Module-1
Chapter-1
An introduction to programming Language

What is a program?
A computer program is a collection of instructions that can be executed by a computer to perform a specific task. A computer program is usually written by a computer programmer in a programming language.

Programming language

A **programming language** is a formal language comprising a set of instructions that produce various kinds of output. Programming languages are used in computer programming to implement algorithms. Most programming languages consist of instructions for computers.

Types of programming languages
There are two **types of programming languages**, which can be categorized into the following ways
1. Low level language
a) Machine language (1GL)
b) Assembly language (2GL)
2. High level language
a) Procedural-Oriented language (3GL)
b) Problem-Oriented language (4GL)
c) Natural language (5GL)

1. Low level language:
- This language is the most understandable language used by the computer toperform its operations. It can be further categorized as

a) Machine Language (1GL): Machine language consists of strings of binary numbers (i.e. 0s and 1s) whichthe processor directly understands. Machine language has the merits of very fast execution speed and efficient use of primary memory.

Merits
- It is directly understood by the processor. So it has faster execution time since the programs written in this language need not to be translated. It doesn't need larger memory.

Demerits
- It is a very difficult task writing program in machine level language since all the instructions are to be represented by 0s and 1s.
- Use of this machine level language makes programming time consuming.
- It is difficult to find an error and to debug them in machine level language.
- It can be used by experts only.

b) Assembly Language: Assembly language is also known as low-level language because to design a program, the programmer requires detailed knowledge of hardware specification. This language uses mnemonics code such as ('ADD' for addition,'STORE' for keeping data etc') in place of 0s and 1s. The program is converted into machine code by assembler. The resulting program is referred to as an object code.

Merits

- It makes programming easier than machine level language since it uses mnemonics code for programming. Eg: ADD for addition, SUB for subtraction, DIV for division, etc.
- It makes the programming process faster.
- Error can be identified much easily as compared to machine level language.
- It is easier to debug than machine language.

Demerits

- Programs written in this language is not directly understood by the computer. So a translator is required which will translate from assembly language to object code.
- It is the hardware dependent language so programmers are forced to think in terms of computer's architecture rather than the problem being solved.
- Since it is the machine dependent language, programs written in this language are very less or not portable.
- Programmers must know its mnemonics codes to perform any task.

2. High level language:

- Instructions written in this language are closely resembles to human languages or English like words. It uses mathematical notations to perform the task. The high-level language is easier to learn. It requires less time to write and is easier to maintain the errors. The high-level language is converted into machine language by one of the two different languages translator, **interpreter or compiler.**
- High level language can be further categorized as

a) Procedural-Oriented language (3GL): Procedural Programming is a methodology for modelling the problem being solved by determining the steps and the order of those steps that must be followed in order to reach a desired outcome or specific program state. These languages are designed to express the logic and the procedure of a problem to be solved. It includes languages such as Pascal, COBOL, C, FORTAN, etc.

Merits

- Because of its flexibility, procedural language is able to solve a variety of problems. Programmer does not need to think in term of computer architecture which makes them focused on the problem.
- Programs written in this language are portable.

Demerits

- It is easier but needs higher processor and larger memory. It needs to be translated.So its execution time is more.

b) Problem-Oriented language (4GL): It allows the users to specify what the output should be, without describing all the details of how the data should be manipulated to produce the result. This is one step ahead from 3GL. These are result oriented and include database query language, eg: Visual Basic, C#, PHP, etc.

The objectives of 4GL

- are to increase the speed of developing programs.
- Minimize user's effort to obtain information from computer.
- Reduce errors while writing programs.

Merits

- Programmer need not to think about the procedure of the program. So, programming is much easier.

Demerits

- It is easier but needs higher processor and larger memory.
- It needs to be translated. So its execution time is more.

c) Natural language (5GL): Natural language are still in developing stage where we could write statements that would look like normal sentences.

Merits

- It is easy to program. Since, the program uses normal sentences, they are easy to understand. The programs designed using 5GL will have artificial intelligence (AI).
- The programs would be much more interactive and interesting.

Demerits

- It is slower than previous generation language as it should be completely translated into binary code which is a tedious task. Highly advanced and expensive electronic devices are required to run programs developed in 5GL. Therefore, it is an expensive approach.

Compiler and Interpreter

1. Compiler

- It is a translator which takes input from high Level Language and produces an output in machine level or assembly language.
- Compiler is more intelligent than an assembler. It checks all kinds of limits, ranges, errors etc.But it's program run time is more and occupies a larger part of memory. Its speed is slow because a compiler goes through the entire program and then translates the entire program into machine codes.

- In the context of java programming, we can say, compiler is a program that translates code written in a high-level programming language (like JavaScript or Java) into low-level code (like Assembly) directly executable by the computer or another program such as a virtual machine.
- For example, the Java compiler converts Java code to JavaBytecode executable by the JVM (Java Virtual Machine). Other examples are V8, the JavaScript engine from Google which converts JavaScript code to machine code or GCC which can convert code written in programming languages like C, C++, Objective-C, Go among others to native machine code.

2. Interpreter

- An interpreter is a program which translates a programming language into a comprehensible language.
- It translates only one statement of the program at a time.
- Interpreters, more often are smaller than compilers.

Difference between Compiler and Interpreter

- Interpreters and compilers are very similar in structure. The main difference is that an interpreter directly executes the instructions in the source programming language while a compiler translates those instructions into efficient machine code.
- An interpreter will typically generate an efficient intermediate representation and immediately evaluate it. Depending on the interpreter, the intermediate representation can be an *AST*, an *annotated AST* or a machine-independent low-level representation such as the *three-address code*.

Compiler	Interpreter
Compiler scans the whole program in one go.	Translates program one statement at a time.
As it scans the code in one go, the errors (if any) are shown at the end together.	Considering it scans code one line at a time, errors are shown line by line.
Main advantage of compilers is it's execution time.	Due to interpreters being slow in executing the object code, it is preferred less.
It converts the the instructions into systematic code.	It doesn't convert the instructions instead it directly works on source language.
Example: C, C++, C# etc.	Example: Python, Ruby, Perl, SNOBOL, MATLAB etc.

Introduction to Object-oriented programming language

Object oriented thinking and basics need for oops paradigm:

- **Object-oriented programming** (**OOP**) is a programming **paradigm** based upon objects (having both data and methods) that aims to incorporate the advantages of modularity and reusability. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs.
- Since the 1980s the word 'object' has appeared in relation to programming languages, with almost all languages developed since 1990 having object-oriented features. Some languages have even had object-oriented features. It is widely accepted that object-oriented programming is the most important and powerful way of creating software.
- Due to the various problems faced by the procedural languages, the concept of OOP was introduced. Therefore, it has a number of advantages over the procedural programming languages.

Let us discuss some of the advantages of the Object-oriented programming:

1. Real World Entities:

- In OOP real world entities are used. Classes and objects can be made of the things that are real and exist in the world. Example of real-world entities are pen, chair, student, hospital, etc. these examples can be considered as classes and their attributes will be considered as their object.
- Let's discuss other example in order to better understand the classes and objects. Suppose, we take a student as class, then objects will be student name, roll number, section, address, mobile number, email, etc.

2. Code Reusability:

- Code reusability is one of the characteristics of object-oriented programming. The feature that explains this point is inheritance. In inheritance, the class and subclasses or parent and child classes can be derived and its data member and member functions can be used as such. This feature saves time and the user do not need to code again and again, if similar features or functionality is required. So, a lot of time can be saved.

3. Easy Management

- Code management becomes very easy in the object-oriented programming. As all the code is divided into a number of elements it becomes easy to manage. For instance, the whole program can be termed as a class and even if it contains a number of functions are written or coded in it, their objects can be made. This makes it easy to manage the code, as later you can initialize the object variable and make the function call.

4. Maintenance:

- Maintenance of code also becomes easy in object-oriented programming. Because of easy management of the code maintenance also becomes easy. If the code is to be used by another programmer, still it will not create any ambiguity or correct guiding of coding is used.

5. Abstraction:

- In abstraction, only the useful data is visible to the user and not the things which are not required. Abstraction is also called as data hiding. For example, when we login into an email id, we can only type the login Id and password and view it. We do not know or cannot see how the data is being verified and how it is being stored. The user is always not aware about where their id and password are going. In which database it is stored? What are the criteria for checking it? etc. Hence, it is very beneficial practice to use this feature as it provides data security.

6. Polymorphism:

- It is another feature of the object-oriented programming. Polymorphism simply means that a function has many forms. The '+' operator, for example, it can be used to add two integers whereas used to join two strings. Similarly, even in the programming one function can be written, this function can be used in different forms depending on its arguments. It is called function overloading.

The advantages of OOP are mentioned below:

- OOP provides a clear modular structure for programs.
- It is good for defining abstract data types.
- Implementation details are hidden from other modules and other modules has a clearly defined interface.
- It is easy to maintain and modify existing code as new objects can be created with small differences to existing ones.
- objects, methods, instance, message passing, inheritance are some important properties provided by these particular languages
- encapsulation, polymorphism, abstraction are also counts in these fundamentals of programming language.

- It implements real life scenario.
- In OOP, programmer not only defines data types but also deals with operations applied for data structures.
- Each object can be easily developed and maintain in program.
- Data hiding and data abstraction increases reliability//y and provide the security.
- It helps to re-usability of code.
- Dynamic binding increases flexibility by permitting the addition of new classes.

Benefits of Oops:

- simulation and modeling.
- User interface design.
- Artificial intelligence and expert systems.
- Neural networks.
- Web designing.
- Component object model(COM).

Summary of OOPs Concepts:

The following are the fundamental OOPs concepts.

- Everything is an Object.
- Computation is performed by the objects communicating with each other, requesting that other objects perform actions. Objects communicate by sending and receiving messages. A message is a request for action bundled with arguments may be necessary to complete the task.
- Each object has its own memory, which consists of other objects.
- Every object is an instance of a class. A class simply represents a grouping of similar objects such as integers or lists.
- The class is the repository for behavior associated with an object. That is, all objects that are instances of the same class can perform the same actions.
- Classes are organized into a singly rooted tree structure, called the inheritance hierarchy. Memory and behavior associated with instances of a class are automatically available to any class associated with descendent in this tree structure.

Chapter-2
An introduction to Java

What is java?

- Java is a **programming language** and a **platform**. Java is a high level, robust, object-oriented and secure programming language.

- Java was developed by *Sun Microsystems* (which is now the subsidiary of Oracle) in the year 1995. *James Gosling* is known as the father of Java. Before Java, its name was *Oak*. Since Oak was already a registered company, so James Gosling and his team changed the Oak name to Java.
- Java is a high-level programming language originally developed by Sun Microsystems and released in 1995. Java runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX such as sun-Solaris,red-hat,ubuntu etc. Java is a popular third generation programming language,which can do any of the thousands of things that a computer software can do. With the features it offers,java has become the language of choice for internet and intranet applications. The kind of functionality the java offers, has contributed a lot towards the popularity of java.

