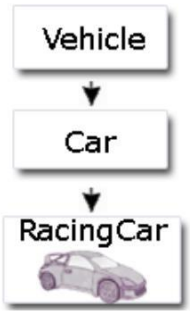
Types of Inheritance

1. **Single level/Simple Inheritance**
2. **Multilevel Inheritance**
3. **Multiple Inheritance (Java doesn't support Multiple inheritance but we can achieve this through the concept of Interface.)**

Pictorial Representation of Simple and Multilevel Inheritance



Simple Inheritance



Multilevel Inheritance

Single level/Simple Inheritance

- ✓ When a subclass is derived simply from its parent class then this mechanism is known as simple inheritance. In case of simple inheritance there is only a sub class and its parent class. It is also called single inheritance or one level inheritance.

Example

```
class A
{
      int x;
      int y;
      int get(int p, int q)
     {
           x=p;
           y=q;
           return(0);
      }
   void Show()
    {
         System.out.println(x);
    }
}
class B extends A
{
         public static void main(String args[])
         {
               A a = new A();
                  a.get(5,6);
                  a.Show();
          }
         void display()
         {
              System.out.println("y");        //inherited "y" from class A
         }
}
```

- ✓ The syntax for creating a subclass is simple. At the beginning of your class declaration, use the extends keyword, followed by the name of the class to inherit from:

```
class A
{

}

class B extends A                    //B is a subclass of super class A.
{

}
```

Multilevel Inheritance

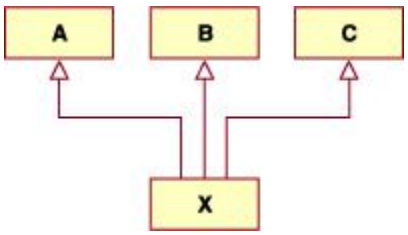
- ✓ When a subclass is derived from a derived class then this mechanism is known as the multilevel inheritance.
- ✓ The derived class is called the subclass or child class for it's parent class and this parent class works as the child class for it's just above (parent) class.
- ✓ Multilevel inheritance can go up to any number of level.

```
class A
{
      int x;
      int y;
      int get(int p, int q)
      {
          x=p;
          y=q;
          return(0);
      }
      void Show()
      {
          System.out.println(x);
      }
}
```

```
class B extends A
{
      void Showb()
      {
            System.out.println("B");
      }
}

class C extends B
{
      void display()
      {
            System.out.println("C");
      }
      public static void main(String args[])
      {
            A a = new A();
              a.get(5,6);
              a.Show();
      }
}
OUTPUT
5
```

Multiple Inheritance

- ✓ The mechanism of inheriting the features of more than one base class into a single class is known as multiple inheritance. Java does not support multiple inheritance but the multiple inheritance can be achieved by using the interface.



- ✓ Here you can derive a class from any number of base classes. Deriving a class from more than one direct base class is called multiple inheritance.

Java does not support multiple Inheritance

In Java Multiple Inheritance can be achieved through use of Interfaces by implementing more than one interfaces in a class.

super keyword

- ✓ The super is java keyword. As the name suggest super is used to access the members of the super class. It is used for two purposes in java.
- ✓ The first use of keyword super is to access the hidden data variables of the super class hidden by the sub class.

Example: Suppose class A is the super class that has two instance variables as int a and float b. class B is the subclass that also contains its own data members named a and b. then we can access the super class (class A) variables a and b inside the subclass class B just by calling the following command.

super.member;

- ✓ Here member can either be an instance variable or a method. This form of super most useful to handle situations where the local members of a subclass hides the members of a super class having the same name. The following example clarifies all the confusions.

Example:

```
class A
{
        int a;
        float b;
        void Show()
        {
              System.out.println("b in super class: " + b);
        }

}
class B extends A
{
        int a;
        float b;
        B( int p, float q)
        {
             a = p;
             super.b = q;
        }
       void Show()
       {
             super.Show();
             System.out.println("b in super class: " + super.b);
             System.out.println("a in sub class:    " + a);
      }
}

 class Mypgm
{
         public static void main(String[] args)
         {
                     B subobj = new B(1, 5);
                        subobj.Show();
         }
}
OUTPUT
b in super class: 5.0
b in super class: 5.0
a in sub class:   1
```

Use of super to call super class constructor: The second use of the keyword super in java is to call super class constructor in the subclass. This functionality can be achieved just by using the following command.

super(param-list);

- ✓ Here parameter list is the list of the parameter requires by the constructor in the super class. super must be the first statement executed inside a super class constructor. If we want to call the default constructor then we pass the empty parameter list. The following program illustrates the use of the super keyword to call a super class constructor.

Example:

```
class A
{
        int a;
        int b;
        int c;
        A(int p, int q, int r)
    {
           a=p;
           b=q;
           c=r;
     }
}

class B extends A
{
           int d;
           B(int l, int m, int n, int o)
           {
                super(l,m,n);
                d=o;
           }

          void Show()
```

```
        {
                System.out.println("a = " + a);
                System.out.println("b = " + b);
                 System.out.println("c = " + c);
                 System.out.println("d = " + d);
         }
}
class Mypgm
{
        public static void main(String args[])
        {
                B b = new B(4,3,8,7);
                  b.Show();
        }
 }
```

OUTPUT

a = 4
b = 3
c = 8
d = 7

Method Overriding

- ✓ Method overriding in java means a subclass method overriding a super class method.
- ✓ Superclass method should be non-static. Subclass uses extends keyword to extend the super class. In the example **class B** is the sub class and **class A** is the super class. **In overriding methods of both subclass and superclass possess same signatures**. Overriding is used in modifying the methods of the super class. In overriding return types and constructor parameters of methods should match.

Below example illustrates method overriding in java.

Example:

```
class A
{
            int i;
            A(int a, int b)
            {
                  i = a+b;
            }
           void add()
           {
                System.out.println("Sum of a and b is: " + i);
            }
}
class B extends A
{
            int j;
            B(int a, int b, int c)
            {
                 super(a, b);
                 j = a+b+c;
            }
         void add()
         {
               super.add();
               System.out.println("Sum of a, b and c is: " + j);
         }
}
class MethodOverriding
{
          public static void main(String args[])
          {
                  B b = new B(10, 20, 30);
                    b.add();
          }
}
```

OUTPUT

Sum of a and b is: 30
Sum of a, b and c is: 60

Method Overloading

- ✓ Two or more methods have the same names but different argument lists. The arguments may differ in type or number, or both. However, the return types of overloaded methods can be the same or different is called **method overloading**. An example of the method overloading is given below:

Example:

```
class MethodOverloading
{
          int add( int a,int b)
          {
               return(a+b);
          }
          float add(float a,float b)
          {
               return(a+b);
          }
          double add( int a, double b,double c)
          {
              return(a+b+c);
          }
}
class MainClass
{
          public static void main( String arr[] )
          {
                MethodOverloading mobj = new MethodOverloading
                (); System.out.println(mobj.add(50,60));
                System.out.println(mobj.add(3.5f,2.5f));
                System.out.println(mobj.add(10,30.5,10.5));
          }
}
OUTPUT
110
6.0
51.0
```

Abstract Class

- ✓ **abstract keyword** is used to make a class abstract.
- ✓ Abstract class can't be instantiated with new operator.
- ✓ We can use abstract keyword to create an abstract method; an abstract method doesn't have body.
- ✓ If classes have abstract methods, then the class also needs to be made abstract using abstract keyword, else it will not compile.
- ✓ Abstract classes are used to provide common method implementation to all the subclasses or to provide default implementation.

Example Program:

```
abstract Class AreaPgm
{
        double dim1,dim2;
        AreaPgm(double x,double y)
        {
                dim1=x;
                dim2=y;
        }
        abstract double area();
}
class rectangle extends AreaPgm
{
        rectangle(double a,double b)
        {
                super(a,b);
        }
        double area()
        {
                System.out.println("Rectangle Area");
                return dim1*dim2;
        }
}
class triangle extends figure
{
        triangle(double x,double y)
        {
                super(x,y);
```

```
        }
        double area()
        {
                System.out.println("Traingle Area");
                return dim1*dim2/2;
        }
}
class MyPgm
{
        public static void main(String args[])
        {

        AreaPgm a=new AreaPgm(10,10); // error, AreaPgm is a abstract class.

                rectangle r=new rectangle(10,5);
                System.out.println("Area="+r.area());

                triangle t=new triangle(10,8);
                AreaPgm ar;
                ar=obj;
                System.out.println("Area="+ar.area());
        }
}
```

final Keyword In Java

The **final keyword** in java is used to restrict the user. The final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

1) final variable: If you make any variable as final, you cannot change the value of final variable(It will be constant).

Example: There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
class Bike
{
        final int speedlimit=90;//final variable
       void run()
       {
                speedlimit=400;
        }
}
Class MyPgm
{
        public static void main(String args[])
       {
                Bike obj=new  Bike();
               obj.run();
        }
}
```

Output:Compile Time Error

---

2) final method: If you make any method as final, you cannot override it.

Example:

```
class Bike
{
       final void run()
       {
               System.out.println("running");
       }
}
class Honda extends Bike
{
       void run()
       {
               System.out.println("running safely with 100kmph");
       }
}
```

```
Class MyPgm
{
        public static void main(String args[])
        {
                Honda honda= new Honda();
                honda.run();
        }
}
```

Output:Compile Time Error

---

3) final class: If you make any class as final, you cannot extend it.

Example:

```
final class Bike
{

}

class Honda extends Bike
{
          void run()
          {
                System.out.println("running safely with 50kmph");
           }
}

Class MyPgm
{

          public static void main(String args[])
        {
                 Honda honda= new Honda();
                        honda.run();
        }
}
```

Output:Compile Time Error

Exception handling:

Introduction

An **Exception**, It can be defined as an abnormal event that occurs during program execution and disrupts the normal flow of instructions. The abnormal event can be an error in the program.

Errors in a java program are categorized into two groups:

1. **Compile-time errors** occur when you do not follow the syntax of a programming language.
2. **Run-time errors** occur during the execution of a program.

Concepts of Exceptions

An exception is a run-time error that occurs during the execution of a java program.

Example: If you divide a number by zero or open a file that does not exist, an exception is raised.

In java, exceptions can be handled either by the java run-time system or by a user-defined code. When a run-time error occurs, an exception is thrown.

The unexpected situations that may occur during program execution are:

- Running out of memory
- Resource allocation errors
- Inability to find files
- Problems in network connectivity

Exception handling techniques:

Java exception handling is managed via five keywords they are:

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

Exception handling Statement Syntax

Exceptions are handled using a try-catch-finally construct, which has the Syntax.

```
try
{
  <code>
}
catch (<exception type1> <parameter1>)
{
  // 0 or more<statements>

}
finally
{
  // finally block<statements>
}
```

1. try Block: The java code that you think may produce an exception is placed within a try block for a suitable catch block to handle the error.

If no exception occurs the execution proceeds with the finally block else it will look for the matching catch block to handle the error.

Again if the matching catch handler is not found execution proceeds with the finally block and the default exception handler throws an exception.

2. catch Block: Exceptions thrown during execution of the try block can be caught and handled in a catch block. On exit from a catch block, normal execution continues and the finally block is executed (Though the catch block throws an exception).

3. finally Block: A finally block is always executed, regardless of the cause of exit from the try block, or whether any catch block was executed. Generally finally block is used for freeing resources, cleaning up, closing connections etc.

Example:

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

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

This would produce following result:

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

Multiple catch Blocks:

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

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

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try.

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

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

                try
                {
                        int a=args.length;
                        System.out.println("a="+a);
                         int b=50/a;
                         int c[]={1}
                }
                catch (ArithmeticException e)
                {
                      System.out.println ("Division by zero");
                }

                catch (ArrayIndexOutOfBoundsException e)
                {
                       System.out.println (" array index out of bound");
                }
     }
}

OUTPUT
Division by zero
array index out of bound
```

Nested try Statements

- ✓ Just like the multiple catch blocks, we can also have multiple try blocks. These try blocks may be written independently or we can nest the try blocks within each other, i.e., keep one try-catch block within another try-block. The program structure for nested try statement is:

Syntax

```
try
{
   // statements
   // statements

   try
   {
      // statements
      // statements
   }
   catch (<exception_two> obj)
   {
      // statements
   }

   // statements
   // statements
}
catch (<exception_two> obj)
{
   // statements
}
```

- ✓ Consider the following example in which you are accepting two numbers from the command line. After that, the command line arguments, which are in the string format, are converted to integers.

- ✓ If the numbers were not received properly in a number format, then during the conversion a NumberFormatException is raised otherwise the control goes to the next try block. Inside this second try-catch block the first number is divided by the second number, and during the calculation if there is any arithmetic error, it is caught by the inner catch block.

Example

```
class Nested_Try
{
   public static void main (String args [ ] )
   {
      try
      {
           int a = Integer.parseInt (args [0]);
           int b = Integer.parseInt (args [1]);
           int quot = 0;

           try
           {
              quot = a / b;
              System.out.println(quot);
           }
           catch (ArithmeticException e)
           {
              System.out.println("divide by zero");
           }
       }
       catch (NumberFormatException e)
       {
           System.out.println ("Incorrect argument type");
       }
   }
}
```

The output of the program is: If the arguments are entered properly in the command prompt like:

OUTPUT

```
java Nested_Try 2 4 6
 4
```

If the argument contains a string than the number:

OUTPUT

```
java  Nested_Try 2 4  aa
 Incorrect argument type
```

If the second argument is entered zero:

OUTPUT

```
java Nested_Try 2 4 0
 divide by zero
```

throw Keyword

- ✓ **throw keyword** is used to throw an exception explicitly. 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*

Creating Instance of Throwable class

There are two possible ways to get an instance of class Throwable,

1. Using a parameter in catch block.
2. Creating instance with **new** operator.

```
new NullPointerException("test");
```

This constructs an instance of NullPointerException with name test.

Example demonstrating throw Keyword

```
class Test
{
        static void avg()
        {
                try
                {
                        throw new ArithmeticException("demo");
                }
                catch(ArithmeticException e)
                {
                        System.out.println("Exception caught");
                }
        }
        public static void main(String args[])
        {
                avg();
        }
}
```

In the above example the avg() method throw an instance of ArithmeticException, which is successfully handled using the catch statement.

---

throws Keyword

- ✓ Any method capable of causing exceptions must list all the exceptions possible during its execution, so that anyone calling that method gets a prior knowledge about which exceptions to handle. A method can do so by using the **throws** keyword.