History of Java Program

- Java was conceived by James Gosling in 1991. Initially it was called as Oak. But it was renamed as Java in 1995. The language evolved from a programming language known as the Oak. Oak was developed in the early 1990s by Sun Microsystems as platform that would support entertainment applications such as video games. Because it implements Object Oriented Principles it is called as Object Oriented Programming Language.
- Originally java started as an elite project (codenamed Green) to find a way of allowing different devices such as TV-top boxes and controllers to use a common language. This language for electronic devices was originally named Oak but failed to find a niche despite its potential. But with the explosion of world, java rose to charts of popularity as it could cater to nearly all platforms. In the following lines, the history of java can be found out
    - 1991 – *James gosling* develops Oak (later named Java) language for programming intelligent consumer electronic devices.
    - 1995 – Java formally announced. Incorporated into Netscape web browser.
    - 1997 –JDK 1.1 launched. Java Servlet API released.
    - 1998−Sun introduces Community source “open licensing”.Sun produces a JDK 1.2 for Linux.
    - 2001–JDK1.3 was released,J2EE(Java enterprise edition),J2SE.J2ME appear.
    - 2002–Java support for web services officially released via the java™ web service developer pack.
    - 2005–Java technology –enabled services devices are available.
    - 2006–The NetbeanIDE 5.0 is released.

Applications:

There are 2 basic types of Java applications:

1. **Standalone:**These run as a normal program on the computer. They may be a simple console application or a windowed application. These programs have the same capabilities of any program on the system. For example, they may read and write files. Just as for other languages, it is easily to write a Java console program than a windowed program.
2. **Applets:**These run inside a web browser. They must be windowed and have limited power. They run in a restricted JVM (Java Virtual Machine) called the sandbox from which file I/O and printing are impossible. (There are ways for applets to be given more power.)

<u>Difference between c++ and Java</u>

- C++ uses only compiler, whereas Java uses compiler and interpreter both.
- C++ supports both operator overloading & method overloading whereas Java only supports method overloading.

- C++ supports manual object management with the help of new and delete keywords whereas Java has built-in automatic garbage collection.
- C++ supports structures whereas Java doesn't support structures.
- C++ supports unions while Java doesn't support unions.

Oops concepts in java:

Object oriented program concepts are used to remove some of the existing problems in procedural approach.

- It ties the data as a critical element.
- It doesn't allow to flow freely around the system.
- It protects and provides security use to decompose of a problem into number of entities called objects.
- The data can be accessed only by the functions.
- The earlier languages are C,C++ are not having the capability to simplify the complexity of problem due to involvement of object concept the simplification is to build the tangible project.

Java supports the following object-oriented programming concepts.

- object
- class
- data encapsulation
- data abstraction
- inheritance
- polymorphism
- data binding
- message passing

Generally, main pillars of oops concepts are data encapsulation, dataabstraction, inheritance and polymorphism. Rest of the concepts supports to build the object-oriented programming.

1. Object: Object is nothing but a runtime entity which having member variables and Member functions.

- It can be represented with "new" keyword.
- In object the data associated with function hence it cannot access directly.
- It can access only through functions.
- It can be defined as a vector, person, place and account.

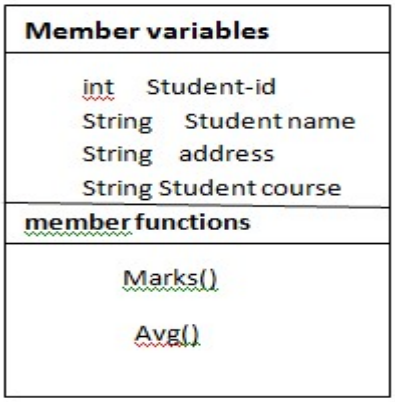
2.Classes: Collection of objects are called classes.It contains member variables and member functions.

- It can be represented with "class" keyword.
- Once we create a class, creates any number of objects (belongs) within the class.

Eg: fruit is a class, it may consider any number of fruit related objects like mango, banana, apple…….

```
Syntax:   class <classname>{                    {
                member variables;
member functions;
                }
```

Example:



3.Data encapsulation:

- It is a mechanism, to **hide the data properties** in a class called encapsulation or **wrapping-up of data** and functions in a given class is also called encapsulation.
- It is extracted from "capsule" word, how the capsule wraps up hidden parts and it act as a safe guard and provide the security.
- The data is not accessible from outside world. Only wrapped functions can access. It is core part of java.

4.Data abstraction: It is a mechanism, which will hide the implementation details in a given class. Hence it provides security and binding data and methods together in a single unit.It refers with "Abstract Data Type" (ADT)

- **Eg**: in mobile communication a user can know only usage of lift call and cancel call, don't know the implementation of background coding, networking and routing parts.
- **Eg:** A driver can drive the car by the functioning of steering, accelerator and gears. But don't know implementation mechanism of engine and circuits.

5.Inheritance: It is an ancient property which derives some common properties from base class to sub class is called inheritance

- It provides the re-usability of code.
- Data shares in the program as well as hide the data
- We can also add additional features to the existing class.

Syntax:

```
class <class name>
{
        instance variables;
        instance methods();
}
Class <subclass name> extends class <basecalss name>
{
Instance variables;
Instance methods();
}
```

6. Polymorphism: It is a Greek termPoly means "many", morphs mean "forms".

- It is an ability to take more than one form is called polymorphism
- It exhibits different behaviors in different instances. It depends upon type of data used in the operation.

- Polymorphism classified in two types

a) Compile time (static):

- It allocates memory at compile time in program execution.
- It is also called early binding polymorphism.
- Example: Method of overloading

b) Runtime (dynamic):

- It allocates memory at run time in program execution.
- It is also called late binding polymorphism.
- Example: Method of overriding

  Eg: Consider the operation of addition for 2 numbers it will generates the sum of integers.
  A=10, B=20 then, A+B=30

  If the operands/strings, the operation would produce third party of string by applying like concatenation operator called operator overloading.

  Eg: string 1+ string 2= string 3

7.Data binding:

- Binding is a method of calling code by itself.
- Data binding are of two types: static binding and dynamic binding.
- Static binding: It occurs during compile time. Private, final, static methods as well as variables involved in static binding.
- Dynamic binding: It occurs at run time. virtual methods are bonded during run time based upon runtime object.

8.Message passing:

- It is a form of communication used in parallel programming.
- The object-oriented programming communications are completed by the sending of messages like functions, signals and data packets.

JAVA BUZZ WORDS

The following is the list of Java Buzz words.

- Simple
- Secure
- Portable
- Object Oriented
- Robust
- Multithreaded
- Architecture – neutral
- Interpreted
- High Performance
- Distributed
- Dynamic

1. **Simple**: Java inherits the C/ C++ syntax and many object-oriented features of C++. This makes most programmers to learn java easy.
2. **Secure:**When a java compatible web browser is used, it is safe to down load the java applets. Java achieves this protection by confining a java program to the java execution environment and not allowing it access to other parts of the computer.
3. **Portable:**It supports WORA (Write Once Run Anywhere) properties of Java.The same java program will run, without change the operating system. For programs to be dynamically downloaded to all the various types of platforms connected to the internet, portable executable code is needed. The java compiler generates the portable code **(byte code)** which is executed by the java virtual machine (JVM) on the computer.

4. **Object Oriented:**Java was designed that everything is like an object. The object model is simple and simple and easy to extend.
5. **Robust:**Java programs behave uniformly across different platforms. Java program must execute reliably in a variety of systems. The ability to create robust programs was given a high priority in the design of java. Because java is a strictly typed language, it checks the code compile time and also at run time.To better understand how java is robust, consider two main reasons for program failure.
    - Memory management
    - Exception handling
6. **Multithreaded:**Java supports multithreaded programming, which allows us to write program that do many things simultaneouslyi.e execute multiple processes simultaneously.
7. **Architecture-Neutral:**Java designers provides a JVM which enables you to write a program once and execute it on different machines. It made the program longevity and portable. The goal of the java designers was "**write once run anywhere, anytime, forever."**
8. **Interpreted and High performance:**Java enables the creation of cross platform program by compiling into an intermediate code representation called java byte code. This code can be executed on any machine that implements JVM.
9. **Distributed:**As java handled TCP/IP protocols it is possible to write distributed programs for internet. Java also supports Remote Method Invocation (RMI). This feature enables a program to invoke methods across a network.
10. **Dynamic:**Java programs carry with them substantial amount of runtime type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe manner. This is crucial to the robustness of the applet environment, in which small fragments of byte code may be dynamically updated on a running system.

SIMPLE JAVA PROGRAM

- In java, a source file is officially called a compilation unit. It is a text file that contains one or more class definitions.
- By convention, the name of that class should match the name of the file that holds the program.
- Java is **case sensitive programming language.**
- **A simple Java program consists of:**

1. The **comments** are ignored by the compiler by using special symbols like

// - for single line comment

/* and end with */ - for multiline comment.

- The class keyword is used to declare a class and follows the class name.

2. class Example

- The entire class definition, including all of its members, will be between the opening curly brace ({) and closing curly brace (})

3. public static void main(String args[])

- All java applications starts execution at main method.
- The **public** keyword is an access specifier, which allows the programmer to control the visibility of class members.
- main() must be declared as public when it is called by code outside of its class when the program is started.
- The keyword **static** allows main()to be called without having instantiate a particular instantiate a particular instance of the class.
- This is necessary since main() is called by the JVM before any objects are made.

- The keyword **void** tells the compiler that main() does not return any value.
- In main(), there is only one parameter that receives String parameter.
- To display the output, **System** is a predefined class that provides access to the system and **out** is the output stream that is connected to the console.
- Output is actually accomplished by the built in function called **println()**

Write a Java program print simple "Hello" using standalone program concept,

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

A) Creating a Java source file

- Java source files must end in an **.java** extension. The root name must be the same as the name of the one public class in the source file. In the program above, the class is named Hello and thus, the file must be named **Hello.java**.
- Just as for other languages, any text editor can be used to save the text of the program into a text file.

B) Compiling a Java source file

- Sun's JDK includes a Java compiler named javac. To compile the above Java program one would type:

  javac Hello.java
- If successful, this creates a file named Hello.class If not successful, it prints out error messages like other compilers. The output of a Java compiler is not executable code. It is bytecode.
- Bytecode is a set of instructions designed by the Java run time system, which is called the Java Virtual Machine. The JVM is an interpreter for bytecode. This running JVM then executes the Java bytecode which is platform independent, provided that you have a JVM available for it to execute.

C) Running a Java program

- To run a standalone program, Sun's JDK provides a Java interpreter called java. To run the **.class** file created above, type:

  java Hello
- Note that **.class** is not specified. The output of running the above program is simply:**Hello**

What is JVM?

- A program written in high level language cannot be run on any machine directly. First, it needs to be translated into that particular machine language. The javac compiler does this thing, it takes java program (.java file containing source code) and translates it into machine code (referred as byte code or .class file).
- Java Virtual Machine (JVM) is a virtual machine that resides in the real machine (your computer) and the machine language for JVM is byte code. This makes it easier for compiler as it has to generate byte code for JVM rather than different machine code for each type of machine. JVM executes the byte code generated by compiler and produce output. JVM is the one that makes java platform independent.