Syntax :

```
type method_name(parameter_list) throws exception_list
{
        //definition of method
}
```

NOTE : It is necessary for all exceptions, except the exceptions of type **Error** and **RuntimeException**, or any of their subclass.

Example demonstrating throws Keyword

```
class Test
{
        static void check() throws
        ArithmeticException {
                System.out.println("Inside check function");
                throw new ArithmeticException("demo");
        }

        public static void main(String args[])
        {
                try
                {
                        check();
                }
                catch(ArithmeticException e)
                {
                        System.out.println("caught" + e);
                }
        }
}
```

finally

- ✓ The finally clause is written with the try-catch statement. It is guaranteed to be executed after a catch block or before the method quits.

Syntax

```
try
{
   // statements
}
```

```
catch (<exception> obj)
{
   // statements
}
finally
{
   //statements
}
```

- ✓ Take a look at the following example which has a catch and a finally block. The catch block catches the ArithmeticException which occurs for arithmetic error like divide-by-zero. After executing the catch block the finally is also executed and you get the output for both the blocks.

Example:

```
class Finally_Block
{
   static void division ( )
   {
      try
      {
         int num = 34, den = 0;
         int quot = num / den;
      }
      catch(ArithmeticException e)
      {
         System.out.println ("Divide by zero");
      }
      finally
      {
         System.out.println ("In the finally block");
      }
   }
class Mypgm
{
   public static void main(String args[])
   {
```

```
        Finally_Block   f=new Finally_Block();
        f.division ( );
    }
}

OUTPUT
Divide by zero
In the finally block
```

Java's Built in Exceptions

Java defines several exception classes inside the standard package **java.lang**.

- ✓ The most general of these exceptions are subclasses of the standard type **RuntimeException**. Since **java.lang** is implicitly imported into all Java programs, most exceptions derived from **RuntimeException** are automatically available.

Java defines several other types of exceptions that relate to its various class libraries. Following is the list of Java **Unchecked RuntimeException**.

Exception	Description
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.

IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
UnsupportedOperationException	An unsupported operation was encountered.

Following is the list of **Java Checked Exceptions** Defined in java.lang.

Exception	Description
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

Creating your own Exception Subclasses

- ✓ Here you can also define your own exception classes by extending **Exception**. These exception can represents specific runtime condition of course you will have to throw them yourself, but once thrown they will behave just like ordinary exceptions.
- ✓ When you define your own exception classes, choose the ancestor carefully. Most custom exception will be part of the official design and thus checked, meaning that they extend Exception but not RuntimeException.

Example: Throwing User defined Exception

```
public class MyException extends Exception
{
    String msg = "";
    int marks=50;
    public MyException()
    {
    }
    public MyException(String str)
    {
       super(str);
    }
    public String toString()
    {
       if(marks <= 40)
       msg = "You have failed";
       if(marks > 40)
       msg = "You have Passed";

       return msg;

    }
}
```

```
class test
{
   public static void main(String args[])
      {
            test t = new test();
            t.dd();
      }
   public void add()
    {
        try
         {
            int i=0;
            if( i<40)
            throw new MyException();
         }
         catch(MyException ee1)
         {
             System.out.println("Result:"+ee1);
         }
     }
}
```

OUTPUT

Result: You have Passed

Chained Exception

- ✓ Chained exceptions are the exceptions which occur one after another i.e. most of the time to response to an exception are given by an application by throwing another exception.
- ✓ Whenever in a program the first exception causes an another exception, that is termed as **Chained Exception**. Java provides new functionality for chaining exceptions.
- ✓ Exception chaining (also known as "nesting exception") is a technique for handling the exception, which occur one after another i.e. most of the time

is given by an application to response to an exception by throwing another exception.

- Typically the second exception is caused by the first exception. Therefore chained exceptions help the programmer to know when one exception causes another.

The **constructors** that support chained exceptions in **Throwable** class are:

Throwable initCause(Throwable)

Throwable(Throwable)

Throwable(String, Throwable)

Throwable getCause()

Packages in JAVA

- ✓ 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.

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.

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 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

Access Modifiers/Specifiers

The access modifiers in java specify accessibility (scope) of a data member, method, constructor or class.

There are 4 types of java access modifiers:

1. private
2. default
3. protected
4. public

1) private access modifier

The private access modifier is accessible only within class.

2) default access modifier

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package.

3) protected access modifier

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

4) public access modifier

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

Understanding all java access modifiers by a simple table.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Interface in java

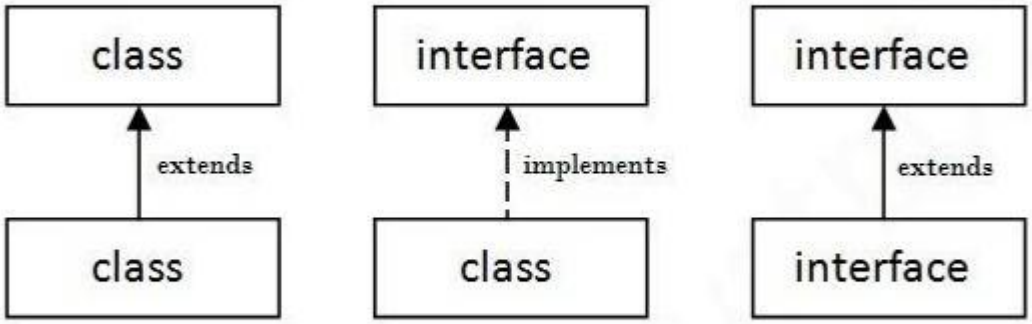
- ✓ An **interface in java** is a blueprint of a class. It has static final variables and abstract methods.
- ✓ The interface in java is **a mechanism to achieve abstraction**. There can be only abstract methods in the java interface does not contain method body. It is used to achieve abstraction and multiple inheritance in Java.
- ✓ It cannot be instantiated just like abstract class.
- ✓ Interface fields are public, static and final by default, and methods are public and abstract.

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

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

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**.



Example 1

In this example, Printable interface has only one method, its implementation is provided in the Pgm1 class.

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

class Pgm1 implements printable
{
        public void print()
        {
                System.out.println("Hello");
        }
}

 class IntefacePgm1
{
        public static void main(String args[])
        {
                Pgm1 obj = new Pgm1 ();
                obj.print();
         }
}
```

Output:

Hello

Example 2

In this example, Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In real scenario, interface is defined by someone but implementation is provided by different implementation providers. And, it is used by someone else. The implementation part is hidden by the user which uses the interface.

```
//Interface declaration: by first user

interface Drawable
{
        void draw();
}
//Implementation: by second user

class Rectangle implements Drawable
{
        public void draw()
        {
                System.out.println("drawing rectangle");
        }
}

class Circle implements Drawable
{
        public void draw()
        {
                System.out.println("drawing circle");
        }
}
```

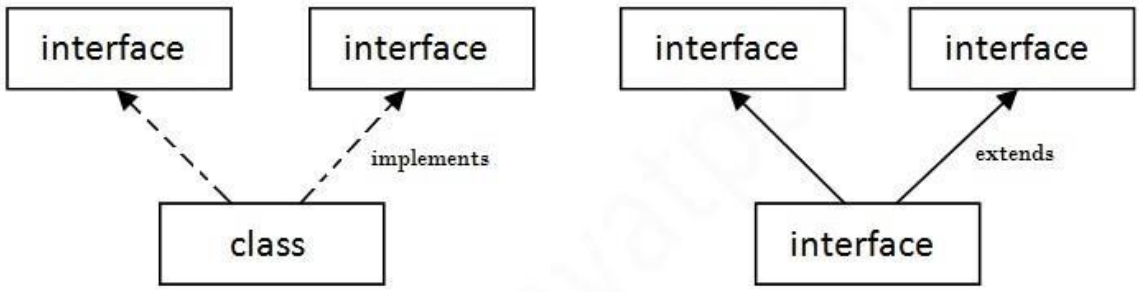
```
//Using interface: by third user
class TestInterface1
{
        public static void main(String args[])
        {
                //In real scenario, object is provided by method e.g. getDrawable()
                Drawable d=new Circle();

                d.draw();
        }
}
```

Output:

drawing circle

Multiple inheritance in Java by interface

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



Multiple Inheritance in Java

Example

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

interface Showable
{
        void show();
}

class Pgm2 implements Printable,Showable
{
        public void print()
        {
                System.out.println("Hello");
        }

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

Class InterfaceDemo
{
        public static void main(String args[])
        {
                Pgm2 obj = new Pgm2 ();
                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 because of ambiguity.
- ✓ But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class. For example:

Example

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

interface Showable
{
        void print();
}

class InterfacePgm1 implements Printable, Showable
{
        public void print()
        {
                System.out.println("Hello");
        }
}
class InterfaceDemo
{
        public static void main(String args[])
        {
                InterfacePgm1 obj = new InterfacePgm1 ();
                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 InterfacePgm2 implements Showable
{
        public void print()
        {
                System.out.println("Hello");
        }
        public void show()
        {
                System.out.println("Welcome");
        }

Class InterfaceDemo2
{
        public static void main(String args[])
        {
                InterfacePgm2 obj = new InterfacePgm2 ();
                obj.print();
                obj.show();
         }
}
```

Output:

```
Hello
Welcome
```

Program to implement Stack

```
public class StackDemo
{
        private static final int capacity = 3;
        int arr[] = new int[capacity];
        int top = -1;

    public void push(int pushedElement)
    {
        if (top < capacity - 1)
        {
            top++;
            arr[top] = pushedElement;
        System.out.println("Element " + pushedElement  + " is pushed to Stack !")
    ;
        printElements();
        }
        else
         {
            System.out.println("Stack Overflow !");
        }
    }

    public void pop()
    {
        if (top >= 0)
         {
            top--;
            System.out.println("Pop operation done !");
        }
        else
        {
            System.out.println("Stack Underflow !");
        }
    }

        public void printElements()
        {
            if (top >= 0)
            {
                System.out.println("Elements in stack :");
                for (int i = 0; i <= top; i++)
                {
                    System.out.println(arr[i]);
                }
```

```
                }
        }

class MyPgm
{

        public static void main(String[] args)
        {
                StackDemo stackDemo = new StackDemo();

                stackDemo.pop();
                stackDemo.push(23);
                stackDemo.push(2);
                stackDemo.push(73);
                stackDemo.push(21);
                stackDemo.pop();
                stackDemo.pop();
                stackDemo.pop();
                stackDemo.pop();
        }

}
```

Output

```
Stack Underflow !
Element 23 is pushed to Stack !
Elements in stack :
23
Element 2 is pushed to Stack !
Elements in stack :
23
2
Element 73 is pushed to Stack !
Elements in stack :
23
2
73
Stack Overflow !
Pop operation done !
Pop operation done !
Pop operation done !
Stack Underflow !
```

Questions

1. Distinguish between Method overloading and Method overriding in JAVA, with suitable examples.(Jan 2014) 6marks
2. What is super? Explain the use of super with suitable example (Jan 2014) 6marks
3. Write a JAVA program to implement stack operations. (Jan 2014) 6marks
4. What is an Exception? Give an example for nested try statements? (Jan 2013) 6 Marks
5. WAP in java to implement a stack that can hold 10 integers values (Jan 2013) 6 Marks
6. What is mean by instance variable hiding ?how to overcome it? (Jan 2013) 04 Marks
7. Define exception .demonstrate the working of nested try blocks with suitable example? (Dec 2011)08Marks
8. Write short notes on (Dec 2011)04Marks
   i) Final class ii) abstract class
9. Write a java program to find the area and volume of a room. Use a base class rectangle with a constructor and a method for finding the area. Use its subclass room with a constructor that gets the value of length and breadth from the base class and has a method to find the volume. Create an object of the class room and obtain the area and volume. (Jan-2006) 8Marks
10. Explain i) Instance variables ii) Class Variables iii) Local variables (Jan-2009) 06 marks

11. Distinguish between method overloading and method overriding? How does java decide the method to call? (Jan-2008-8Marks) 6Marks
12. Explain the following with example.
    i) Method overloading ii) Method overriding (jun-2006) 8Marks

13. Write a java program to find the distance between two points whose coordinates are given. The coordinates can be 2-dimensional or 3-dimensional (for comparing the distance between 2D and a 3D point, the 3D point, the 3D x and y components must be divided by z). Demonstrate method overriding in this program. (May-2007)10 marks
14. What is an interface? Write a program to illustrate multiple inheritance using interfaces. (Jan-2010) 8Marks
15. Explain packages in java.
16. What are access specifiers? Explain with an example.
17. With an example explain static keyword in java.
18. Why java is not support concept of multiple inheritance? Justify with an example program.
19. Write a short note on:
    1. this keyword
    2. super keyword
    3. final keyword
    4. abstract
20. Illustrate constructors with an example program

Module 2
(Objects and arrays, Namespaces, Nested classes, Constructors, Destructors.) covered in module 1.

Introduction to Java: Java's magic: the Byte code; Java Development Kit (JDK); the Java
Buzzwords, Object-oriented programming; Simple Java programs. Data types, variables and
arrays, Operators, Control Statements.

Introduction to Java

Java is a general-purpose object oriented programming language developed by Sun Microsystems.

Two types of Java applications can be created.

1. Stand alone Java application.
2. Web applets.

Stand alone Java application

Stand alone Java applications are programs written in Java to carry out a specific tasks on a stand alone system.

Executing a stand-alone Java program contains two phases:

a. Compiling source code into bytecode using **javac** compiler.

b. Executing the bytecode program using **java** interpreter.

Java environment includes a large number of development tools and hundreds of classes and methods.

Java development tools are part of the systems known as Java development kit (JDK) and the classes and methods are part of the Java standard library known as **Java standard Library (JSL)** also known as **Application Program Interface (API)**.

Java Features/Java Buzzwords:

1. Compiler and Interpreted
2. Architecture Neutral/Platform independent and portable
3. Object oriented
4. Robust and secure.
5. Distributed.
6. Multithreaded and interactive.
8. Dynamic and extendible.

1. Compiled and Interpreted:

Usually a computer language is either compiled or interpreted. Java combines both these approaches;

First java compiler translates source code into bytecode instructions. Bytecodes are not machine instructions,

In second step, java interpreter generates machine code that can be directly executed by the machine that is running the java program.

2. Architecture Neutral/Platform independent and portable

The concept of **Write-once-run-anywhere** (known as the Platform independent) is one of the important key features of java language that makes java as the most powerful language.

The programs written on one platform can run on any platform provided the platform must have the JVM.

3. Object oriented

In java everything is an Object. Java classes can be easily extended since it is based on the Object model. Java is a pure object oriented language.

4. Robust and secure

Java is a robust language; (Robust programming is **a style of programming that focuses on handling unexpected termination and unexpected actions**.)

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

Absence of pointers in java, accounts for full data security.

5. Distributed

Java is designed for the distributed environment of the internet.

7. Multithreaded and interactive

With Java's multi-threaded feature it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct smoothly running interactive applications.

8. Dynamic and extendible

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 an extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

Java Development kits(jdk)

Java development kit has a number of Java development tools.

(1) Appletviewer: Enables to run Java applet.
(2) javac: Java compiler.
(3) java: Java interpreter.
(4)javah :Produces header files for use with native methods.
(5) javap :Java disassembler.
(6) javadoc :Creates HTML documents for Java source code file.
(7) jdb :Java debugger which helps us to find the error.

Introduction to Java : **Java Program**

File name : **First.java**

```
class First
{
  public static void main(String args[ ])
  {
     System.out.println(“Welcome to JAVA”);
  }
}
```

“class First” declares a class, which is an object-oriented construct. **First** is a Java identifier that specifies the name of the class.

A single java program file can have ‘n’ number of classes. **But the name of the class which contains main function must match the name of the file.** Hence, this program name is First.java

Introduction to Java : First Java Program

Since, Java is a pure object oriented programming language, even main function must be declared inside the class.

Since, main is public and static no need of creating an object of type class First, the executor in this JVM (Java Virtual Machine) will call the main function by using class name.
Ex:

```
First.main();
```

Execution of a java program starts from main function.

public key word is an access specifier that declares the main method accessible by executor.

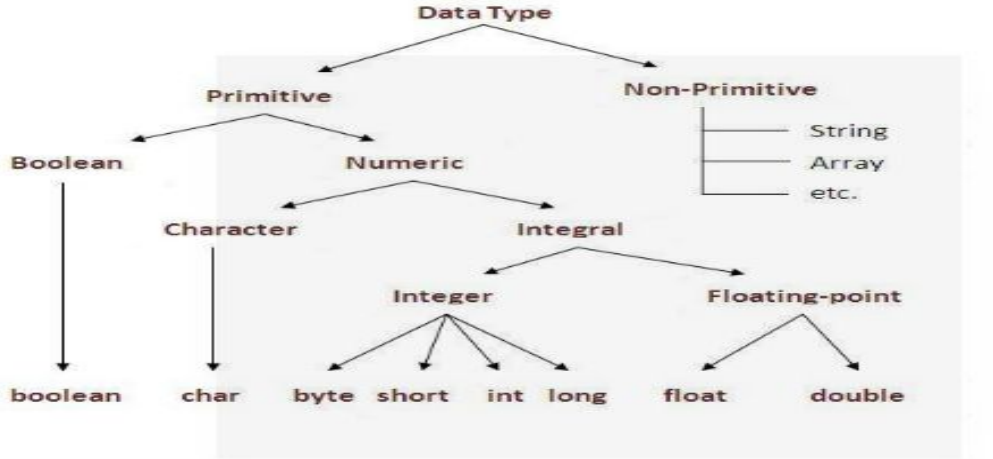
static keyword defines the method as one that belongs to the entire class and not for a particular object of the class. **main must always be declared as static.**

The type modifier void specifies that the method main is not returning any value.

Introduction to Java : **Data types**

In java, data types are classified into two categories

1.Primitive & 2.Non-Primitive Data type



Data Type	Default Value	Default size
boolean	False	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Integer Type

Java provides four types of Integers. They are byte, short, int, long. All these are sign, positive or negative.

byte: smallest integer type is byte. This is a signed 8-bit type that has a range from –128 to 127. A byte variable is declared with the keyword "byte". Ex: byte b, c;

short: short is a signed 16-bit type. It has a range from –32768 to 32767. Specially used in 16 bit computers. Short variables are declared using the keyword short. Ex: short a, b;

int: Most commonly used Integer type is int. Signed 32 bit type has a range from –2147483648 to 2147483648. Ex: int a, b, c;

long: long is a 64 bit type and useful in all those occasions where int type of memory is not enough. Ex: long a, b;

Floating point types

Floating point numbers also known as real numbers are useful when evaluating an expression that requires fractional precision. The two floating-point data types are float and double.

float: float type specifies a single precision value that uses 32-bit storage. float keyword is used to declare a floating point variable. Ex: float a, b;

double: Double type of variable is declared with double keyword and uses 64-bit value.
Ex: double c;

characters: Java data type to store characters is char. char data type of Java uses Unicode to represent characters. Unicode defines a fully international character set that can have all the characters of human language. Java char is a 16-bit type. The range is 0 to 65536.
Ex: char a='p';

boolean: Java has a type called boolean for logical values. It can have only one of two possible values. They are true or false. Ex: boolean a; a=true;

Stack memory is used to **store primitive type values and the addresses of objects.**

Non-primitive types
A variable of type class is termed as non-primitive types in java.

All variables/objects of type class must acquire memory dynamically using **new** operator in java.

There is no delete operator to reclaim the memory that has already been allocated.

The job of reclaiming memory will be done by a system program, which is part of JVM termed as **garbage collector.**

Introduction to Java : **Non-primitive types**

Ex:

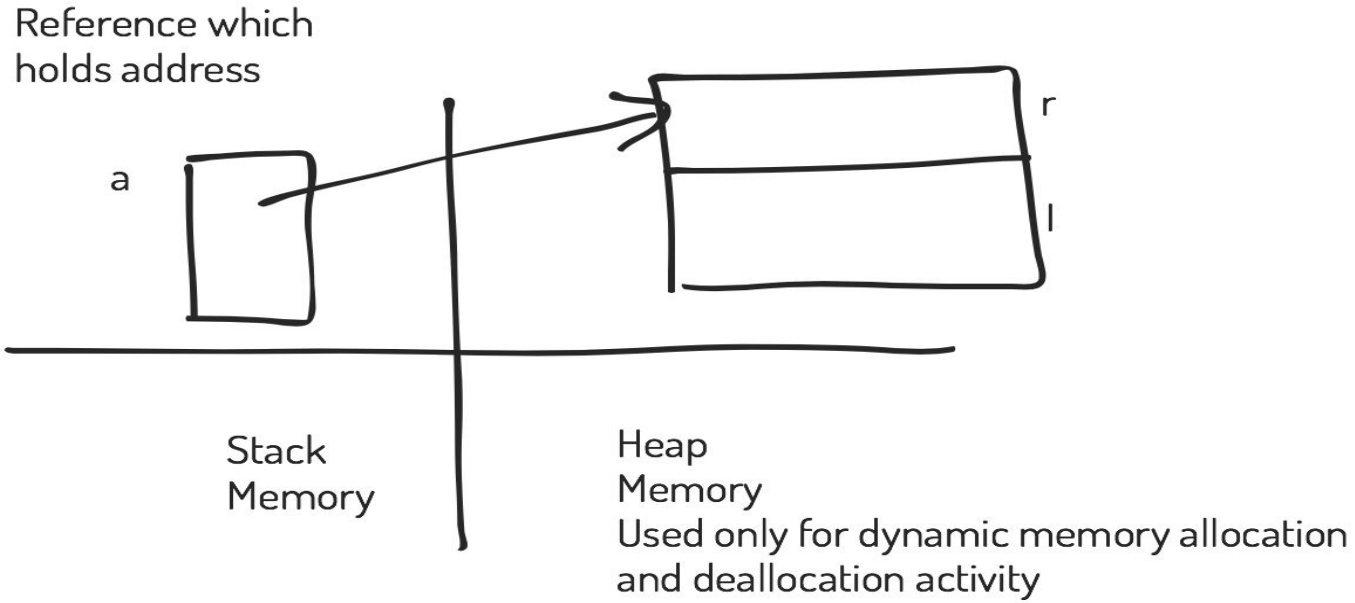
```
class complex {
        private  int r, i;     // no colon after private/public keyword.

        public void display( )
        {
                System.out.println("In display function ");
        }
}  // no ; required

class test {
        public static void main(String args[] ) {
                complex   a;
                a  = new complex( );
                a.display( );
        }
}
```

Introduction to Java : **Non-primitive types**

In the statement "complex a", 'a' is termed as reference.

Reference definition

A reference is **an address that indicates where an object's variables and methods are stored**.



Since, address cannot be accessed explicitly in Java, members of class are referred by only (.) dot operator.

Memory which is allocated to store information of type complex in heap will be reclaimed automatically, when the program is about to conclude its execution.

Glimpse of Inheritance

Inheritance is one of the main pillar in any OOP language, and hence java too.

Inheritance helps one class to inherit the properties of another.

Properties of a class are nothing but DM, MF, SDM, SMF, which can be inherited and used for other purpose.

"extends" is the keyword used to inherit the properties from one class to another.

Introduction to Java : Glimpse of Inheritance

Class, which shares its attributes are termed as base class.

Class, which receives the attributes are termed as derived class.

Ex:

```
class base   {
        public void print()  {
                System.out.println("In base print");
        }
}

class drd extends base {
        public void display() {
                System.out.println("In drd display( )");
        }
}
```

Ex:

```
class Test {
    public static void main(String[] args){
        drd a = new drd( );      a.print();
    }
}
```

Understanding "System.out.println" statement in java

Java **System.out.println()** is used to print an argument that is passed to it.

The statement can be broken into 3 parts which can be understood separately as:

System: It is a final class defined in the java.lang package.

If a class is declared as final it cannot be inherited further

out: This is an instance of **PrintStream** type, **out** is a public and static member field of the System class.

Ex:

```
package java.lang;
class PrintStream
{
  // task of PrintStream is to communicate with standard output device
 public void Println()
 {
   ……...
 }
}
```

```
final class System
   {
       public static PrintStream out;  //has a kind of relation -

      public static InputStream in;

       public System()
       {  out = new PrintStream(); }
   }
```

println(): PrintStream class has a public method **println()**, hence we can invoke the same on "out" as well. This is an upgraded version of print(). It prints any argument passed to it and adds a new line to the output. System.out represents the Standard Output Stream.

COMMAND LINE ARGUMENTS in C

Arguments that are passed to main program

Any type of value passed via CL will be stored as strings/char arrays in memory.

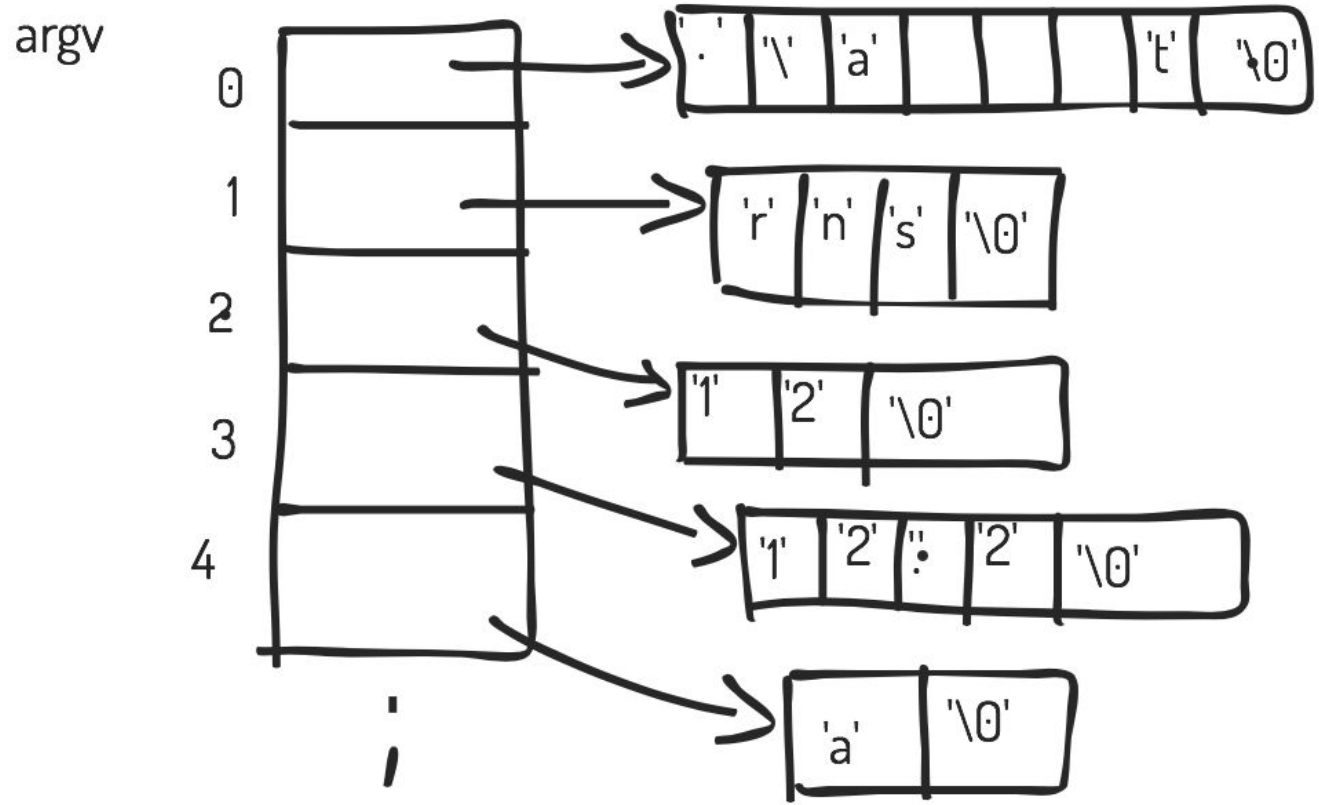
Two arguments are there in C/C++ which represents command line.

int main(int argc, char * argv[ ])

First argument maintains argument count
Second argument maintains information about arguments in string format.