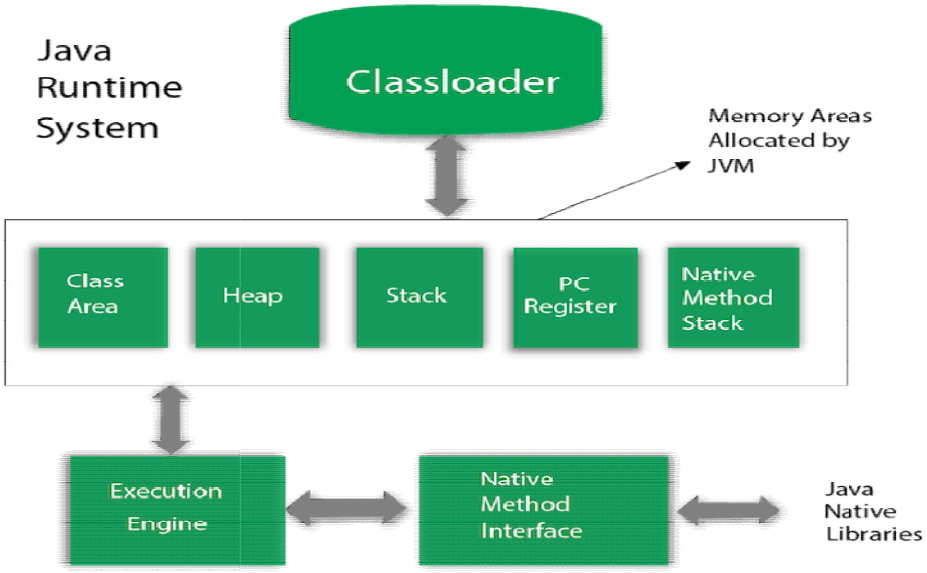
- Hence, primary function of JVM is to execute the byte code produced by compiler. Each operating system has different JVM, however the output they produce after execution of byte code is same across all operating systems. Which means that the byte code generated on Windows can be run on Mac OS and vice versa. That is why we call java as platform independent language.
- Java Virtual Machine converts byte code to the machine-specific code. JVM runs the program by using libraries and files given by Java Runtime Environment.It can execute the java program line by line hence it is also called as interpreter.
- JVM is easily customizable for example, we can allocate minimum and maximum memory to it.
- It is independent from hardware and the operating system. So, you can write a java program once and run anywhere.

Significance of JVM

- JVM provides a platform-independent way of executing Java source code.
- It has numerous libraries, tools, and frameworks.
- Once you run Java program, you can run on any platform and save lots of time.
- JVM comes with JIT(Just-in-Time) compiler that converts Java source code into low-level machine language. Hence, it runs more faster as a regular application.

Architecture of JVM

- Let's understand the internal architecture of JVM. It contains classloader, memory area, execution engine etc.



Let us discuss the working principle

- **Class Loader:** The class loader reads the .class file and save the byte code in the **method area**.
- **Method Area**: There is only one method area in a JVM which is shared among all the classes. This holds the class level information of each .class file.
- **Heap**: Heap is a part of JVM memory where objects are allocated. JVM creates a Class object for each .class file.
- **Stack**: Stack is a also a part of JVM memory but unlike Heap, it is used for storing temporary variables.

- **PC Registers**: This keeps the track of which instruction has been executed and which one is going to be executed. Since instructions are executed by threads, each thread has a separate PC register.
- **Native Method stack:** A native method can access the runtime data areas of the virtual machine.
- **Native Method interface**: It enables java code to call or be called by native applications. Native applications are programs that are specific to the hardware and OS of a system.
- **Garbage collection**: A class instance is explicitly created by the java code and after use it is automatically destroyed by garbage collection for memory management.

Difference between JDK, JRE, and JVM

JVM:

- JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.
- JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each OS is different from each other. However, Java is platform independent. There are three notions of the JVM: *specification*, *implementation*, and *instance*.
- The JVM performs the following main tasks:
    - Loads code
    - Verifies code
    - Executes code
    - Provides runtime environment

JRE:

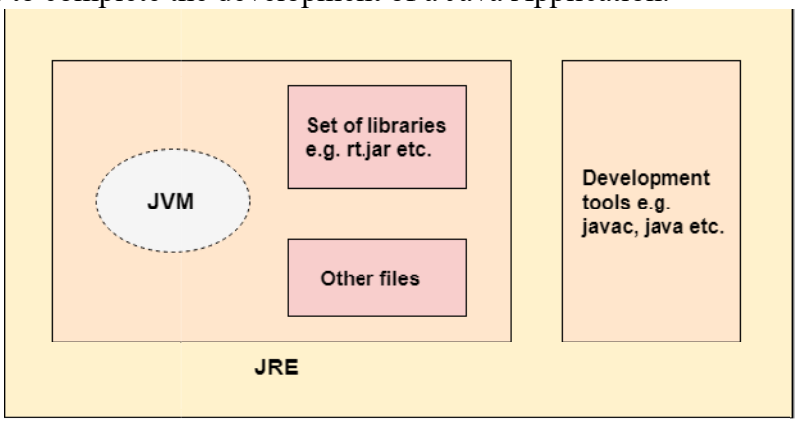
- JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.
- The implementation of JVM is also actively released by other companies besides Sun Micro Systems.

JRE

JDK:

- JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and applets. It physically exists. It contains JRE + development tools.

- JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:
    - Standard Edition Java Platform
    - Enterprise Edition Java Platform
    - Micro Edition Java Platform
- The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.



JDK

Chapter - 3

Java tokens

- In Java, a program is a collection of classes and methods, while methods are a collection of various expressions and statements. Tokens in Java are the small units of code which a Java compiler uses for constructing those statements and expressions. Java supports 5 types of tokens which are:
    - Keywords
    - Identifiers
    - Literals
    - Operators
    - Special Symbols

a) Keywords

- Keywords in Java are predefined or reserved words that have special meaning to the Java compiler. Each keyword is assigned a special task or function and cannot be changed by the user. We cannot use keywords as variables or identifiers as they are a part of Java syntax itself.
- A keyword should always be written in lowercase as Java is a case sensitive language.
- Java supports various keywords, some of them are listed below:

01. abstract	02. boolean	03. byte	04. break	05. class
06. case	07. catch	08. char	09. continue	10. default
11. do	12. double	13. else	14. extends	15. final
16. finally	17. float	18. for	19. if	20. implements
21. import	22. instanceof	23. int	24. interface	25. long
26. native	27. new	28. package	29. private	30. protected
31. public	32. return	33. short	34. static	35. super
36. switch	37. synchronized	38. this	39. throw	40. throws
41. transient	42. try	43. void	44. volatile	45. while

46. assert	47. const	48. enum	49. goto	50. strictfp

b) Identifier

- Java Identifiers are the user-defined names of variables, methods, classes, arrays, packages,and interfaces. Once we assign an identifier in the Java program, we can use it to refer the value associated with that identifier in later statements.
- There are some de facto standards which you must follow while naming the identifiers such as:
  1. Identifiers must begin with a letter, dollar sign or underscore.
  2. Apart from the first character, an identifier can have any combination of characters.
  3. Identifiers in Java are case sensitive.
  4. Java Identifiers can be of any length.
  5. Identifier name cannot contain white spaces.
  6. Any identifier name must not begin with a digit but can contain digits within.
  7. **keywords** can't be used as identifiers in Java.

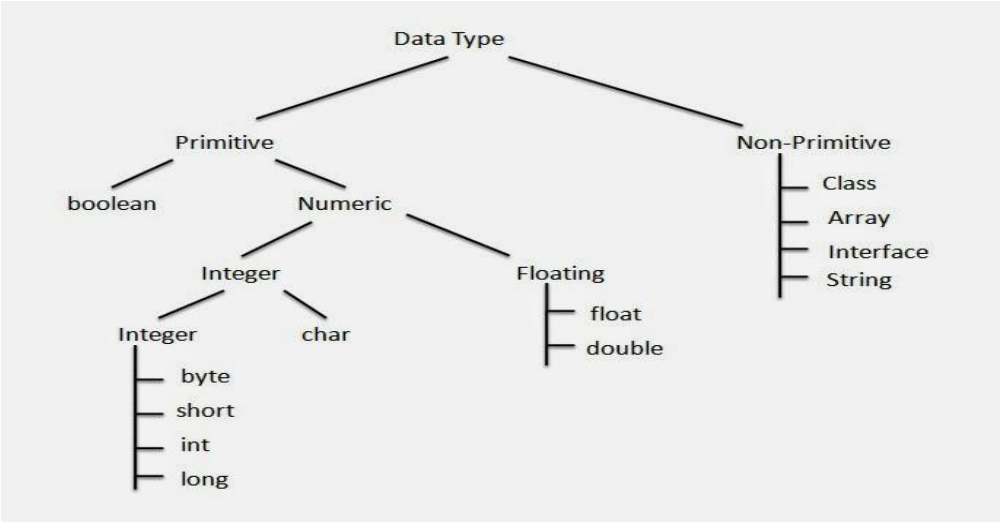
c) Literals

- Literals in Java are similar to normal variables but their values cannot be changed once assigned. In other words, literals are constant variables with fixed values. These are defined by users and can belong to any data type.
- Java supports five types of literals which are as follows:

i. Integer: e.g. 5,15,100 etc
ii. Floating Point: e.g. 5.5, 15.25, 105.538 etc
iii. Character: e.g. 'A', 'b', 'z', 'X' etc.
iv. String: e.g. "ABIT", "CSE", "Cuttack"
v. Boolean: e.g. TRUE and FALSE

Data types in java

- Data types specify the classification of data, it allocates the memory for variable it tells what kind of data values is stored in it.
- Each type of data (such as integer, character, hexadecimal, packed decimal, and so forth) is predefined as part of the programming language and all constants or variables defined for a given program must be described with one of the data types. These are two types
  1. Primitive data type
  2. Non primitive data type



Primitive Datatype:

i) boolean

- boolean data type represents only one bit of information either true or false . Values of type boolean are not converted implicitly or explicitly (with casts) to any other type. But the programmer can easily write conversion code.

  Example: // A Java program to demonstrate boolean data type

```
class booldatatype
{
   public static void main(String args[])
   {
boolean b = true;
      if (b == true)
System.out.println("welcome dear students");
   }
}
```

ii)byte

- The byte data type is an 8-bit signed two's complement integer.The byte data type is useful for saving memory in large arrays.
- Size: 8-bit (1 byte)
- Value: -128 to 127

  Example :

```
class JavaExample {
    public static void main(String[] args) {
    byte num;
    num = 113;
System.out.println(num);
  }
}
```

iii) short

- The short data type is a 16-bit signed two's complement integer. Similar to byte, use a short to save memory in large arrays, in situations where the memory savings actually matters.
- Size: 16 bit (2-bytes)
- Value: -32,768 to 32,767 (inclusive)

iv) int

- It is a 32-bit signed two's complement integer.
- Size: 32 bit
- Value: -231 to 231-1

Example:
```
class JavaExample {
    public static void main(String[] args) {
  short num;
 num = 150;
System.out.println(num);
    }
   }
```

v) long:

- The long data type is a 64-bit two's complement integer.
- Size: 64 bit
- Value: -263 to 263-1.

Example:
```
class Java Example {
    public static void main(String[] args) {
  long num = -12332252626L;
System.out.println(num);
    }
  }
```

vi) float:

- The float data type is a single-precision 32-bit IEEE 754 floating point. Use a float (instead of double) if you need to save memory in large arrays of floating point numbers.
- Size: 32 bits
- Suffix: F/f
- Example: 9.8f

Example:
```
class JavaExample{
    public static void main(String[] args) {
float num = 19.98f;
System.out.println(num);
    }
    }
```

vii) double:

- The double data type is a double-precision 64-bit IEEE 754 floating point. For decimal values, this data type is generally the default choice.