By default argc value will be 1, representing the path of the executable file.
Ex: executable file in unix versions are ./a.out
In windows os executable file will be .exe extension, having the .c file name as the primary one. (.exe is secondary file name)

COMMAND LINE ARGUMENTS in C



COMMAND LINE ARGUMENTS in C

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;
    printf("%d\n",argc);
    for(i=0;i<argc;i++)
      printf("%s\n",argv[i] );
    return 0;
}
```

Introduction to Java : **Built-in class String**

String is a build in class in java to store a string or set of character informations within it.

Since, String is a class a reference must be created and then object must be created using new operator.

The following constructor is associated with the String class in java.

String()

Initializes a newly created String object so that it represents an empty character sequence.

Ex: String a = new String( ); System.out.println(a);

Reference ‘a’ is basically pointing to empty string set.

a=”rnsit” will allocate sufficient dynamic memory to store rnsit and that memory will be referenced by a.

Ex:

String a = new String(“rnsit”);

String class contains a parameterized constructor which receives “rnsit” as a value, memory is allocated to store the passed value, and the passed value is copied to the acquired memory.

Values stored in String instances such as ‘a’ are **immutable.**

a[0] =’J’; cannot be done.

A new set of values can be assigned to ‘a’, but older values cannot be modified in terms of characters.

a[0] = ‘R’; not allowed

a= a + “ engineering”; allowed; here old value is combined with new string

a = “rnsit engineering college” allowed; here old value is replaced by new string.

(Remaining constructors of String class will be considered afterwards).

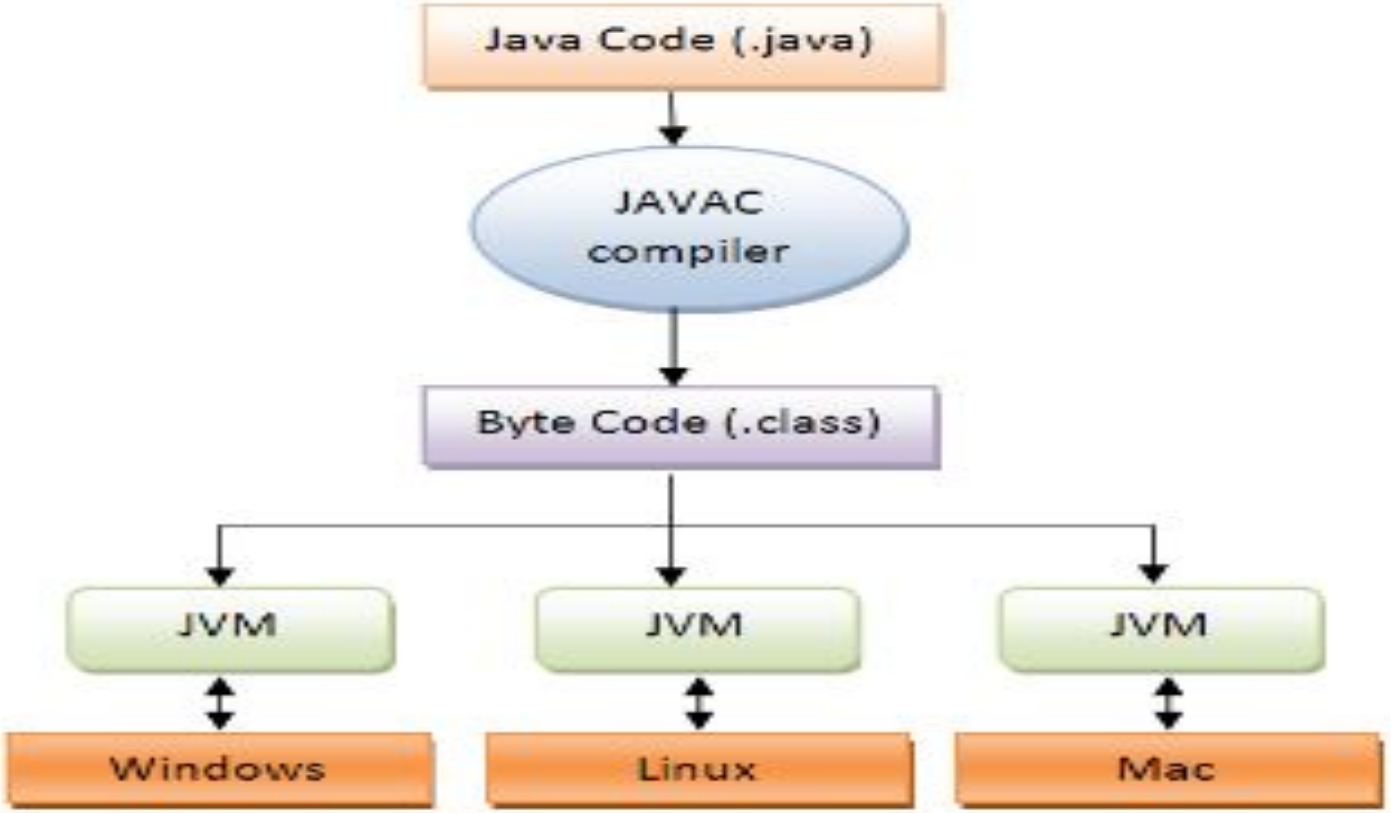
Introduction to Java : **JVM (Java Virtual Machine)**

The concept of Write-once-run-anywhere (known as the Platform independent) is one of the important key features of java language that makes java as the most powerful language.

The programs written on one platform can run on any platform provided the platform must have the JVM(Java Virtual Machine).

A Java virtual machine(JVM) is a virtual machine that can execute Java bytecode. It is the code execution component of the Java software platform.

Introduction to Java : **JVM (Java Virtual Machine)**



Introduction to Java : **Java Program structure**

Java program structure contains six stages.

(1) Documentation section:

The documentation section contains a set of comment lines describing the program.

(2) Package statement:

The first statement allowed in a Java file is a package statement. This statement declares a package name and informs the compiler that the classes defined here belong to a package.

package student; // student is a directory in java

A package is a namespace that organizes a set of related classes.

Package are nothing but different directories in computer.

(3) Import statements:

Import statements instruct the compiler to load the specific class, which belongs to the mentioned package.

import student.test; student is main directory test is subdirector

Package or directory word are synonym in java

All the classes in test subdirectory will be imported or included to the source file, before compilation.

(4) Class definition:

A Java program may contain multiple class definitions.

(5) Main method class:

The main method creates objects of various classes and establishes communication between them. On reaching the end of the main the program terminates and the control goes back to the operating system.

Introduction to Java : **Accepting values from console**

In order to accept values from console built-in class Scanner must be used.

```
Scanner  sc = new Scanner(System.in);
```

After instantiating the object of type Scanner, the constructor of scanner will be called by passing **System.in** as the parameter. **in** is a public static member of type InputStream.

It is as binding the Scanner object with the standard input device, such as keyboard.

```
int n = sc.nextInt();
```

nextInt is a member function which is invoked using sc object.

The above statement accept's a value of type integer from the standard input device i.e keyboard.

Introduction to Java : **Accepting values from console**

Scanner class is widely used to parse/read text for strings and primitive types.

It is the simplest way to get input in Java.

Scanner class is declared in java.util package.

The facilities provided by the System class are the code to communicate with standard input, standard output, and error output streams;

All classes in java whether built-in or user defined, they inherit the properties of Supreme base class **Object**.

Fields of System class are:

1. **public static final InputStream in:**

   This stream is already open and ready to supply input data from standard input devices. Typically this stream corresponds to keyboard input or another input source specified by the host environment or user.

2. **public static final PrintStream out:**

3. **public static final PrintStream err:**

“final“ keyword is used to create constants in java. No constant keyword in java.

Ex:

```
class test   {
    public static void main(String[] args) {
            final int i=90;
            i = 900; // generates CTE
    }   }
```

Introduction to Java : **Arrays in Java**

Syntax for declaring an array variable or reference to array:

dataType[ ] arrayRef; OR dataType arrayRef[ ];

int[ ] myList; OR int myList [ ]; // 1-d array

Further memory will be allocated to store array element type using new operator.

arrayRef = new dataType[arraySize];

The above statement does two things:

It creates an array using new dataType[arraySize];

It assigns the reference of the newly created array to the variable arrayRefVar.

```
int [ ]  myList;
…….
….
  myList = new int[4];
```

All these can be combined into one statement as shown below.

dataType[ ] arrayRefVar = new dataType[arraySize];

```
int [ ]  myList = new int[4];
```

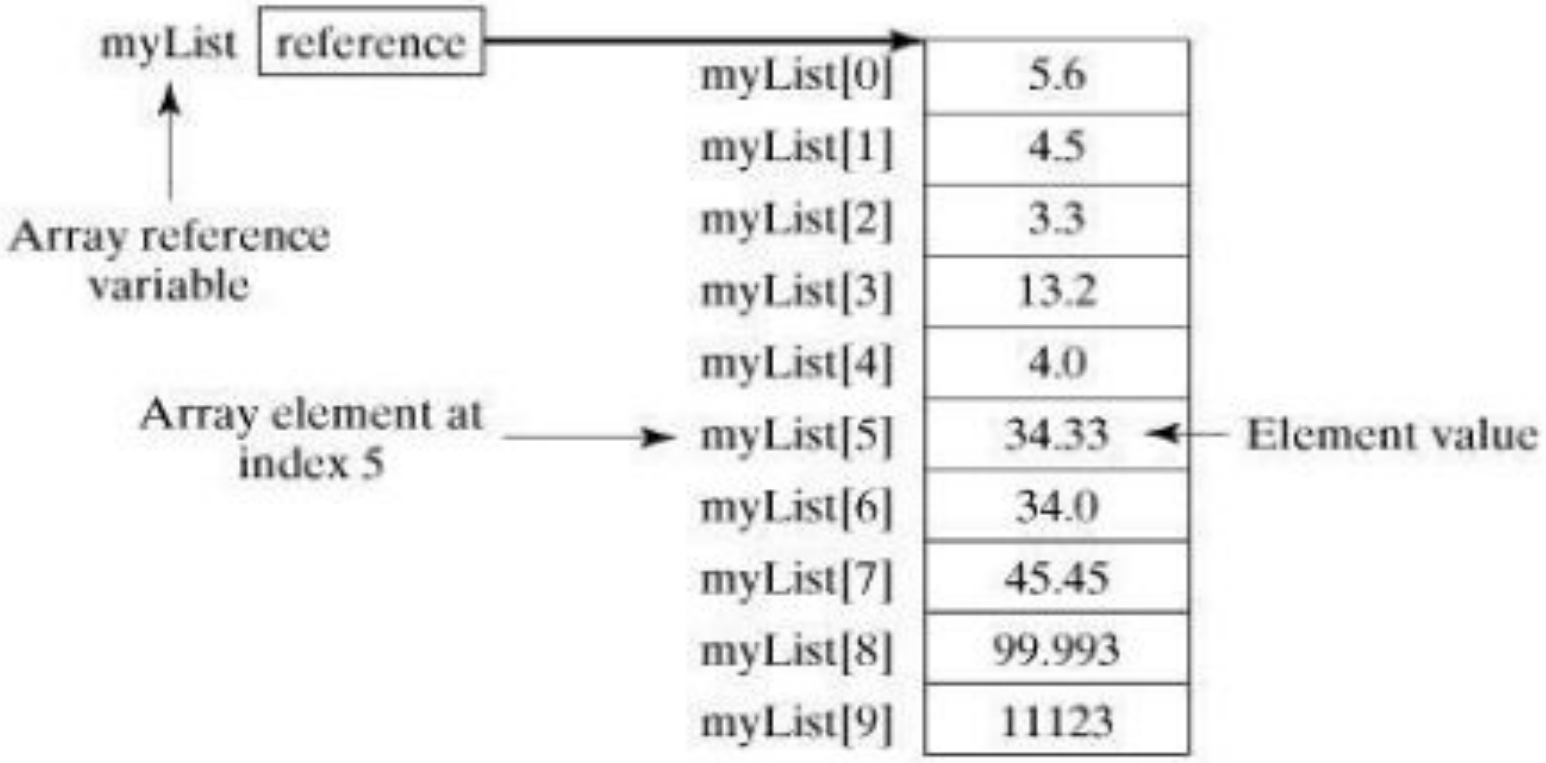
Alternatively you can create arrays and initialize as follows:

dataType[] arrayRefVar = {value0, value1, ..., valuek};

Ex:

```
int [ ] myList = {1, 2, 3, 4};

double[ ] myList = {5.6, 4.5, 3.3, 13.2, 4.0, 34.33, 34.0, 45.45, 99.993, 11123};
```

Initial values are counted and that will be the size of the array.


myList
reference
Array reference variable
Array element at index 5
myList[0]
myList[1]
myList[2]
myList[3]
myList[4]
myList[5]
myList[6]
myList[7]
myList[8]
myList[9]
5.6
4.5
3.3
13.2
4.0
34.33
34.0
45.45
99.993
11123
Element value

```
Ex:   class test  {
            public static void main( String[] argos) {
                 int [ ] a = {1, 2, 3, 4};

                for(int i=0;  i<4; i++)
                      System.out.println(a[i]);

                for(int i=0;   i<a.length;   i++)
                      System.out.println( a[i] );

                a[5]=50; //generates RTE ArrayIndexOutOfBoundsException
           }   }
```

length is the property associated with any type of arrays created in java.

length conveys the number of elements that can be kept in an array.

Array size must be considered in java programs, if not RTE will be generated.

1-D Arrays

Java program to accept n integer values and store it in an array and print it.

Java program to accept n char values and store it in an array and print it.

```
String b;
System.out.println("Enter "+n+" different values");
for(int i=0;i<n;i++)
{
        b= sc.next();
        a[i]=b.charAt(0);
}
```

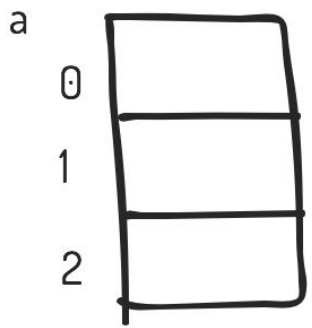
Java program to accept n student values store it in array and print it.

Java program to accept n student values store it in array and print it.

```
import java.util.*
class student  {
        private String name,usn;
        private static Scanner sc;
        public static void create_sc()
        {  sc = new Scanner(System.in); }

        public void accept()
        {
                System.out.println("Enter name ");
                name=sc.nextLine();
                System.out.println("Enter usn ");
                usn=sc.next();
        }
        public void display()
        {  System.out.println(name+" "+usn);  }        }
```

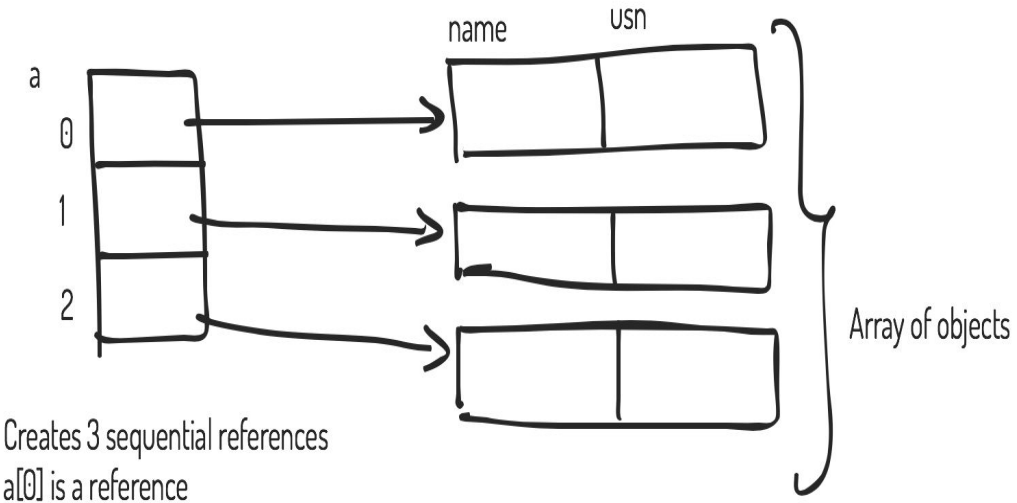
```
public class First {
    public static void main(String[] args) {
        student.create_sc();
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        student [] a = new student[n];   //  1

        for(int i=0;i<a.length;i++)
            a[i] = new student();        //  2

        System.out.println("Enter values for array ");
        for(int i=0;i<a.length;i++)
            a[i].accept();

        System.out.println("Array values are ");
        for(int i=0;i<a.length;i++)
            a[i].display();                                  }          }
```

student [ ]. a. =. new student [3];



Creates 3 sequential references
a[0] is a reference
a[1] is a reference
and a[2] is also a reference

student [ ]. a. =. new student [3];



Creates 3 sequential references
a[0] is a reference
a[1] is a reference
and a[2] is also a reference

2-D Arrays

Java program to accept n integer values store it in an 2d-array and print it.

Java program to accept n char values store it in an 2d-array and print it.

Java program to accept n student values store it in 2d-array and print it.

Introduction to Java : **Arrays in Java - 2-D Arrays**

2-D arrays are a collection of 1-d arrays in java.

GF: <data-type> [][] <2d-array-name> = new <data-type> [row-size][column-size];

Ex: int [ ][ ] a = new int [3][2];

char [ ][ ] b = new char[3][2];

Numerical 2-d array

```
Scanner sc = new Scanner(System.in);
int m,n,i,j;
m=sc.nextInt();
n=sc.nextInt();

int [][] a = new int[m][n];

for(i=0;i<m;i++)
        for(j=0;j<n;j++)
                a[i][j] = sc.nextInt();

for(i=0;i<m;i++)
{
        for(j=0;j<n;j++)
                System.out.print(a[i][j]+"\t");
        System.out.println();
}
```

Character 2-d array

```
Scanner sc = new Scanner(System.in);
int m,n,i,j;
m=sc.nextInt();
n=sc.nextInt();

char [][] a = new char[m][n];

for(i=0;i<m;i++)
        for(j=0;j<n;j++)
                a[i][j] = sc.next().charAt(0);

for(i=0;i<m;i++)
{
        for(j=0;j<n;j++)
                System.out.print(a[i][j]+"\t");
        System.out.println();
}
```

Student 2-d array

```
package first;
import java.util.*;
class student      {
        private String name,usn;
        private static Scanner sc;
        public static void create_sc()
        {       sc = new Scanner(System.in);      }
        public void accept()   {
                System.out.println("Enter name ");
                name=sc.next();
                System.out.println("Enter usn ");
                usn=sc.next();
        }
        public void display()  {
                System.out.print("[ "+name+","+usn+" ]");   }
}
```

Student 2-d array

```
public class First {

    public static void main(String[] args) {
        student.create_sc();
        Scanner sc = new Scanner(System.in);
        int m,n,i,j;
        m = sc.nextInt();          n = sc.nextInt();

        student [][] a = new student [m][n];

        for(i=0;i<a.length;i++)
          for(j=0;j<a[i].length;j++)
              a[i][j] = new student();
```

Student 2-d array

```
            System.out.println("Enter values for array ");
            for(i=0;i<a.length;i++)
                  for(j=0;j<a[i].length;j++)
                        a[i][j].accept();

            System.out.println("Array values are ");
            for(i=0;i<a.length;i++)   {
                  for(j=0;j<a[i].length;j++)
                        {   a[i][j].display();    System.out.print(" ");   }
                  System.out.println();
            }
      }
}
```

Introduction to Java : **for-each loop in Java**

for-each enables to traverse the complete array sequentially without using an index variable.

GF:

```
for( type  var : array)
{
      Statements using var;
}
```

var just receives the value of array from 0th index to length-1 index in each iteration of the foreach loop.

value in var can be used by the statements inside foreach loop.

If ‘var’ is modified array contents will not be modified.

Introduction to Java : **for-each loop in Java**

Ex:

```
int [] a = {11,22,33,44};

for(int   i  :  a)
        System.out.println(i);
```

Limitations of foreach loop

1. For-each loops are **not appropriate when array contents are to be modified.**

2. For-each loops do not keep track of index. Hence, array index cannot be obtained using For-Each loop.

3. For-each only iterates forward over the array in single steps.

4. For-each **cannot process two decision making statements** at once

Introduction to Java : Printing array of strings using foreach

Ex:

```
String [] a = {"rnsit", "msrit", "rvce"};

for(String i : a)
        System.out.println(i);
```

Ex:

```
int [][]  a = new int[3][2];
int i,j,k=1;

for(i=0;i<a.length;i++)
        for(j=0;j<a[i].length;j++)
                a[i][j] = k++;


for(int [] p : a)
        for (int q : p )
                System.out.println(q);
```

Introduction to Java : Type casting in Java

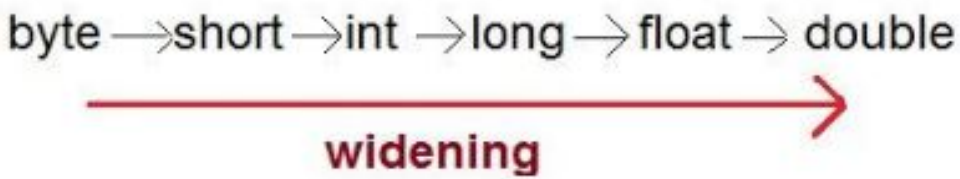
It is often necessary to store a value of one type into the variable of another type.

Type casting can be done in two ways.

Ex:

```
int x = 10;
byte y = (byte)x;
```

In Java, type casting is classified into two types,Widening Casting(Implicit)



Introduction to Java : Type casting in Java

Narrowing Casting(Explicitly done)

Widening or Automatic type conversion

Automatic Type casting take place when, the two types are compatible or the target type is larger than the source type

Ex:

```
class Test  {
  public static void main(String[] args)   {
     int i = 100;
     long l = i; //no explicit type casting required
     float f=l; //no explicit type casting required
     System.out.println("Int value "+i);
    System.out.println("Long value "+l);
    System.out.println("Float value "+f);            }  }
```

Narrowing or Explicit type conversion

When assigning a larger type value to a variable of smaller type, then there is a need to perform explicit type casting.

Ex:

```
class Test {
   public static void main(String[] args) {
     double d = 100.04;
     long l = (long)d; //explicit type casting required
     int i = (int)l;//explicit type casting required
     System.out.println("Double value "+d);
     System.out.println("Long value "+l);
     System.out.println("Int value "+i);
   }
}
```

Java operators:

Java operators can be categorized into following ways:

(1) Arithmetic operator
(2) Relational operator
(3) Logical operator
(4) Assignment operator
(5) Increment and decrement operator
(6) Conditional operator
(7) Bitwise operator
(8) Special operator.

Arithmetic operator: The Java arithmetic operators are:

+	: Addition
-	: Subtraction
•	: Multiplication
/	: Division
%	: Remainder

Relational Operator:

<	: Is less then
<=	: Is less then or equals to
➢	: Is greater then
>=	: Is grater then or equals to
==	: Is equals to
!=	: Is not equal to

Logical Operators:

&&	: Logical AND
||	: Logical OR

Introduction to Java : Operators in Java

!	: Logical NOT

Assignment Operator:

+=	: Plus and assign to
-=	: Minus and assign to
*=	: Multiply and assign to.
/=	: Divide and assign to.
%=	: Mod and assign to.
=	: Simple assign to.

Increment and decrement operator:

++	: Increment by One {Pre/Post)
--	: Decrement by one (pre/post)

Conditional Operator: Conditional operator is also known as ternary operator. The conditional operator is :

Exp1 ? exp2 : exp3

Bitwise Operator: Bit wise operator manipulates the data at Bit level. These operators are used for tasting the bits. The bit wise operators are:

&	: Bitwise AND
!	: Bitwise OR
^	: Bitwise exclusive OR
~	: One's Complement.
<<	: Shift left.
>>	: Shift Right.
>>>	: Shift right with zero fill

Example:

Introduction to Java : Operators in Java

Bitwise operator works on bits and performs bit-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
```

<<	**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

Special Operator - Instanceof operator

The **instanceof** is an operator that returns true if the object on the left hand side is an instance of the class given in the right hand side.

This operator allows us to determine whether the object belongs to the particular class or not.

Ex: class comp { }

```
comp a = new comp( );
if  ( a  instanceof  comp)
      System.out.println("a is an instanceof comp");

if  (a  instanceof  Object)
      System.out.println("a is an instanceof Object too");
```

Dot operator in Java

The dot(.) operator is used to access the instance members (using objects) or class members of a class.

length Vs length()

int [] a = new int[3];

array.length :

length is a final variable applicable for arrays. With the help of a length variable, size of the array can be obtained.

String a=”rnsit”;

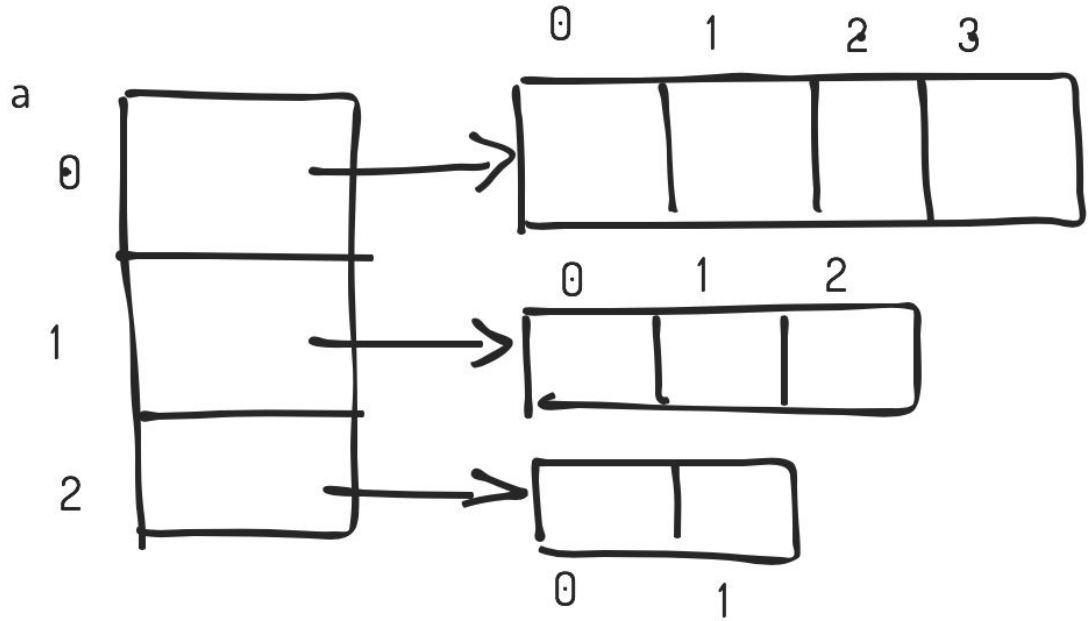
a.length() :

length() method is a method which is applicable for string objects. length() method returns the number of characters present in the string.

Introduction to Java : Skewed Array

A standard 2-d array stores equal amount of information in each row.

A skewed array is an array of arrays, where in each row will store different number of informations within them.



Introduction to Java : Skewed Array

Ex:

```
int [][] a  =  new int[3][];

a[0] = new int[4];
a[1] = new int[3];
a[2] = new int[2];

for(int i=0;i<a.length;i++)
        System.out.println(a[i].length);

Scanner sc = new Scanner(System.in);
for(int i=0;i<a.length;i++)
        for(int j=0;j<a[i].length;j++)
                a[i][j] = sc.nextInt();
```

Introduction to Java : Overriding in Java

In an object-oriented programming language,

Overriding is a feature that allows a derived class to provide a specific implementation of a method that is inherited by its superclasses.

When a method in a subclass has the same name, same formal parameter type, and same return type(or sub-type) as a method in its super-class,

then the method in the subclass is said to *override* the method in the super-class.

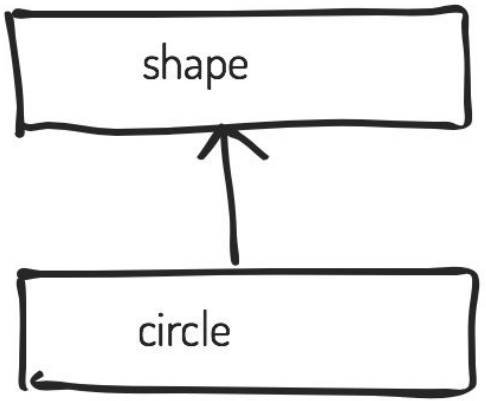
Introduction to Java : Overriding in Java

Ex:

```
class shape  {
        public void area()
        {  System.out.println("Nothing to calculate"); }
}
class circle extends shape  {
        public void area( )   //overridden function
        {  System.out.println("Area of a circle");  }
}
```

Signature of method area( ) in both shape and circle classes are same.

Method area( ) in circle is considered as **overridden** function.



Introduction to Java : Overriding in Java

In the first step of inheritance, member functions of base class can be inherited and in-turn used in the derived class.

In the second step, member functions of base class that are inherited can be restructured to solve a different task, pertaining to the needs of the derived class.

In order to do this, the member functions which is inherited from base class, will be redefined with the same signature in the scope of derived class.

This task is termed as **function overriding**.

The task of member function in base class is termed as **generalized** task
and the member function overridden in the derived class is termed as **specialized** task.

Depending on the logical necessity in derived class, the functions that are inherited might be or might not be overridden.

Introduction to Java : Overriding in Java

Two types of polymorphism exists in OOP

Compile time polymorphism (function overloading)

Runtime polymorphism (function overriding)

Compile time polymorphism means, compiler can decide which member function to call during compilation time itself.

Ex: comp a = new comp( ); a.accept(); // an example of CTP

Class comp is not inheriting any base class properties and a is not **generic reference/base class** reference, hence compiler knows which member function to call.