Example:
```
class JavaExample{
    public static void main(String[] args) {
  double num = -42937737.9d;
System.out.println(num);
    }
    }
```

viii) char

- The char data type is a single 16-bit Unicode character. A char is a single character.
- Value: '\u0000' (or 0) to '\uffff' 65535

Example: class JavaExample

```
{
public static void main(String[] args) {
char ch = 'Z';
System.out.println(ch);
}
}
```

Non-primitive Data types:

- These data types are derived from primitive data types. They are:

i. **Reference:** reference data types are the more sophisticated members of the data type family. They don't store the value, but store a reference to that value Java keeps the reference, also called address, to that value, not the value itself.These are reference variables.

ii. **Array:**An **array** is a single object that contains multiple values of the same type. There is nothing special about arrays of objects (non-primitives) given a type T, an array of references to T is defined.

Syntax:T[] array= new T[42];// declares an array of 42 elements

iii. **Class:**In order to create a new non-primitive or reference variable for this class, we have to create a new instance of the Product class. The new keyword is used to create an object

Example:

```
class product
{
//Body of the code
}
product P=new product(); //P is the object of class product
```

iv. **Interface:** An interface is like a dashboard or control panel for a class which contains abstract methods and final static variables. It has the buttons, but the function is elsewhere. We won't go into detail on implementing interfaces since the focus is on the interface as a non-primitive, or reference, data type.

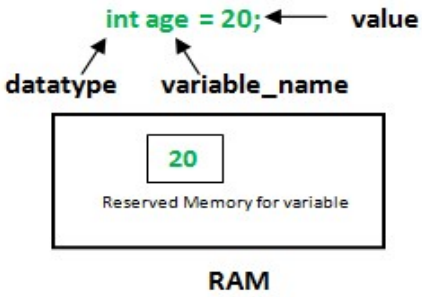
Example:

```
interface products
{
Public void getitemid();
Public void getprice();
}
```

Variables in Java

- A variable is the name given to a memory location. It is the basic unit of storage in a program.
- The value stored in a variable can be changed during program execution.
- A variable is only a name given to a memory location; all the operations done on the variable effects that memory location.
- In Java, all the variables must be declared before they can be used.

Declarations of variables



- **datatype**:Type of data that can be stored in this variable.
- **variable_name**: Name given to the variable.
- **value**: It is the initial value stored in the variable.

Examples:

- float simpleInterest; //Declaring float variable
- int time = 10, speed = 20; //Declaring and Initializing integer variable
- char var = 'h'; // Declaring and Initializing character variable

There are three types of variables in Java:
1. Local Variables
2. Instance Variables
3. Static Variables

1. Local Variables:

- A variable defined within a block or method or constructor is called local variable.
- These variables are created when the block in entered or the function is called and destroyed after exiting from the block or when the call returns from the function.
- The scope of these variables exists only within the block in which the variable is declared. i.e. we can access these variables only within that block.

Example:

```
public class StudentDetails
        {
        public void StudentAge()
          {   //local variable age
         int age = 0;
        age = age + 5;
     System.out.println("Student age is : " + age);
          }
         public static void main(String args[])
            {
          StudentDetails obj = new StudentDetails();
          obj.StudentAge();
               }
        }
```

2. Instance Variables:

- Instance variables are non-static variables and are declared in a class outside any method, constructor or block.

- As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.
- Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier then the default access specifier will be used.

Example:

```
import java.io.*;
class Marks
{
   //These variables are instance variables.
   //These variables are in a class and are not inside any function
   int engMarks;
   int mathsMarks;
   int phyMarks;
}
class MarksDemo
{
   public static void main(String args[])
   {   //first object
      Marks obj1 = new Marks();
      obj1.engMarks = 50;
      obj1.mathsMarks = 80;
      obj1.phyMarks = 90;
       //second object
      Marks obj2 = new Marks();
      obj2.engMarks = 80;
      obj2.mathsMarks = 60;
      obj2.phyMarks = 85;
       //displaying marks for first object
      System.out.println("Marks for first object:");
      System.out.println(obj1.engMarks);
      System.out.println(obj1.mathsMarks);
      System.out.println(obj1.phyMarks);

      //displaying marks for second object
      System.out.println("Marks for second object:");
      System.out.println(obj2.engMarks);
      System.out.println(obj2.mathsMarks);
      System.out.println(obj2.phyMarks);
   }
}
```

3. Static Variables:

- Static variables are also known as Class variables.
- These variables are declared similarly as instance variables, the difference is that static variables are declared using the static keyword within a class outside any method constructor or block.
- Unlike instance variables, we can only have one copy of a static variable per class irrespective of how many objects we create.

- Static variables are created at start of program execution and destroyed automatically when execution ends.
- To access static variables, we need not to create any object of that class, we can simply access the variable as: **class_name.variable_name;**

Example:

```
import java.io.*;
class Emp {
   // static variable salary
  public static double salary;
  public static String name = "Harsh";
}
 public class EmpDemo
{
    public static void main(String args[]) {
      //accessing static variable without object
Emp.salary = 1000;
System.out.println(Emp.name + "'s average salary:" + Emp.salary);
  }
   }
```

SCOPE AND LIFETIME OF VARIABLES

- Java allows the variables to be declared within any block. A block defines a begun with an opening curly brace and ended by a closing curly brace.
- A block defines a scope. Thus each time you start a new block, you are creating a new scope.
- A scope determines what objects are visible to other parts of the program.
- It also determines the lifetime of those objects.

There are two major scopes in java.

- **class scope:** Variable or methods defined within a class is called to be in class scope.
- **Method scope:** Variable or methods defined within a function is called to be in method scope.

NOTE:

- Variables declared within the scope are not visible to the code that is defined outside that scope. Scopes can be nested.
- The objects declared in the outer scope will be visible to the code within inner scope. But the objects declared within inner scope will not be visible outside it.

Example:

```
class Scope
    {
      public static void main(String args[])
         {
              int x=10;
              if(x==10)
                 {
                     int y=20;
System.out.println("x= "+x+"y= "+y);
```

```
                    }
System.out.println("x="+x);
                }
            }
```

Operators in java

Definition: Operator is a symbol which will operate on oneor more than one operand. Hence operator can be of two types:

1. Unary: Operator that works on a single operand.
2. Binary: Operator that works on two operands.

Example: A=10,B=20
C=A+B//Binary operator + adds A and B, result is 30
D=-B // Unary operator (-) negates value of B, result is -20

Types of operator:

1.Arithmetic Operators
2.Unary Operators
3.Assignment Operators
4.Relational Operators
5.Logical Operators
6.Ternary Operators
7.Bitwise Operators
8.Shift Operators

1.Arithmetic Operators: Java Arithmetic operators are used for simple math operations, they are: Addition (+), Subtraction (-), Multiplication (*), Division (/), Modulo (%)

Example

```
public class operators
{
public static void main(String[] args)
{
int a = 20, b = 10, c = 0, d = 20, e = 40, f = 30;
String x = "Thank", y = "You";
// + and - operator
System.out.println("a + b = "+(a + b));
System.out.println("a - b = "+(a - b));
// + operator if used with strings
// concatenates the given strings.
System.out.println("x + y = "+x + y);
// * and / operator
System.out.println("a * b = "+(a * b));
System.out.println("a / b = "+(a / b));
// modulo operator gives remainder on dividing first operand with second
System.out.println("a % b = "+(a % b));
/*if denominator is 0 in division then Arithmetic exception is thrown. Uncommenting below line
would throw an exception, will be discussed in Exception Handling concept*/
// System.out.println(a/c);
}
}
```

Output

```
a+b = 30
a-b = 10
x+y = ThankYou
a*b = 200
a/b = 2
a%b = 0
```

2.Unary Operators

Operator Name	Description
Unary Minus (-)	For decreasing the value
Unary Plus (+)	For converting the negative values into positive
Increment operator (++)	Used for increasing of the operand by 1
Post-Increment	The value is incremented first then the result is computed
Pre-Increment	The value is incremented later then the result is computed
Decrement Operator(--)	Used for decreasing the value of operand by 1
Post-Decrement	The value is decremented first then the result is computed
Pre-Decrement	The value is decremented later then the result is computed
Logical not Operator (!)	Used for inverting the values of boolean

Example: // Java program to illustrate unary operators

```
public class operators
{
public static void main(String[] args)
{
int a = 20, b = 10, c = 0, d = 20, e = 40, f = 30;
boolean condition = true;
// pre-increment operator
// a = a+1 and then c = a;
c = ++a;
System.out.println("Value of c (++a) = " + c);
// post increment operator
// c=b then b=b+1
c = b++;
System.out.println("Value of c (b++) = " + c);
// pre-decrement operator
// d=d-1 then c=d
c = --d;
System.out.println("Value of c (--d) = " + c);
// post-decrement operator
// c=e then e=e-1
c = --e;
System.out.println("Value of c (--e) = " + c);
// Logical not operator
```

```
System.out.println("Value of !condition =" + !condition);
}
}
```

3.Assignment Operators

Assignment operators are used to assigning values to the left operand. Its types are,

Operator	Description
+=	To add the right and left operator and then assigning the result to the left operator
-=	To subtract the two operands on left and right and then assign the value to the left operand
*=	To multiply the two operands on left and right and then assign the value to the left operand
/=	To divide the two operands on left and right and then assign the value to the left operand
^=	To raise the value of left operand to the power of right operator
%=	To apply modulo operator

Example: // Java program to illustrate assignment operators

```
public class operators
{
public static void main(String[] args)
{
int a = 20, b = 10, c, d, e = 10, f = 4, g = 9;
// simple assignment operator
c = b;
System.out.println("Value of c = " + c);
// This following statement would throw an exception as value of right operand must be initialised before an assignment, and the program would not compile.
// c = d;
// instead of below statements,shorthand assignment operators can be used to provide same functionality.
a = a + 1;
b = b - 1;
e = e * 2;
f = f / 2;
System.out.println("a,b,e,f = " + a + ", + b + "," + e + "," + f);
a = a - 1;
b = b + 1;
e = e / 2;
f = f * 2
// shorthand assignment operator
a += 1;
b -= 1;
e *= 2;
f /= 2;
System.out.println("a,b,e,f (using shorthand operators)= " +a + "," + b + "," + e + "," + f);
}
}
```

4.Relational Operators

Java Relational operators are used to check the equality and for comparison.