Runtime polymorphism means, compiler cannot decide, which member function to call until runtime, in other words.

The decision of a member function to call will be staggered until execution time.

Introduction to Java : Overriding in Java

Ex:
```
class base{
        public void disp( ) { System.out.println("in base"); }
}
class drd  extends base {
        public void disp( ) { System.out.println("in drd"); }
}

drd a = new drd( );       a.disp( );
```

drd is inheriting properties from class base, and 'a' is not a base class reference.
Hence, a.disp( ) accounts for CTP.

```
base a = new base( );
```

'a' is a base class reference, and 'a' can point to an object of type base or 'a' can point to an object of type drd.

a.disp( ); accounts for RTP.

a = new drd( ); //approved by compiler

'a' is a reference of type base and can point to an object of type drd.

a.disp( ); accounts for RTP.

Steps required to experience RTP

1) Inheritance 2) Function overriding 3) base class reference

4) overridden function and base class function must be called, using base class reference.

In other words Generalization and Specialization must be invoked

.

Introduction to Java : Overriding in Java

Use of Method overriding : is to achieve **specialized** functions and to also can achieve **Runtime polymorphism.**

If function overriding is done in class hierarchy, then 2 types of functions can be used, one is generalized property and other one is specialized property.

OOP language provides the facility of using one base class reference to access these multiple functionalities in the inheritance hierarchy, termed as **base-class/generic reference**.

Generic reference means it can point to an object of itself or the object of a class derived from it.

Only in inheritance hierarchy these generic references exist.

Main usage of generic references is **“one interface multiple invocation”.**

Introduction to Java : Overriding in Java

```
Ex:     class circle extends shape  {
                    public void area()   //overridden function
                    {  System.out.println("Area of a circle");  }

                    public  void display( )
                    {  System.out.println("in display function");  }
                }

          shape  p = new circle( );
          p.display( );  // generates CTE
```

Using base class reference only the properties passed on to derived class can be accessed, since display( ) is a member function of class circle, it cannot be invoked using ‘p’.

Introduction to Java : Overriding in Java

Rules for overriding a method

1. final methods of base cannot be overridden in derived class.

2. Static methods of base cannot be overridden, if a static method of the same signature is defined in derived class, the super class static method is hidden, termed as **method hiding**.

	Superclass Instance Method	Superclass Static Method
Subclass Instance Method	Overrides	Generates a compile-time error
Subclass Static Method	Generates a compile-time error	Hides

3. Private methods of a base class cannot be overridden

4. Signature of overridden method in derived class must match the signature of the method in base class.

5. The method present in base class can be called from the overridden method using the 'super.method-name( )' keyword.

Ex:
```
class shape  {
        public void area()
        {  System.out.println("Nothing to calculate"); }        }

    class circle extends shape  {
        public  void area()   //overridden function
        {  super.area();   System.out.println("Area of a circle");  }
```

Introduction to Java : Reading values from a file in Java

Ex:

```
import java.io.File;
Scanner sc=null;
try {
  sc = new Scanner(new File("/Users/hemanth/eclipse-workspace/first/src/first/No"));
   }
   catch(Exception e)
   {}

   int n=sc.nextInt();
   for(int i=0;i<n;i++)
         System.out.println(sc.nextInt());
```

File: No

4 1 2 3 4

Introduction to Java : Wrapper classes in java

A **Wrapper class** is a **class** which contains the primitive data types (int, char, short, byte, etc). In other words, **wrapper classes** provide a way to use primitive data types (int, char, short, byte, etc) as objects. These **wrapper classes** come under **java.util** package.

Integer is a wrapper class to store int type of value in java
Float is a wrapper class to store float type of value in java

Ex: **Integer a = 10;** // a is an object of built-in wrapper class Integer and its value is 10.
Float b = 20; // object b is of type built-in wrapper class Float and its value is 20.

There are several members functions associated with each wrapper classes which can be used for logical purpose in the program.

All primitive types as a wrapper class in java to store primitive values in class type.

Introduction to Java : Command Line Arguments in Java

Ex: //Run menu - Run Configurations

```
for(String i : args)
        System.out.println(i);
```

Values passed from CLA will be stored from 0th index itself.

Ex: consider CLA passed 12 12.2 R RNSIT

```
int i = Integer.parseInt(args[0]);
float j = Float.parseFloat(args[1]);
char a = args[2].charAt(0);
String b = args[3];

System.out.println(i);     System.out.println(j);
System.out.println(a);  System.out.println(b);
```

Module 3

Classes, Inheritance,Exception Handling:
Classes: Classes fundamentals; Declaring objects; Constructors, this keyword, garbage collection.
Inheritance: inheritance basics, using super, creating multi level hierarchy, method overriding.
Exception handling: Exception handling in Java.

Shortcut key to type **System.out.println**

Type sysout ctrl+space

To add **getters** and **setters** function for each data members

Create the required class

Place the cursor inside class & right click

Choose option “Source”

Class in Java

"A class is a template for an object, and encapsulates the fields and methods for the object. Class methods provide access to manipulate fields.

" fields" of an object are often called instance variables."

Ex:

```
class Rectangle  {
        private int length;   //Data Member or instance Variables or fields
        private int width;

        public Rectangle()  { } // ZPC

public void getdata(int x,int y)   //Methods or Instance level functions or member functions
        { length=x;width=y;  }

        public int rectArea()    //Method returning a value
        {  return(length*width);  }
    }
```

Class in Java

```
class RectangleArea       {
   public static void main(String args[])  {
        Rectangle rect1=null;
        Rect1 = new Rectangle();
 // object creation.  rect1 is a reference type variable which can hold a null value;
// rect1 is a referential entity which refers to an instance.

    rect1.getdata(10, 20);   //invoking methods using object with dot(.)
    int area1  =  rect1.rectArea();                System.out.println("Area1="+area1);
  }  //End of main
}//End of class
```

After defining a class, it can be used to create objects by instantiating the class.

Each object acquires memory to hold information for its instance variables **(i.e. its state)**.

Class in Java : **Creating instance of a class/Declaring objects**

```
Rectangle rect1=new Rectangle();
                    OR
Rectangle rect1;       // rect1 will be automatically initialized to null value
rect1 = new Rectangle();  // rect1 will be pointing to a memory of type Rectangle
```

Above two statements declares an object rect1 which is of type Rectangle class using **new** operator, this operator dynamically allocates memory for an object and returns a reference to it.

In java all class objects must acquire memory dynamically.

Java's primitive types are not considered as objects, rather they are considered as variables.

```
Rectangle rect2 = rect1;
```

will make both references (rect1 & rect2) to point to the same instance.

There will be no separate instance which will be pointed by rect1 and rect2 (Shallow copy)

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.

Constructors have no explicit return type.

Typically, constructors are used to provide initial values to the instance variables defined by the class, or to perform any other startup procedures required to create a fully formed object.

Java automatically provides a default constructor that initializes all member variables to zero if primitive numerical type. If field is a reference type then it will be automatically initialized to null value (char types to a whitespace).

However, once a constructor is defined, the default constructor is no longer used.

The Constructors

```
class comp
{
        private int r;                  private char k;
        private comp j;

        public void disp()
        { System.out.println(r+" "+j+" "+k); }
}
public class First {
        public static void main(String[] args) {
                comp a = new comp(); a.disp();
        }
}
```

this keyword

this keyword is used to refer to the invoking object.

If there is ambiguity between the instance variable and formal-parameter, this keyword resolves this ambiguity.

Ex:
```
class drd  {
        private int i;
        public void access(int i) {
                i = i;   //LHS 'i' and RHS 'i' are formal parameters
                // this.i = i;  Remedy
                System.out.println(this.i); }
}

drd a = new drd( );      a.access(10);          Output: 0
```

Program to create a Singly linked list in java

Garbage Collection

In Java deallocation of memory of objects is done automatically by JVM.

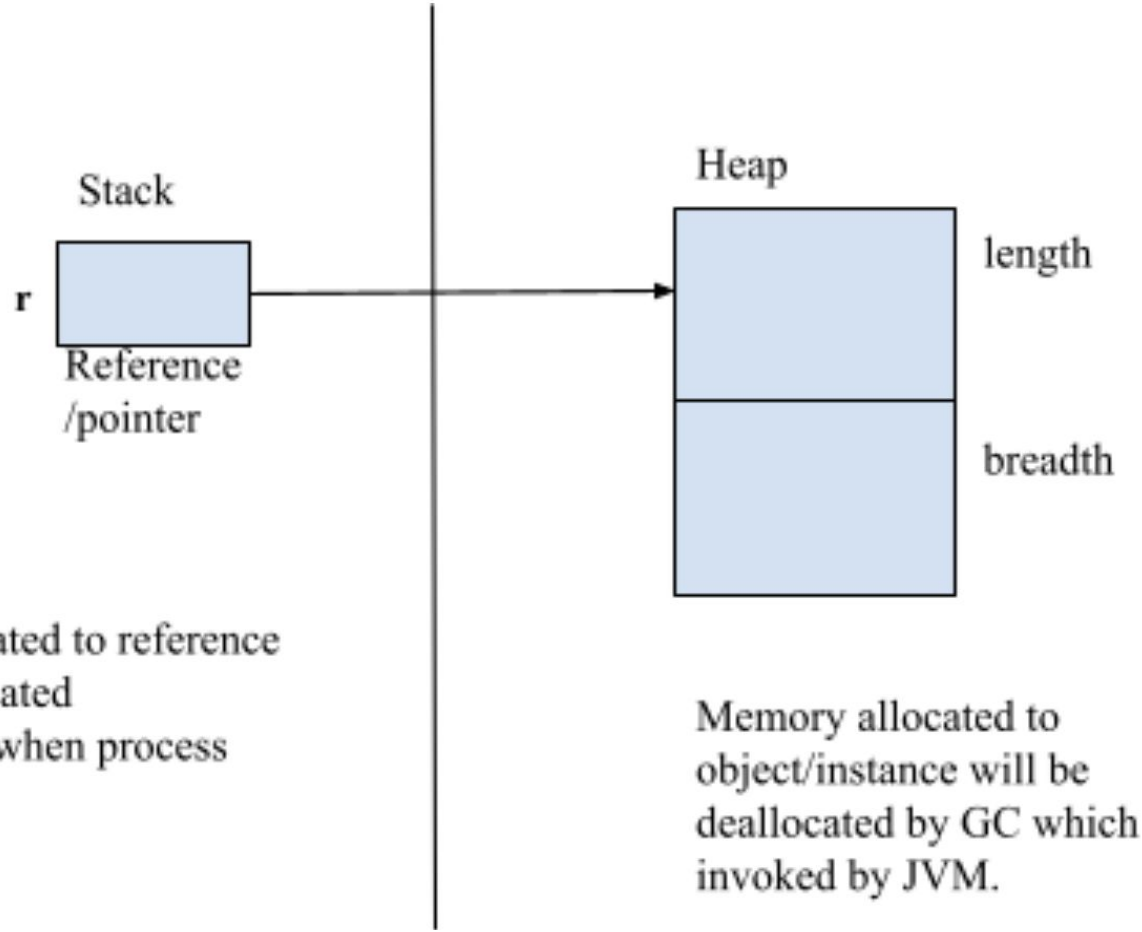
Java contains system programs such as GC, JVM etc.,

JVM helps java source program to get executed on native OS

GC which is also a system program and part of JVM will be invoked by JVM when a process comes to an end, or when it is necessary to reclaim resources such as main memory that are allocated to objects.

```
Rectangle r = new Rectangle( );        r=null;
```

Garbage Collection



Garbage Collection

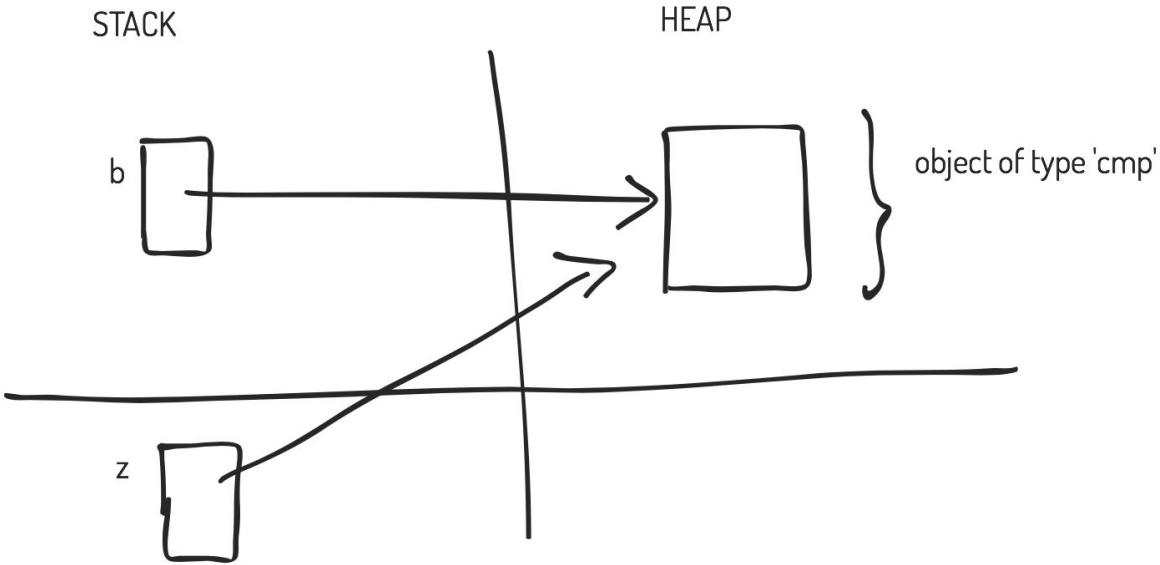
When there is no reference to an object, then that object is assumed to be no longer needed and the memory allocated to the object can be deallocated.

This technique is called Garbage Collection, which is accomplished automatically by the JVM.

Ex:
```
class cmp  {
        public   void   add(cmp z)  {
       // z is an another reference to b & 'this' is an another reference to a invoking object
            cmp t = new cmp();
        }  }

cmp b = new cmp( );    cmp a = new cmp( );      a.add(b);
```

as soon as control is out of add, 'z' reference will be de-allocated as they are local references. Instances pointed by z still remain becz they are being pointed by b.

Garbage Collection



Consider t, whose **reference (not instance)** will be deleted automatically once the control is out of add, the **object's memory** (dynamically acquired memory) pointed by t will be reclaimed automatically when GC system program is invoked by JVM.