Operator Name	Description
== (equals to) ex. x==y	True if x equals y, otherwise false
!= (not equal to) ex. x!=y	True if x is not equal to y, otherwise false
< (less than) ex. x<y	True if x is less than y, otherwise false
> (greater than) ex.x > y	True if x is greater than y, otherwise false
>= (greater than or equal to) ex. x>=y	True if x is greater than or equal to y, otherwise false
<= (less than or equal to) ex. x<=y	True if x is less than or equal to y, otherwise false

Example

```
// Java program to illustrate relational operators
public class operators
{
public static void main(String[] args)
{
int a = 20, b = 10;
String x = "Thank", y = "Thank";
int ar[] = { 1, 2, 3 };
int br[] = { 1, 2, 3 };
boolean condition = true;
//various conditional operators
System.out.println("a == b :" + (a == b));
System.out.println("a < b :" + (a < b));
System.out.println("a <= b :" + (a <= b));
System.out.println("a > b :" + (a > b));
System.out.println("a >= b :" + (a >= b));
System.out.println("a != b :" + (a != b));
// Arrays cannot be compared with relational operators because objects store references not the value
System.out.println("x == y : " + (ar == br));
System.out.println("condition==true :" + (condition == true));
}
```

5.Logical Operators

Operator Name	Description
&& (Logical AND)	Returns the value if both the conditions are true otherwise returns zero.
|| (Logical OR)	Returns the value if even one condition is true

Example: // Java program to illustrate logical operators

```
public class operators
{
public static void main(String[] args)
{
String x = "java";
String y = "class@3";
Scanner s = new Scanner(System.in);
```

```
System.out.print("Enter username:");
String uuid = s.next();
System.out.print("Enter password:");
String upwd = s.next();
// Check if user-name and password match or not.
if ((uuid.equals(x) &&upwd.equals(y)) ||
(uuid.equals(y) &&upwd.equals(x))) {
System.out.println("Welcome user.");
}
else
{
System.out.println("Wrong uid or password");
}
}
}
```

6. Ternary Operators in Java

- Ternary java operators are a shorthand version the 'if else' statements.
- Syntax- **condition? if true : if false**

Example: // Java program to illustrate max of three numbers using ternary operator.

```
public class operators
{
public static void main(String[] args)
{
int a = 20, b = 10, c = 30, result;
//result holds max of three numbers
result = ((a > b) ? (a > c) ? a :
c : (b > c) ? b : c);
System.out.println("Max of three numbers = "+result);
}
}
```

7.Bitwise Operators

Bitwise java operators are used to perform operations on single bitwise values.

Operator Name	**Description**
Bitwise AND operator (&)	The & operator compares corresponding bits of two operands. If both bits are 1, it gives 1 else 0.
Bitwise OR operator (|)	The | operator compares corresponding bits of two operands. If either of the bits is 1, it gives 1 else 0.
Bitwise XOR operator (^)	The ^ operator compares corresponding bits of two operands. If corresponding bits are different, it gives 1 else 0.
Bitwise Complement operator (~)	The ~ operator inverts the bit pattern. It makes every 0 to 1 and every 1 to 0.

Example: // Java program to illustrate bitwise operators

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

```
{
int a = 0x0005;
int b = 0x0007;
// bitwise and Ex. 0101 & 0111=0101
System.out.println("a&b = " + (a & b));
// bitwise OR Ex. 0101 | 0111=0111
System.out.println("a|b = " + (a | b));
// bitwise xor Ex. 0101 ^ 0111=0010
System.out.println("a^b = " + (a ^ b));
// bitwise Complement Ex. ~0101=1010
System.out.println("~a = " + ~a);
// can also be combined with assignment operator to provide shorthand assignment
// a=a&b
a &= b;
System.out.println("a= " + a);
}
}
```

8.Shift operators:

Operator Name	Description
<< (left shift operator)	Shifts the value to the left which specified by the right operand.
>> (right shift operator)	Shifts the value by zeroes defined by left operand.
>>> (unsigned right shift operator)	It fills the void on left with zeroes which are set by the right operand.

Example: // Java program to illustrate shift operators

```
public class operators
{
public static void main(String[] args)
{
int a = 0x0005;
int b = -10;
// left shift operator
// 0000 0101<<2 =0001 0100(20)
// similar to 5*(2^2)
System.out.println("a<<2 = " + (a << 2));
// right shift operator
// 0000 0101 >> 2 =0000 0001(1)
// similar to 5/(2^2)
System.out.println("a>>2 = " + (a >> 2));
// unsigned right shift operator
System.out.println("b>>>2 = "+ (b >>> 2));
}
}
```

Expression:

An expression can be any combination of variables, literals, and operators. They also can be method calls, because methods can send back a value to the object or class that called the method.

Types of Expressions

There are three types of expressions in Java:

1. Those that produce a value, i.e. the result of (1 + 1)
2. Those that assign a variable, for example (v = 10)
3. Those that have no result but might have a "side effect" because an expression can include a wide range of elements such as method invocations or increment operators that modify the state (i.e. memory) of a program.

Examples of Expressions

1. Expressions that Produce a Value

- Expressions that produce a value use a wide range of Java arithmetic, comparison or conditional operators. For example, arithmetic operators include +, *, /, <, >, ++ and %. Some conditional operators are&&, ||, and the comparison operators are <, <= and >. See the Java specification for a complete list.
- These expressions produce a value:3/2, 5% 3, pi + (10 * 2) and so on.

2.Statements:

- Statements are roughly equivalent to sentences in natural languages. A statement forms a complete unit of execution. The following types of expressions can be made into a statement by terminating the expression with a semicolon (;):
  - Assignment expressions
  - Any use of ++ or --
  - Method calls
  - Object creation expressions
- These kinds of statements are called expression statements. Here are some examples of expression statements:
  - Value = 8933.234; //assignment statement
  - Value++; //increment statement
  - System.out.println(Value); //method call statement
  - Integer integerObject = new Integer(4); //object creation statement
- In addition to these kinds of expression statements, there are two other kinds of statements. A declaration statement declares a variable. You've seen many examples of declaration statements.
  - double Value = 8933.234; // declaration statement

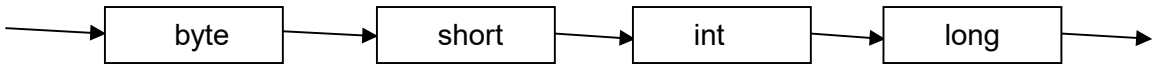
TYPE CONVERSION AND CASTING

Type conversion:

- It is the process of converting one data type into another data type.
- If the two types are compatible, then Java will perform the conversion automatically.

Java supports two kinds' type conversions

1. Implicit type conversion (wider conversion): It process the conversion of small data type Into big data type, used to increase the size of memory of variable and it cannot loss of data it is also called as wider conversion



Wider conversion

Example:

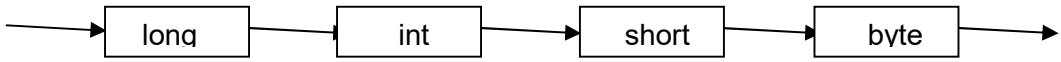
```
class Test
{
   public static void main(String[] args)
   {
      int i = 100;
     // automatic type conversion
      long l = i;
     float f = l;
      System.out.println("Int value "+i);
      System.out.println("Long value "+l);
      System.out.println("Float value "+f);
   }
}
```

Output:

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

NOTE: It is always possible to assign an **int** value to a **long** variable. However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no automatic conversion defined from **double** to **byte**.

2. Explicit type conversion(narrow conversion): It is process the conversion of big data type into small data type, used to reduce the size of memory of variable and it may loss of data it is also called as narrow conversion



Narrow conversion

Example 1: //Java program to illustrate explicit type conversion

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

Output:

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

Example 2://Java program to illustrate Conversion of int and double to byte

```
class Test {
   public static void main(String args[])
   {
      byte b;
      int i = 257;
      double d = 323.142;
      System.out.println("Conversion of int to byte.");
      b = (byte) i; //i%256
      System.out.println("i = " + i + " b = " + b);
      System.out.println("\nConversion of double to byte.");
      b = (byte) d; //d%256
      System.out.println("d = " + d + " b= " + b);
   }
}
```

Output:
Conversion of int to byte.
i = 257 b = 1
Conversion of double to byte.
d = 323.142 b = 67

Blocks

- A block is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed. The following listing shows two blocks from the MaxVariablesDemo (in a .java source file) program, each containing a single statement:

```
if (Character.isUpperCase(aChar)) {
System.out.println("The character " + aChar + " is upper case.");
}
else {
System.out.println("The character " + aChar + " is lower case.");
}
```

Control statements

Depending on the results of computations the flow of control may require to jump one part of the program to another part such jumps are called control statement the control statements are of two types:

1. decision control statements
2. loop control statements
3. jump control statements

1. Decision control statements:

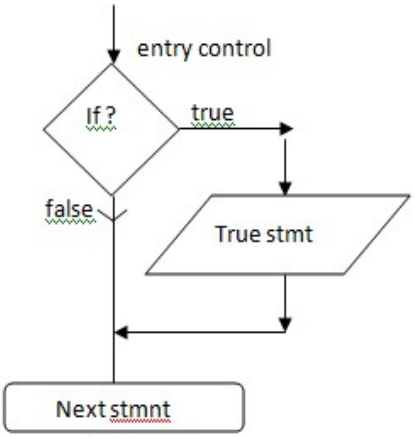
Depending on the result of evalution of an expression a statements are executed.java provide the following decision control statements.

- if/statements
- if/else statements
- nested if/else statements
- if/else if ladder statements
- Switch statement

a) ifstatement: The if statement conditionally process ,when the condition is true ,it selects only one item from one

Syntax:
```
if(condition)
    {
  // body of code
    }
```

flowchart:

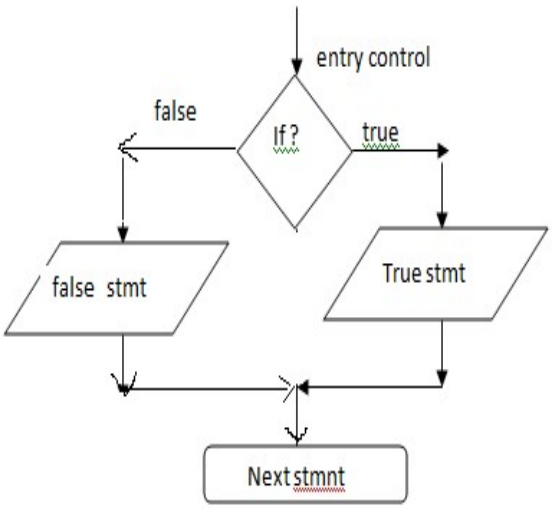


Example:
```
import java.io.*;
class  ifstatement
{
public static void main( String args[])
{
int a=10,b=20;
 if(a>b)
{
System.out.println("biggest is =:" +  a);
}
}
}
```

b) if/else statements: The if/else statement conditionally process, it selects only one item from two different items when the condition is true.