Garbage Collection

There is no fixed time when GC will be called by JVM, it is left to the discretion of JVM to invoke GC.

If a process is executed continuously, without concluding and there is a lot of memory acquisition activity done by the program, then more amount of main memory will be used.

GC will be called oftenly by JVM to reclaim the required memory, whenever needed.

Ex: cmp a = new cmp( ); cmp b = new cmp( );

When control is about to **exit out of main** local references of main such as 'a' & 'b' will be deleted. As soon as the local references are deleted, the memory they are pointing to will be reclaimed by GC because it is the end of the process.

Garbage Collection

If there is an abrupt exit out of a process

```
import java.lang.System;
…….
System.exit(0);  // 0 means no errors and control is exiting out of current process
```

Then also JVM takes suitable measures to call GC before control leaves the process to reclaim main memory that has been acquired previously.

In the above scenario instance memory must be re-claimed even if they are being pointed by references. If not done then those locations will account for GARBAGE COLLECTION.

Native OS will not be responsible to deallocate those acquired memory locations, it will be the responsibility of JVM.

Garbage Collection

Garbage Collection program cannot be called explicitly.

A request can be placed to JVM for garbage collection by calling **System.gc()** method.
gc( ) is a static method of the System class (as it is called by class name).

GC Increases memory efficiency and nullifies the chances for memory leak.

finalize() method (similar to destructor)

Sometimes an object needs to perform some specific cleanup task before it is deallocated, such as releasing any resources held or closing an open connection.

Garbage Collection

Releasing resources word will not be considered for releasing memory allocations in java

Because dynamically acquired memory locations will be reclaimed automatically in java by GC.

Releasing resources word is more applicable for those resources, which cannot be reclaimed automatically by GC.

Ex: closing the file handler or closing the connection established to DBMS.

To handle such situations, **finalize()** method is used.

Signature of finalize() method which is part of super class Object.

```
protected void finalize()
{
    //finalize-code
 }
```

Garbage Collection

Necessity of explicit invocation of GC, can be for the following situation.

Ex: Scanner instance created to read information from keyboard must be closed using sc.close( ). Which can be done in finalize( ) method even if it is a static data member.

finalize() method is called by the garbage collection thread (if defined for a class) before deallocating objects.

It's the last step for any object to perform cleanup activity.

JVM -> GC -> finalize

Garbage Collection

```
Ex:  class temp{
            public void finalize()
            {
                  System.out.println("in finalize method");
            }
      }
class Test {
      public static void main(String[] args)    {
                  temp r = new temp();
                  r=null;
            // if r is not set to null gc() will not be called, if gc() is not called then
            //finalize() will not be called.
                  System.gc();  // just a request of jvm
      }     }
```

Garbage Collection

Ex: To release the standard input device attached to scanner

```
import java.util.Scanner;
class node
{  public int info;      public node next; }

class list {
    private node first; // first is just a reference variable
    public static Scanner sc;

    public list() { first=null; }

    public void i_f( )  {   }

    public void disp( )  {   }
```

Garbage Collection

```
    protected void finalize()
    {
        System.out.println("In finalize method");
        sc.close(); // valid statement for finalize.
    }   }
public class test {
    public static void main(String[] args) {
        list.sc = new Scanner(System.in);

        list l = new list();    l.i_f();  l.i_f();  l.i_f();  l.i_f();      l.disp();

        l=null;
        System.gc();
        }           }
```

Chapter 7 is important from a Java language perspective, but not part of syllabus.

Topics to be considered and understood

- Using Objects as parameters
- A Closer Look at Argument Passing
- Returning Objects
- Introducing Access Control
- Understanding static
- Exploring the String Class
- Using Command-Line Arguments

Inheritance (code reusability)

One of the most important features of Object Oriented Programming.

Inheritance means to inherit **fields & methods** that are already present in a different class.

Classes which share fields & methods to other classes are termed as **base/super/parent class**.
Class which inherits fields & methods are termed as **derived/sub/child class**.

Inheritance serves the purpose of **code reusability**.

To inherit a class properties in java the keyword used is **extends**.

Inheritance (code reusability)

Access Modifier	Within base class	Within derived class
default	Yes	Yes
private	Yes	No
public	Yes	Yes
protected	Yes	Yes

If “Yes” member is accessible in the scope
If “No” member is not accessible in the scope

Inheritance (code reusability)

```
class base  {
        int defi;
        static int sdefi;
        void default_disp()
                { System.out.println("in default MF "+defi); }
        static void default_disp_static()
                { System.out.println("in Static default MF "+sdefi); }

        private int pri;
        private static int spri;
        private void private_disp()
                { System.out.println("in private MF "+pri); }
        private static void private_disp_static()
                { System.out.println("in Static private MF "+spri); }
```

Inheritance (code reusability)

```
protected int proi;
protected static int sproi;
protected void protected_disp()
        { System.out.println("in protected MF "+proi); }
protected static void protected_disp_static()
        { System.out.println("in Static protected MF "+sproi); }


public int pubi;
public static int spubi;
public void public_disp()
        { System.out.println("in protected MF "+pubi); }
public static void public_disp_static()
        { System.out.println("in Static public MF "+spubi); }
```

Inheritance (code reusability)

```
        public void base_private_access()
        {
                pri=89;  private_disp();
                spri=89;  private_disp_static();
                System.out.println(this.getClass());
        }
} // end of class base
```

Inheritance (code reusability)

```
class drd extends base {
        public void access( ) {
            defi = 90;        default_disp();
            sdefi = 90;       default_disp_static();

            base_private_access();

            proi=87;     protected_disp();
            sproi=87;    protected_disp_static();
        }
    }
```

Inheritance (code reusability)

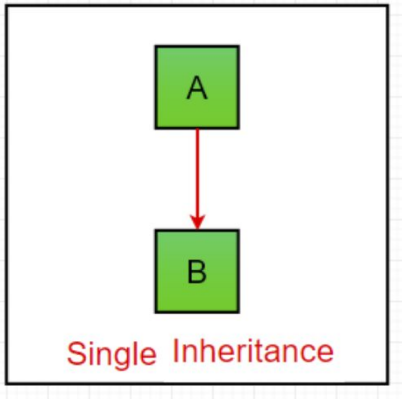
```
public class First {
        public static void main(String[] args) {
                drd d = new drd();
                d.access();

                d.pubi=86;      d.public_disp();
                d.spubi=86;     d.public_disp_static();
        }
}
```

Types of Inheritance

1. Single/Simple Inheritance
2. Multilevel Inheritance
3. Multiple Inheritance

Java doesn't support Multiple inheritance but we can achieve this through the concept of **interface**.

1. Single/Simple Inheritance



Single/Simple Inheritance

When a subclass is derived from its parent class then this mechanism is known as simple inheritance.

In case of simple inheritance there is only a sub class and its parent class.

It is also called single inheritance or one level inheritance.
Ex:

Object Superclass, Representing Single Inheritance

The Object class, in the java.lang package, is the **supreme implicit base class** in java.

Each and every class in java, whether built-in or user-defined inherits the properties of supreme base class Object.

Several methods are available in Object class, which can be called by derived classes without overriding them or by overriding them (final methods cannot be overridden).

Some of the methods in Object class are as follows.

```
public final Class getClass()
public int hashCode()
public boolean equals(Object obj)
public String toString()
public final void notify()
```

Object Superclass

“final” member functions of Object class can be inherited and used in derived class.

Non-final member functions of Object class can be inherited and overridden in derived class.

Overridden functions provides the facility to experience Specialized functions in derived classes, in class hierarchy.

Functions defined in Object class scope provides Generalized task.

Functions that are inherited and if extended further in derived class provides Specialized task.

It is not compulsory to override the non-final member function that are inherited from base class.

Object Superclass

```
class comp
{
    private int r,i;

    public comp(int r, int i)
    { this.r = r;  this.i = i;  }

    public String toString( ) // overridden function
    {
        System.out.println("In generalized toString( ) \n"+super.toString() );
        System.out.println("In Overridden Specailized toString() ");
        String a = r +"+i"+i;
        return a;
    }
```

Object Superclass

```
        public boolean equals(Object obj) //overridden function
        {
            // a is referred by this reference
            // b is referred by base class reference
            System.out.println("In generalized equals( ) ");
            System.out.println(super.equals(obj));
            System.out.println("In Overridden Specialized equals( ) ");

            comp t = (comp)obj;

            if (this.r == t.r && this.i == t.i)
                return true;
            return false;
        }
    }
```

Object Superclass

```
public class First {
    public static void main(String[] args) {
        comp a = new comp(1,2);      comp b = new comp(1,2);

        System.out.println("Calling equals( ) ");
        System.out.println("******* ********* ");
        if (a.equals(b))
            System.out.println("a is equal to b");
        else
            System.out.println("a is not euqal to b");

        System.out.println("\n\nCalling toString() ");
        System.out.println("******* ********** ");
        System.out.println(a);
    }        }
```

An example of Single inheritance

```
class Bicycle {   //base class
    public int gear, speed;
    public Bicycle(int gear, int speed)
    {       this.gear = gear;      this.speed = speed;    }
    public void applyBrake(int decrement)
    {     speed -= decrement;     }
    public void speedUp(int increment)
    {    speed += increment;     }
    @Override
    public String toString()
    {
        String a = "No of gears are "+ gear +"\n";
        a = a + "speed of bicycle is "+ speed;
        return a;
    }     }
```

An example of Single inheritance

```
class MountainBike extends Bicycle     {
        public int seatHeight;
        public MountainBike(int gear, int speed,  int startHeight)   {
                // invoking base-class(Bicycle) parameterized constructor
                super(gear, speed);
                seatHeight = startHeight;       }
        public void setHeight(int newValue)
        {    seatHeight = newValue;      }
        @Override
        public String toString()
        {
                String a =  super.toString()+ "\n"+"seat height is ";
                a = a + seatHeight;
                return a;
        }      }
```

An example of Single inheritance

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

                MountainBike mb = new MountainBike(3, 100, 25);
                System.out.println("Mountain bike\n"+mb);



                Object bi = new Bicycle(1,2);
    //base class reference,  referencing an instance of derived class type
                System.out.println("\n\nBicycle\n"+bi);
        }
}
```

"super" keyword

A subclass can call a constructor defined by its superclass by use of the following form of **super**:

super(*arg-list*);

arg-list specifies any arguments needed by the constructor in the superclass.

super( ) (base class constructor call) must always be the first statement executed inside a subclass' constructor.

A Second Use for super

G.F: super.member

Here, *member* can be either a member function or a datamember.

This second form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass (overridden functions).

This type of invocation can be coded in any part of the function.

“super” keyword

```
class base{  protected int i;  }

class drd extends base  {
        private int i;
        public void access() {
                super.i=10;
                i=20;
                System.out.println("i in base " + super.i);
                System.out.println("i in drd " + i);
        }
}
public class First {
        public static void main(String args[]) {
                drd  d = new drd();          d.access();
        } }
```

A Superclass Reference can refer to a Subclass Object

A reference variable of type superclass can be assigned to an instance of subclass.

By making a base class reference, refer to instance of derived class,
Members that are inherited from base class to derived class can be accessed by a base class reference. Members of the derived class cannot be accessed by base class reference.

Most important use of base class reference is "one interface, multiple access".

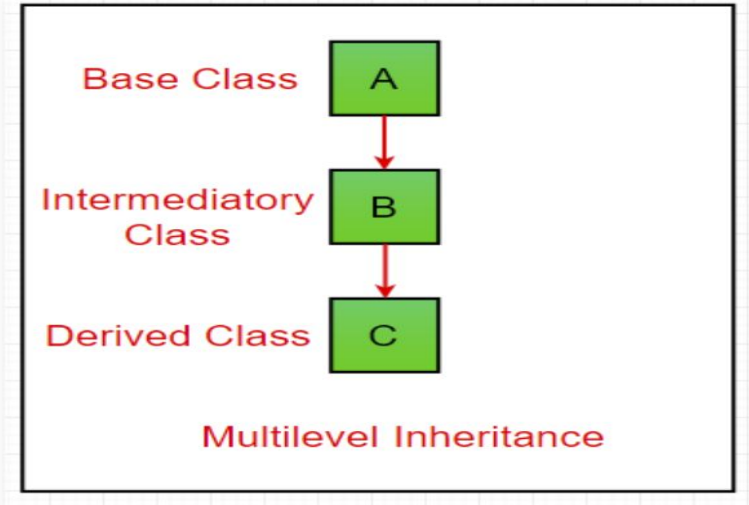
By using a single base class reference, members that are passed to derived classes can be accessed at any level of inheritance.

(Considering, overridden functions, since they are inherited and overridden in derived class, they can also be accessed by base class references)

"One interface, multiple access" is possible only when **mutually exclusive** classes are linked together with inheritance hierarchy.

Multilevel Inheritance

When a class is inheriting from a derived class then this mechanism is known as **"multilevel inheritance".**



Ex: B is inheriting properties from class A and B is also base class for derived class C

There is no restriction imposed on multilevel inheritance.

Multilevel Inheritance

```
class person    {
    private String name;
    public person(String s)
    {    setName(s); }

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

    public String getName()
    { return name; }

    @Override
    public String toString()
    {   return "Name = " + name;    }   }
```

Multilevel Inheritance

```
class Employee extends person     {
        private int empid;
        public Employee(String sname, int id)
        {    super(sname);       setEmpid(id);    }

        public void setEmpid(int id)   { empid = id; }


        public int getEmpid()      { return empid; }

        @Override
        public String toString()  {
          String a = super.toString()+" ";
          a = a + "Empid = " + empid;
          return a;
        }
}
```

Multilevel Inheritance

```
class HourlyEmployee extends Employee    {
    private double hourlyRate;          private int hoursWorked;

    public HourlyEmployee(String sname, int id, double hr, int hw)  {
      super(sname,id);          hourlyRate = hr;     hoursWorked = hw;       }

    public double GetGrosspay()
    {   return (hourlyRate * hoursWorked);  }
    @Override
    public String toString()  {
            String a = super.toString()+" ";
            a = a + " Hourly Rate = " + hourlyRate;
            a = a + " Hours Worked = " + hoursWorked;
            a = a + "Gross pay = " + GetGrosspay();
     return a;                }    }
```

Multilevel Inheritance

```
class First
{
    public static void main(String[] args)
    {
        HourlyEmployee emp = new HourlyEmployee("AB",1,15,1800);
        emp.GetGrosspay();
        System.out.println(emp);
    }
}
```

Constructors in inheritance hierarchy

Constructor in the inheritance hierarchy will be called in the order of derivation.

That is first constructor of base class will be called, followed by the constructor in the derived class. **Constructors are executed in the order of hierarchy.**

Ex:
```
class A {
    public A()  {   System.out.println("Inside A's constructor."+this.getClass( ));   }
}

class B extends A    {
    public B()  { System.out.println("Inside B's constructor."+this.getClass( ));   }
}

class C extends B   {
    public C()  {    System.out.println("Inside C's constructor."+this.getClass( )); }
}
```

Constructors in inheritance hierarchy

```
class CallingCons
{
      public static void main(String args[])
      {
            C c = new C();
      }

}
```

Output: Inside A's constructor.
Inside B's constructor.
Inside C's constructor.

Method with different signatures acting as overloaded instance methods

```
Ex:class A {
      int i, j;
       A(int a, int b)  {  i = a;    j = b; }
        void show()  {  System.out.println("i and j: " + i + " " + j);  }
      }

      class B extends A  {
         int k;
         B(int a, int b, int c)  {    super(a, b);  k = c; }
       // overload show()
        void show(String msg)   {  System.out.println(msg + k);    }
      }

            B subOb = new B(1, 2, 3);
            subOb.show("This is k: "); // this calls show() in B
            subOb.show(); // this calls show() in A
```

Dynamic method dispatch (DMD)

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than at compile time.

Main requirements for DMD are

1. Inheritance hierarchy (at least one parent child relationship).
2. Overridden function in derived class.
3. Base class reference, pointing to derived class instance.
4. Calling overridden function by base class reference.

A superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time.

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**.

Dynamic method dispatch (DMD)

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.

Generalization and Specialization

Function in base class exhibit generalized property.

Overridden functions in derived classes exhibit specialized property.

Ex: toString( ) method in Object base class and overridden toString( ) in user-defined classes.

Generalization and Specialization

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.

Overridden methods are another way that Java implements the "one interface, multiple methods" aspect of polymorphism.

Part of the key to successfully applying polymorphism is understanding that the superclasses and subclasses form a hierarchy which moves from lesser to greater specialization. Used correctly, the superclass provides all elements that a subclass can use directly. It also defines those methods that the derived class must implement on its own. This allows the subclass the flexibility to define its own methods, yet still enforces a consistent interface.Thus, by combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses.

Difference between Method overloading and overriding

Method Overloading
Occurs at compile time.
Different types and number of parameters must be passed to different function
Return type can be different for different functions.
Scope of the functions must be same

Method Overriding
Occurs at execution time.
Parameters must be the same in type and number.
Return type cannot be different, it has to be the same.
Scopes of the overridden functions will be in different classes; scopes that share hierarchical relationships between them.

Abstract class
Abstract class is a collection of one or more undefined member functions.

Abstract classes are used to provide some common functionality across a set of related classes.
Word “functionality” does not signify generalized property, but signifies the compulsion on derived classes to provide the functionality.

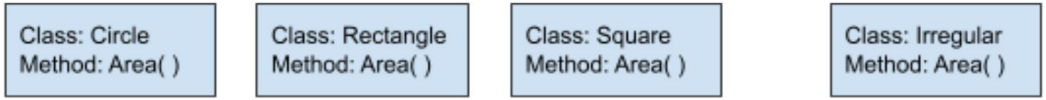
Definition (of undefined methods) must be done compulsorily in derived class.

Some base classes will be unable to provide a generalized functionality for the hierarchy, these methods are termed as **"abstract methods"**

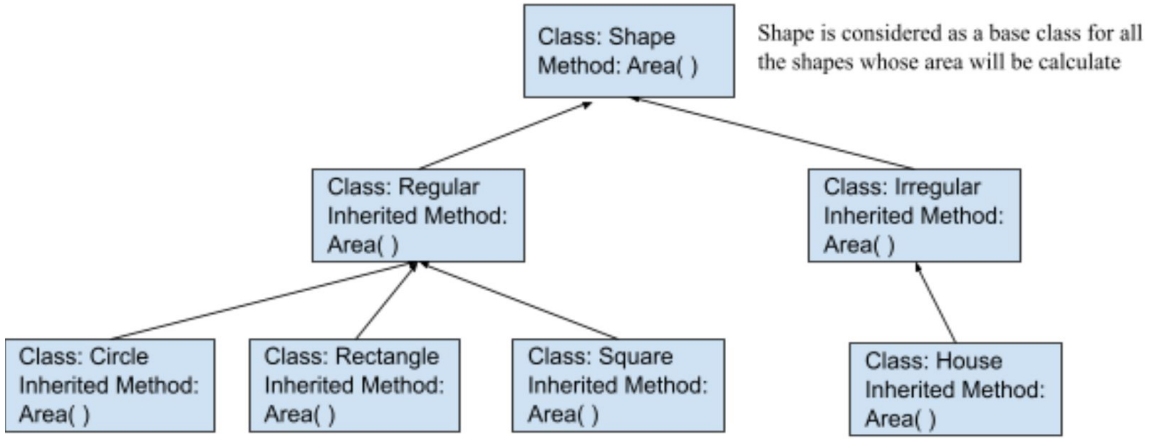
Class which contains at least one abstract method is termed as **"abstract class"**.

(Abstract means nonrealistic)

Abstract class : Another scenario, where abstract class is required



Any modification needed on all these classes (like to add method to find volume( )), requires changes in each class, since they are all different classes



Now, if we want all the shapes to calculate volume of it, then only one modification is enough to base class Shape.
Adding abstract method volume() to class Shape, will pass the properties to all subclasses.

ONE OF THE USAGES OF INHERITANCE AND ABSTRACT CLASSES.

Why abstract class ???

Consider an application program to process **employee** and **student type** information.

Why abstract class ???

Further, it can be observed that 2 classes have some fields in common, like name, gender and address.

Consider, methods such as accept( ) and display( ) to be present in both classes.

These common information can be accumulated in a base class, as shown below.

Ex:
```
class person {
        private String name,gender
        private String addr;

        public void  accept( );
        public void display( );  }
```

The member functions can be defined in class person, but the object of type student and employee has significance rather than the generic object type person.

Hence, class person will be converted as abstract and the method definition will be delegated to the derived class, which are going to inherit the properties of class person.

Abstract class

An abstract class is defined in java using the keyword “abstract”.

If a single member function in a class is defined as abstract, then the class must be compulsorily defined as abstract class.

Considering, the previous example, methods accept( ) and display( ) will be defined as abstract as follows.

Ex:

```
abstract class person
{
        private String name, gender;
        private String addr;

        public abstract void  accept( );
        public abstract void display( );
}
```

Abstract class

The class person can be used as a base class for student and employee class.

The derived class student and employee has to **compulsorily** override the accept() and display() member functions that are inherited and not defined in abstract class person.

Ex:

```
abstract class person  {
        protected String name;

        public abstract void accept( );

        public String toString() {
            String a = name;
            return a;
        }
}
```

Abstract class

```
class input
{  public static Scanner sc = new Scanner(System.in);  }

class student extends person {
        String usn;

        public void accept()
        { name=input.sc.next();    usn=input.sc.next(); }

        public String toString()
        {
                String a = super.toString();
                a = a + " " + usn ;
                return a;
        }
}
```

Abstract class

```
class employee extends person {
        String eid;

        public void accept()
        { name=input.sc.next();    eid=input.sc.next(); }

        public String toString()
        {
                String a = super.toString();
                a = a + " " + eid ;
                return a;
        }
}
```

Abstract class

```
class First
{
    public static void main(String [] args)
    {
        student s = new student();
        s.accept();
        System.out.println(s);

        employee e = new employee();
        e.accept();
        System.out.println(e);
    }
}
```

Abstract base class reference

```
person p = new student();    p.accept();
System.out.println(p);

p = new employee();              p.accept();
System.out.println(p);
```

If there is a member defined in class student or employee which is not inherited from abstract base class, then those member cannot be invoked by, abstract base class reference p.

```
p.name = "abc";  works because it is inherited from
p.usn = "123" ; //CTE
```

Assume class student contains a member function access( ) of its own, then

```
p.access( );  //CTE
```

Abstract base class with constructors

If there is instance level fields in abstract class, **it can be** initialized with the constructors of abstract class.

(NOTE: an instance of abstract base is not created here, but an instance of abstract base is created in derived class)

After creating an instance of type derived class, the abstract base class constructor is called on the instance of it, which is present in derived class.

Ex:
```
abstract class person  {
        protected String name;

        public person(String n)  {
            System.out.println("in person constr "+this.getClass());
            name=n;  }
        public abstract void accept( );
```

Abstract base class with constructors

```
        public String toString() {
                String a = name;
                return a;
        } }
        class student extends person {
                String usn;
                public student(String n, String u)   {
                        super(n);
                        System.out.println("in student constr");
                        usn=u;     }

        public void accept() { }
```

Abstract base class with constructors

```
        public String toString()  {
                String a = super.toString();
                a = a + " " + usn ;
                return a;
        }  }
        class First   {
                public static void main(String [] args)  {
                        person p = new student("a","1");
                        System.out.println(p);
                }      }
```

Output:

```
in person constr  class first.student
in student constr
a 1
```

Properties of abstract class

An object of abstract class cannot be instantiated, if done it raises CTE.
Ex: person p = new person(); ///CTE

Reference of an abstract class type can always be created.

Multiple abstract classes/Multiple base classes cannot be inherited by a single derived class.
Ex: class drd extends base, extends base1 // CTE

Any derived class of an abstract class **must define all of the abstract methods** in the superclass, or be itself declared abstract (must be another abstract class)

Abstract class can have abstract and non-abstract methods.

Abstract class can have final methods, which must be defined in abstract class itself.

Abstract class can have constructors and static methods also.

Properties of abstract class

Abstract class can have data members and static DM also.

Data members of abstract classes will acquire memory only in derived classes.

Member function of abstract class will be invoked by the object of derived class.

Abstract class

```
/*Abstract class can have abstract and non-abstract methods,
  Abstract class cannot be instantiated.
  Abstract class can have final methods.
  Abstract class can have constructors and static methods also. */
abstract class base {
        public  int i;  // data member of abstract class   field
        abstract void abm(); // declared method not defined
        public void nonabm()   // defined member function
        {  System.out.println("in non abstract method"); }
        final public void finonabm() // final method
        { System.out.println("final non-abstract method"); }
        public static void snonabm() // static member function
        { System.out.println("Static method of abstract class"); }
        public base() // constructor
        { System.out.println("ZPC abstract base class");              i=90;        }
}
```

Abstract class

```
class der extends base {
        public der()
        { System.out.println("ZPC der"); }

        public void abm()
       {  System.out.println("Defined function - Abstract type");  }
}
class test {
 public static void main(String[] args)  {
                base d = new der();
                d.i = 900; // base class data member
                d.abm(); // invoking defined function - abstract function
                d.snonabm(); // invoking static non abstract member function
                d.finonabm(); // invoking final non abstract member function
                d.nonabm(); // invoking non-final non abstract member function
         }          }
```

“final” keyword in Java

“final” keyword in java has 3 uses

1. can be used to create a constant
2. can be used on method to avoid overriding
3. can be used to prevent inheritance

Constant creation

final members must be initialized at the point of declaration.

final fields can be initialized inside the constructor. If more than one constructor is there for the class, then it must be initialized in all of them, otherwise compile time error will be thrown.

```
Ex:  class test {
            public static void main(String args[]) {
                 final int i = 10;
                      i=i+1; //CTE
      } }
```

“final” keyword in Java

final to Prevent Overriding

If a method in the base class is defined using the final keyword, then these members cannot be overridden in derived classes.

Ex:
```
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 an attempt is made, a compile-time error will result.

Overridden method call decisions are usually made during run-time, since final methods are not overridden, these method invocation is decided during run-time itself.

“final” keyword in Java

final to Prevent Inheritance

In order to prevent a class from inheriting, it can be preceded by the keyword final.

Declaring a class as final implicitly declares all of its methods as final.

A class in java cannot be created as both abstract and final.

Ex:

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

Exception in java

Exception word is synonym for runtime errors.

Exceptions are events that occur during the execution of programs that stop execution.

Ex: divide by zero, array access out of bound, File not found, NullPointer Exception etc

```
class test {
        public static void main(String[] args) {
                int i=10;
                i=i/0;
                System.out.println("After the statement");
 } }
```

Exception generated is as follows

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

Since, exceptions are runtime errors, information regarding the same will be encapsulated within an object in java.

Exception in java

Exception information contains..

Reason why exception has been generated (/ by zero)
package name, class name, method name
line number "hello.test.main(test.java:15)"
etc., where runtime error was generated

will be encapsulated or bundled in an object by JVM and will be handled either by JVM or programmers.

If JVM catches an exception the only remedy is that process comes to an halt.

If a program catches the exception by using built-in keywords in java, a remedy can be provided for the same in the program and program execution can continue (Robust).

(Robust : "Java programs are Robust"

Software Engineering definition for Robust

Resistant or impervious to failure regardless of user input or unexpected conditions.)

Exception in java

```
class cmp {
        public void access()
        {  }
}

class test {
    public static void main(String[] args)   {
        cmp i=null;
        i.access();
        System.out.println("After the statement");
     }
 }
```

```
Exception in thread "main" java.lang.NullPointerException
        at hello.test.main(test.java:20)
```

Exception in java

Exceptions that are generated usually indicate different types of error conditions.

Some common built-in exception classes available in Java are.

FileNotFoundException *to handle file not found condition*
IOException *to handle input output error condition*
SocketTimeoutException
NullPointerException *Dereferencing a null reference*
ArrayIndexOutOfBoundsException *Trying to read outside the bounds of an array*
ArithmeticException *Dividing an integer value by zero*
etc.,

Five keywords are used in java to handle exceptions in program itself

try, catch, throw, throws and **finally**.

Exception in java

G.F:

```
try
 {
      Statements that may generate exception or run-time error
      Since, each exception has a name associated with it, catch block will be coded to
      catch that type of exception.
 }
 catch(Type_of_exception e)
 {
    Remedy for exception, will be provided here.
    "e" is a reference variable which is ready to point to the exception object that will be
     generated by JVM
 }
```

Exception is a built-in class in java which is the supreme base class for all types of exception classes that are available in java.

Exception in java

```
class cmp {
      public void access() {  }
}
class test {
     public static void main(String[] args)   {
       cmp i=null; // this will not generate exception
       try {
             i.access();  // this will generate exception; i.e NullPointerException
          }
        catch(Exception e)  //Generic catch block
       {
              System.out.println(e);
/* the above statement prints the name of the exception generated, which is not a remedy
    for the generated exception.  */
       } }   }
```