Syntax:
```
if(condition)
    {
  // true statement;
    }
  else
    {
  //false statement;
    }
```

flow chart:



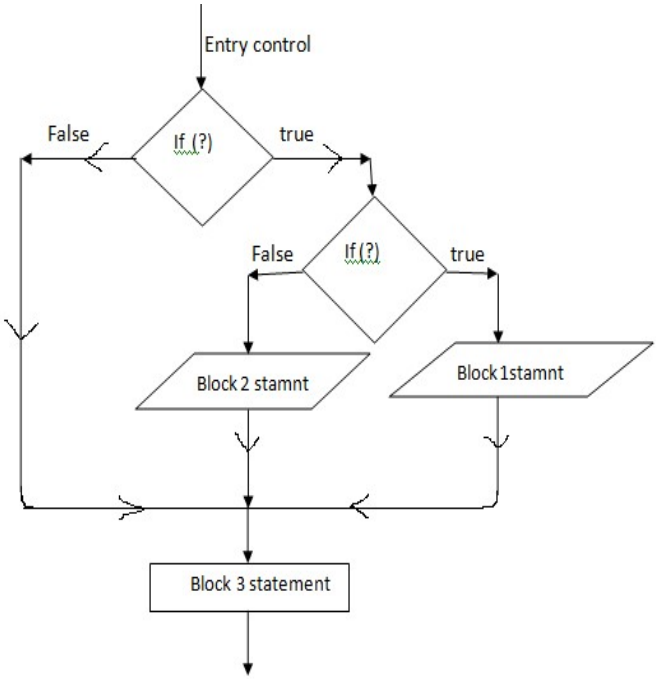
Example:
```
import java.io.*;
class  ifelse
{
public static void main( String args[])
{
int a=10,b=20;
if(a>b){
System.out.println("biggest is =:" +  a);
}
else{
System.out.println("biggest is =:" +  b);
}
}
}
```

c) Nested if/else statements: the if/else statement is placed within another if –else statement it is also conditionally process, it selects only one item from multiple selections of items when the condition is true

Syntax:

```
if(condition)
 {
  else if(condition)
    {
  //block1  statement;
     }
  else
    {
  //block2 statements;
    }
 }
else
{
 //block 3 statemnts;
}
```

flow chart:



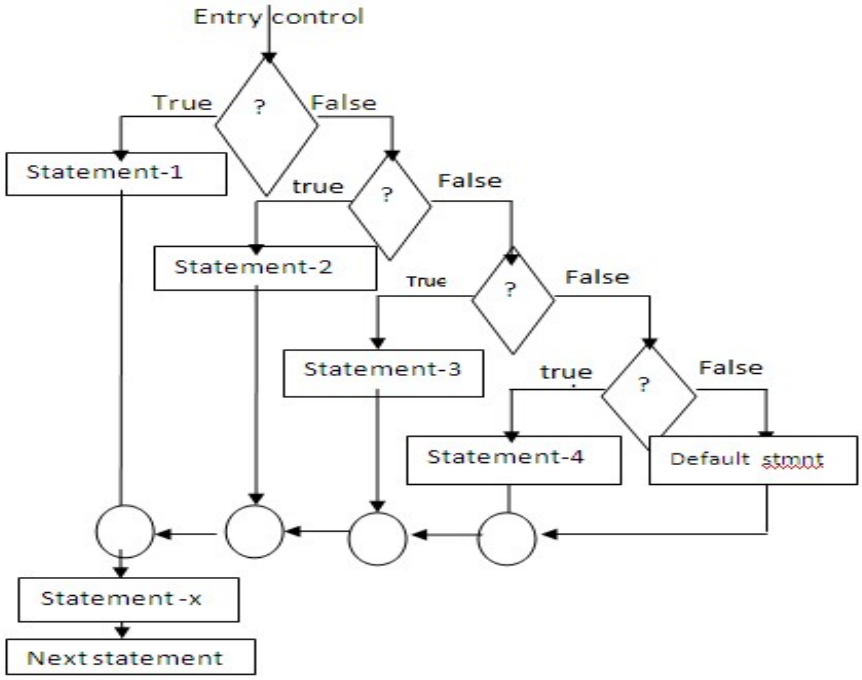
Example:

```
import java.io.*;
import java.util.Scanner;
class nestedifelse
      {
  public static void main(String args[])
        {
        int sal;
        int tot=0;
        Scanner sc=new Scanner(System.in);
System.out.println("enter the basic salary");
sal=sc.nextInt();
       if((sal>5000 &&sal<10000))
      {
        if((sal>=10000 &&sal<20000))
        {
int hra=4000,ta=200;
        tot=sal+hra+ta;
System.out.println("total sal is=:" + tot);
        }
        else
        {
        int hra=2500,ta=100;
        tot=sal+hra+ta;
System.out.println("total sal is=:" + tot);
        }
        } //end of outer if
    else
```

```
        {
    int hra=5000,ta=250;
     tot=sal+hra+ta;
System.out.println("total sal is=:" + tot);
        }
        }
        }
```

d) The else-if ladder: It consisting the multi path decisions the conditions are evaluated from top to bottom it is also called as entry control loop, when all the conditions becomes the false then the final else containing the default statement will be executed

```
if (condition-1)
   Statement-1;
 else if (condition-2)
   Starement-2;
 else if (condition-3)
   Statement-3;
 else if (condition-n)
   Statement-n;
else
  default statement;
Statement-x;
```



Example://Java Program to demonstrate the use of If else-if ladder.
//It is a program of grading system for fail, D grade, C grade, B grade, A grade and A+.

```
public class IfElseIfExample {
public static void main(String[] args) {
   int marks=65;
   if(marks<50){
      System.out.println("fail");
   }
   else if(marks>=50 && marks<60){
      System.out.println("D grade");
   }
   else if(marks>=60 && marks<70){
      System.out.println("C grade");
   }
   else if(marks>=70 && marks<80){
      System.out.println("B grade");
   }
```

```
        else if(marks>=80 && marks<90){
            System.out.println("A grade");
        }else if(marks>=90 && marks<100){
            System.out.println("A+ grade");
        }else{
            System.out.println("Invalid!");
        }
    }
    }
```

Output:C grade

Example: //Program to check POSITIVE, NEGATIVE or ZERO:

```
public class PositiveNegativeExample {
public static void main(String[] args) {
    int number=-13;
    if(number>0){
    System.out.println("POSITIVE");
    }else if(number<0){
    System.out.println("NEGATIVE");
    }else{
    System.out.println("ZERO");
    }
}
}
```

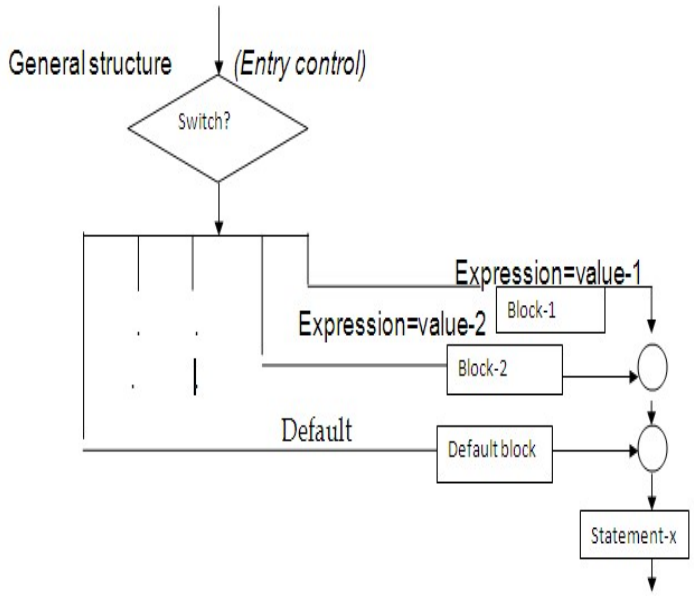
Output: NEGATIVE

d) Switch-Statement: The switch statement will useful for test the multiway decision statements known as switch, it test the value of given variable (or)Expression against a list of case values when a match is found a block of statements associated with that case is executed the general form of switch is as follows

Syntax: Switch (expression)

```
        {
  case value 1: statement-1;
        break
  case value 2: statement-2;
        Break
  case value 3: statement-3;
         break;
        …….
              ……..
default:default statement;
    } Statement-x;
```

flowchart:



Example:

```
import java.util.Scanner;
class Main {
   public static void main(String[] args) {
      char operator;
      Double number1, number2, result;
```

```
        Scanner scanner = new Scanner(System.in);        // create an object of Scanner class
System.out.print("Enter operator (either +, -, * or /): ");
        operator = scanner.next();// ask user to enter operator
System.out.print("Enter number1 and number2 respectively: ");
        // ask user to enter numbers
        number1 = scanner.nextDouble();
        number2 = scanner.nextDouble();
        switch (operator) {
            // performs addition between numbers
            case '+':
                result = number1 + number2;
System.out.print(number1 + "+" + number2 + " = " + result);
                break;
            // performs subtraction between numbers
            case '-':
                result = number1 - number2;
System.out.print(number1 + "-" + number2 + " = " + result);
                break;
            // performs multiplication between numbers
            case '*':
                result = number1 * number2;
System.out.print(number1 + "*" + number2 + " = " + result);
                break;
            // performs division between numbers
            case '/':
                result = number1 / number2;
System.out.print(number1 + "/" + number2 + " = " + result);
                break;
            default:
System.out.println("Invalid operator!");
                break;
        }
    }
}
```

Output:

```
Enter operator (either +, -, * or /): *
Enter number1 and number2 respectively: 1.4
-5.3
1.4*-5.3 = -7.419999999999999
```

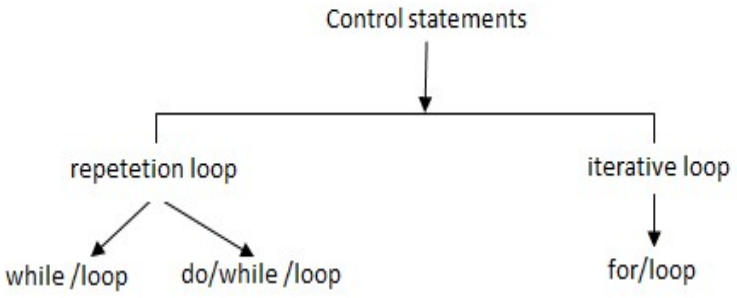
Rules for switch statement:

- The switch expression must be an integral type.
- Case labels must be constants or constants expressions as well as must unique.
- No two labels can have same value, case labels must end with semicolons.
- The break statement transfers the control out of the switch statement.
- The break statement is an optional and the default label is also an optional.

- There can be at least one default label .it is permitted to nest switch statements.

Loop statements

It allows to run block of statements repeatedly for certain number of times if the condition is true the repetition is continuing when the condition is false the repetition stop and control passed to the next statements.java provides three types of loops control statements



- whileloop
- do …. while loop
- forloop

1. While loop:

- It is an entry control loop, executes the statements repeatedly based on the condition
- If the condition is true executes the block of statements reputedly, when the condition becomes false the control immediately pass to the next statements the statements are simple or compound

Syntax: while(condition)

```
{
//block of ststements;
}
```

Example 1: /* print the series of no 1 to 20*/

```
import java.io.*;
import  java.util.Scanner;
class count
{
public static void main(String args[])
{
 int i=1;
  while(i<21)
  {
System.out.println(" i value is =:" + i);
i=i+1;
    }
   }
  }
```

Example 2: /* WAP to print Fibonacci series between 20 */

```
import   java.io.*;
  import   java.util.Scanner;
class fibonacci
   {
  public static void main(String args[])
    {
```

```
    int a=0,b=1,c=0;
System.out.println(a);
System.out.println(b);
    while(c<21)
     {
      c=a+b;
System.out.println("\t" + c);
      a=b;
      b=c;
        }
       }
      }
```

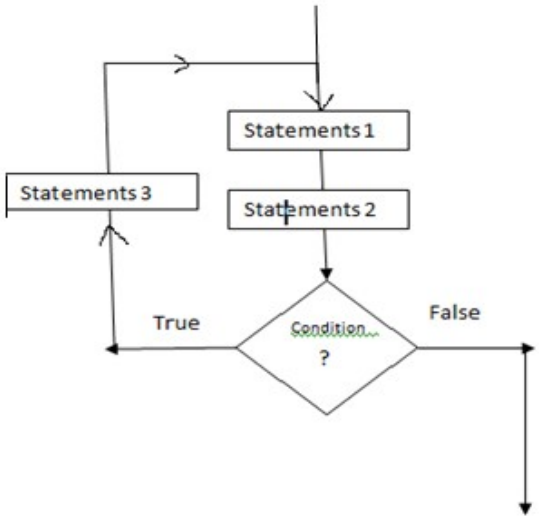
2. Do /while loop:

- It is an exit control loop that is executes the program first and then check the condition next this process repeatedly done until to satisfy the condition,
- Note: it executes the block of statements at least once

Syntax:do

```
{
 -------------
 --------------
} while (condition);
```

flowchart:



Example:

```
import java.io.*;
 import java.util.Scanner;
 class dowhile
 {
 public static void main(String args[])
  {
  int i =1;
do {
System.out.println(" i value is =:" + i);
i=i+1;
        } while(i<21);
      }
    }
```

3. For/loop:

- It is also an entry control loop. It executes the section of code a fixed number of times. It is usually used when we know before entering the loop how many times,we want to execute the code.
- It is a powerful and dynamic loop for processing yields and i.e. it checks the condition first and then executes the program next, it is the counter-based loop provides more concise loop control structure. The general form of the for is as follows

Syntax:

```
for (initialization; test condition; increment/decrement)
{
  Body of the loop;
```

```
}
```

- Initialization: this section will useful for declare the initialization values to the variable
  Ex:i=1,n=1
- Test condition: used for specifying for the conditional expressions
  Ex: num<100; or i==10;
- Increment/decrement: used for increment the variable value or decrement the variable value
  i=10
  Ex: i++=>10,++i=11 (or) i--=10, --i=9;

It would be declaring different types

```
for(initialization; testcondition;)    for(;testcondition;)          for(;testcond; incre/decre)
   {                                       {                                 {
   ---------------                         ---------------                   ---------------
   ---------------                         ---------------                   ---------------
   Increment/decrement;                 increment/decrement;                 ---------------
   }                                       }                                 }
```

Example:

```
import java.io.*;
class forloop
{
public static void main(String args[])
{
 for(int i=0;i<10;i++)
   {
System.out.println(" i value is =:" + i);
i=i+1;
      }
    }
    }
```

NOTE:

- In place of condition section, it is also possible to use 2 or more relational expressions.
- Multiple initializations are also possible

Jump statements in java

Java supports the following jump statements
1) Break statement
2) Continue statement
3) Return statement
4) Exit statement

1. Break statement: It immediately terminate the loop in a program and generate the results.It used for passes the control to the inner most enclosing while ,for, do, or switch statement. The following program display the message on the screen for 10 times if use presses any key it discontinues with the help of a break statement

Ex: /* example of break */

```
import java.io.*;
class breakstatement
```

```
{
public static void main(String args[])
{
for(int i=0;i<10;i++)
 {
   if(i==5)
    break;
System.out.print("\t" + i);
     }
    }
   }
```

2. Continue statement: It skip the condition statement variable value or statement when it is the loop does not terminate when a continue statement is encountered but the remaining loop statements are skipped and the computation proceeds directly to the next pass through the loop.

Ex: /* example of continue */

```
import java.io.*;
class continuestatement
{
public static void main(String args[])
{
for(int i=0;i<10; i++)
 {
   if(i==5)
  continue;
System.out.print("\t"  +  i);
       }
     }
    }
```

3. return: The last control statement is return. The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method. At any time in a method the return statement can be used to cause execution to branch back to the caller of the method. Thus, the return statement immediately terminates the method in which it is executed.

4. exit():

- The java.lang.System.exit() method exits current program by terminating running Java virtual machine. This method takes a status code. A non-zero value of status code is generally used to indicate abnormal termination.
- Following is the declaration for java.lang.System.exit() method:
  - public static void exit(int status)
  - exit(0) : Generally used to indicate successful termination.
  - exit(1) or exit(-1) or any other non-zero value: Generally indicates unsuccessful termination.

Note: This method does not return any value.

The following example shows the usage of java.lang.System.exit() method.

Example: // A Java program to demonstrate working of exit()

```
import java.util.*;
import java.lang.*;
```

```
class useofexit
{
   public static void main(String[] args)
   {
      for (int i = 0; i<10; i++)
      {
         if (i>= 5)
         {
            System.out.println("exit...");
              // Terminate JVM
            System.exit(0);
         }
         else
            System.out.print("\t " +i);
      }
      System.out.println("End of Program");
   }
}
```

Output: 1 2 3 4
exit...

Reading data from keyboard

There are many ways to read data from the keyboard. For example:

- InputStreamReader and BufferedReader class
- Scanner

a) InputStreamReader class and BufferedReader class

- InputStreamReader class can be used to read data from keyboard.It performs two tasks:
  - connects to input stream of keyboard
  - converts the byte-oriented stream into character-oriented stream
- BufferedReader class
  - BufferedReader class can be used to read data line by line by readLine() method.

Example 1: Reading data from keyboard by InputStreamReader and BufferdReader class

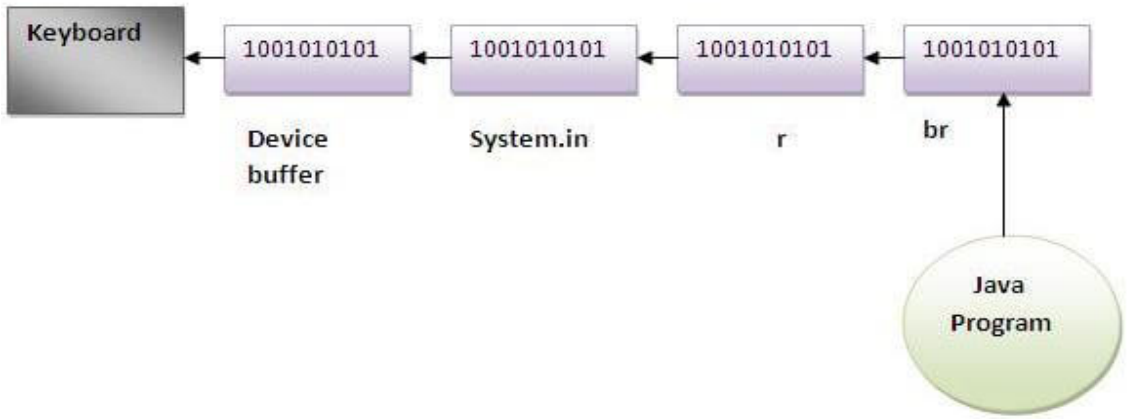
In this example, we are connecting the BufferedReader stream with the InputStreamReader stream for reading the line by line data from the keyboard.

```
import java.io.*;
class readfromkeyboard  {
public static void main(String args[])throws Exception{
InputStreamReader r=new InputStreamReader(System.in);
BufferedReader br=new BufferedReader(r);
System.out.println("Enter your name");
String name=br.readLine();
System.out.println("Welcome "+name);
 }
}
```

Output:

Enter your name

Chinmayee
Welcome Chinmayee



Example 2:Reading data from keyboard by InputStreamReader and BufferdReader class until the user writes stop

```
import java.io.*;
class readfromkeyboard{
public static void main(String args[])throws Exception{
  InputStreamReader r=new InputStreamReader(System.in);
 BufferedReader br=new BufferedReader(r);
  String name="";
 while(!name.equals("stop")){
   System.out.println("Enter data: ");
   name=br.readLine();
   System.out.println("data is: "+name);
  }
 br.close();
 r.close();
 }
}
```

Output:

Enter data: Chinmayee
data is: Chinmayee
Enter data: 10
data is: 10
Enter data: stop
data is: stop

<u>b) Scanner Class in Java</u>

- Scanner is a class in java.util package used for obtaining the input of the primitive types like int, double, etc. and strings. Java provides various ways to read input from the keyboard, the java.util.Scanner class is one of them.
- It is the easiest way to read input in a Java program.
- To create an object of Scanner class, we usually pass the predefined object System.in, which represents the standard input stream. We may pass an object of class File if we want to read input from a file.

- The Java Scanner class breaks the input into tokens using a delimiter which is whitespace by default. It provides many methods to read and parse various primitive values.
- The Java Scanner class is widely used to parse text for strings and primitive types.
- The Java Scanner class provides nextXXX() methods to return the type of value such as nextInt(), nextByte(), nextShort(), next(), nextLine(), nextDouble(), nextFloat(), nextBoolean(), etc. To get a single character from the scanner, you can call next().charAt(0) method which returns a single character.
- Java Scanner Class Declaration

```
public final class Scanner
    extends Object
    implements Iterator<String>
```

How to get Java Scanner?

- To get the instance of Java Scanner which reads input from the user, we need to pass the input stream (System.in) in the constructor of Scanner class. For Example:

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

- To get the instance of Java Scanner which parses the strings, we need to pass the strings in the constructor of Scanner class. For Example:

```
Scanner in = new Scanner("Hello Javatpoint");
```

Java Scanner Class Methods

The following are the list of Scanner methods:

boolean	nextBoolean()	It scans the next token of the input into a boolean value and returns that value.
byte	nextByte()	It scans the next token of the input as a byte.
double	nextDouble()	It scans the next token of the input as a double.
float	nextFloat()	It scans the next token of the input as a float.
int	nextInt()	It scans the next token of the input as an Int.
String	nextLine()	It is used to get the input string that was skipped of the Scanner object.
long	nextLong()	It scans the next token of the input as a long.

Example 1:

Use of Java Scanner class to read a single input (a string through in.nextLine() method) from the user.

```
import java.util.*;
public class ScannerExample {
public static void main(String args[]){
    Scanner in = new Scanner(System.in);
    System.out.print("Enter your name: ");
    String name = in.nextLine();
    System.out.println("Name is: " + name);
    in.close();
    }
}
```

Output:

```
Enter your name: Chinmayee Rout
Name is: Chinmayee Rout
```

Example 2:

//Program to accept multiple data from user

```
import java.util.Scanner;
public class ScannerClassExample{
    public static void main(String args[]){
        String s = "Hello, welcome to Java Programming";
        //Create scanner Object and pass string in it
        Scanner scan = new Scanner(s);
        //Check if the scanner has a token
        System.out.println("Boolean Result: " + scan.hasNext());
        //Print the string
        System.out.println("String: " +scan.nextLine());
        scan.close();
        System.out.println("--------Enter Your Details-------- ");
        Scanner in = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = in.next();
        System.out.println("Name: " + name);
        System.out.print("Enter your age: ");
        int i = in.nextInt();
        System.out.println("Age: " + i);
        System.out.print("Enter your CGPA: ");
        double d = in.nextDouble();
        System.out.println("CGPA: " + d);
        in.close();
        }
}
```

Output:

```
Boolean Result: true
String: Hello, welcome to Java Programming
-------Enter Your Details---------
Enter your name: ABC
Name: ABC
Enter your age: 20
Age: 20
Enter your CGPA: 9.5
CGPA: 9.5
```

Example 3:

//program to separate tokens based on specified delimeter

```
import java.util.*;
public class ScannerClassExample {
    public static void main(String args[]){
        String str = "Hello/This is JavaProgramming class/I am Chinmayee.";
        //Create scanner with the specified String Object
        Scanner scanner = new Scanner(str);
        System.out.println("Boolean Result: "+scanner.hasNextBoolean());
        //Change the delimiter of this scanner
        scanner.useDelimiter("/");
```

```
        //Printing the tokenized Strings
        System.out.println("---Tokenizes String---");
      while(scanner.hasNext()){
         System.out.println(scanner.next());
      }
        //Display the new delimiter
        System.out.println("Delimiter used: " +scanner.delimiter());
        scanner.close();
        }
    }
```

Output:

```
Boolean Result: false
---Tokenizes String---
Hello
This is JavaProgramming class/
I am Chinmayee.
Delimiter used: /
```

Example 4:

```
// Java program to read some values using Scanner class and print their mean.
importjava.util.Scanner;
 publicclassScannerDemo
{
   publicstaticvoidmain(String[] args)
   {
      // Declare an object and initialize with predefined standard input object
      Scanner sc = newScanner(System.in);
      // Initialize sum and count of input elements
      intsum = 0, count = 0;
      // Check if an int value is available
      while(sc.hasNextInt())
      {
         intnum = sc.nextInt();          // Read an int value
         sum += num;
         count++;
      }
      intmean = sum / count;
      System.out.println("Mean: "+ mean);
   }
}
```

Output:

```
101
223
238
892
99
500
```

728
Mean: 397

Arrays

Definition: An array is a group of related data items which are stored in homogeneous memory location that shares a common name.
Particular value indicated by writing a number called "index number" arrays are classified into different types they are

- **one dimensional array**
- **two-dimensional array**
- **Multi-dimensional array**

1. One dimensional array:

It is list of homogenous items can be given one variable name using only one subscript. Individual e value of an array is called an element array must be declared and created in the computer memory before they are used creation of an array contains two sections

- declarations of array
- creating memory locations

Syntax 1: type arrayname[];//declaration
type arrayname[]=new type [size];
Type→refers datatype
Arrayname→defines the name of array.
Example: int number[];
int number[]=new int[5];
Syntax2: type arrayname[]={ list of values };
Example: int number[]={34,45,56,67};
Example: // WAP in java to find sum of an array element

```
import java.util.Scanner;
publicclassArray_Sum
{
publicstaticvoid main(String[]args)
{
int n, sum =0;
      Scanner s =new Scanner(System.in);
System.out.print("Enter no. of elements you want in array:");
      n =s.nextInt();
int a[]=newint[n];
System.out.println("Enter all the elements:");
for(inti=0;i< n;i++)
{
        a[i]=s.nextInt();
        sum = sum + a[i];
}
System.out.println("Sum:"+sum);
}
}
```

Output:

```
Enter no. of elements you want in array:5
Enter all the elements:
1
2
3
4
5
Sum:15
```

Passing Array to a Method in Java

Program://Java Program to demonstrate the way of passing an array to method.

```
class Testarray2{
//creating a method which receives an array as a parameter
static void min(int arr[]){
int min=arr[0];
for(int i=1;i<arr.length;i++)
 if(min>arr[i])
  min=arr[i];
System.out.println(min);
}
public static void main(String args[]){
int a[]={33,3,4,5};//declaring and initializing an array
min(a);//passing array to method
}
}
```

Output: 3

Anonymous Array in Java

- Java supports the feature of an anonymous array, so you don't need to declare the array while passing an array to the method.

Program: //Java Program to demonstrate the way of passing an anonymous array to method.

```
public class TestAnonymousArray{
//creating a method which receives an array as a parameter
static void printArray(int arr[]){
for(int i=0;i<arr.length;i++)
System.out.println(arr[i]);
}
public static void main(String args[]){
printArray(new int[]{11,22,33,44});//passing anonymous array to method
}
}
```

Output:

```
11
22
33
44
```

Returning Array from the Method

Program: //Java Program to return an array from the method

```
class TestReturnArray{
//creating method which returns an array
static int[] get(){
return new int[]{10,30,50,70,90};
}
public static void main(String args[]){
//calling method which returns an array
int arr[]=get();
//printing the values of an array
for(int i=0;i<arr.length;i++)
System.out.print("\t"+arr[i]);
}
}
```

Output:

```
10      30      50      70      90
```

2. Two dimensional array:

- Twodimensional array is collection of homogenous data items that shares common name using two subscripts.it stores table of data. it the best for representation for matrix
- First subscripts refers the number of rows and second subscripts refers the number of columns.

Syntax: type arrayname[][];//array declaration.
type arrayname[][]=new int[size1][size2];//allocation of memory.

Example:

1. Addition of 2 Matrices in Java

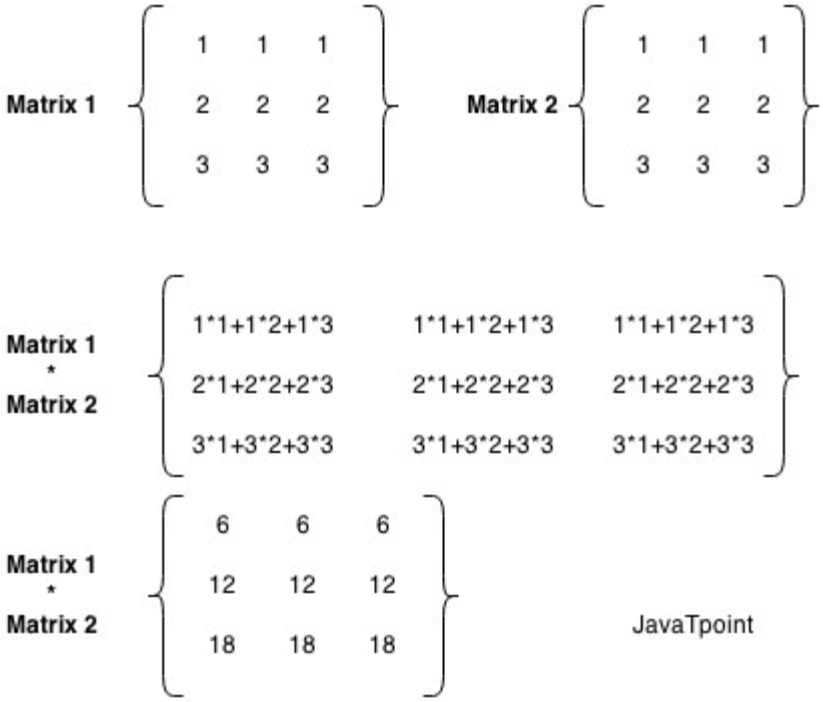
```
//Java Program to demonstrate the addition of two matrices in Java
class arrayaddition{
public static void main(String args[]){
//creating two matrices
int a[][]={{1,3,4},{3,4,5}};
int b[][]={{1,3,4},{3,4,5}};
//creating another matrix to store the sum of two matrices
int c[][]=new int[2][3];
//adding and printing addition of 2 matrices
for(int i=0;i<2;i++){
for(int j=0;j<3;j++){
c[i][j]=a[i][j]+b[i][j];
System.out.print(c[i][j]+" ");
}
System.out.println();//new line
}
}
}
```

Output:

```
2       6       8
6       8       10
```

2. Multiplication of 2 Matrices in Java

In the case of matrix multiplication, a one-row element of the first matrix is multiplied by all the columns of the second matrix which can be understood by the image given below.



Program://Java Program to multiply two matrices

```
public class MatrixMultiplication{
public static void main(String args[]){
//creating two matrices
int a[][]={{1,1,1},{2,2,2},{3,3,3}};
int b[][]={{1,1,1},{2,2,2},{3,3,3}};
//creating another matrix to store the multiplication of two matrices
int c[][]=new int[3][3];  //3 rows and 3 columns
//multiplying and printing multiplication of 2 matrices
for(int i=0;i<3;i++){
for(int j=0;j<3;j++){
c[i][j]=0;
for(int k=0;k<3;k++)
{
c[i][j]=c[i][j] + a[i][k]*b[k][j];
}//end of k loop
System.out.print(c[i][j]+" ");  //printing matrix element
}//end of j loop
System.out.println();//new line
}
}
}
```

Output:

```
6    6    6
```

12 12 12
18 18 18

Multi –dimensional Array:

- Multi-dimensional array is collection of homogenous data items that shares common name using multi subscripts

Syntax:type arrayname[][]…[];
type arrayname[][]….[]=new int[size1][size2[size3]….[size n];

MODULE-2

Chapter – 1

LECTURE-1

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

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

Classes and Objects:

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

i. Objects in Java:

- Software objects have a state and a behaviour. A software object's state is stored in fields and behaviour is shown via methods. So in software development, methods operate on the internal state of an object and the object-to-object communication is done via methods.
- An entity that has state and behaviour is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.
- An object has three characteristics:
  - **State:** represents the data (value) of an object.
  - **Behaviour:** represents the behaviour (functionality) of an object such as deposit, withdraw, etc.
  - **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.
- For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behaviour.
- **An object is an instance of a class.** A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.
- **Object Definitions:**
  - An object is *a real-world entity*.
  - An object is *a runtime entity*.
  - The object is *an entity which has state and behaviour*.
  - The object is *an instance of a class*.

ii. Classes in Java:

- A class is a blueprint from which individual objects are created.
- A class can contain any of the following variable types.
  - Local variables: Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
  - Instance variables: Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
  - Class variables: Class variables are variables declared within a class, outside any method, with the static keyword.

- A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.
- A class in Java can contain:
  - Fields (attributes)
  - Methods
  - Constructors
  - Blocks
  - Nested class and interface
- **Syntax to declare a class:**

```
class <class_name>{
   field;
   method;
}
```

Instance variable in Java

- A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

Method in Java

- In Java, a method is like a function which is used to expose the behavior of an object.
- Advantage of Method
  - Code Reusability
  - Code Optimization

"new" keyword in Java

- The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

i. Object and Class Example: main within the class

- In this example, we have created a Student class which has two data members id and name. We are creating the object of the Student class by new keyword and printing the object's value.
- Here, we are creating a main() method inside the class.

File: Student.java

```
//Java Program to illustrate how to define a class and fields
//Defining a Student class.
class Student{
 int id; //field or data member or instance variable
 String name;   //creating main method inside the Student class
  public static void main(String args[]){
  Student s1=new Student();//creating an object of Student
  System.out.println(s1.id);//accessing member through reference variable
  System.out.println(s1.name);
 }
}
```

Output:

```
0
null
```

ii. Object and Class Example: main outside the class

- We can have multiple classes in different Java files or single Java file. If you define multiple classes in a single Java source file, it is ideal to save the file name with the class name which has main() method.

File: TestStudent.java

```
/*Java Program to demonstrate having the main method in another class*/
class Student
{
 int id;
 String name;
```

```
}
//Creating another class TestStudent1 which contains the main method
class TestStudent{
 public static void main(String args[]){
  Student s1=new Student();
  System.out.println(s1.id);
  System.out.println(s1.name);
 }
}
```

Output:

```
0
Null
```

LECTURE-2

Initialization of object

There are 3 ways to initialize object in Java.

- By reference variable
- By method
- By constructor

1) Object and Class Example: Initialization through reference

- Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

File: TestStudent.java

```
class Student{
 int id;
 String name;
}
class TestStudent{
 public static void main(String args[]){
  Student s1=new Student();
  s1.id=101;
  s1.name="ABC";
  System.out.println(s1.id+" "+s1.name); //printing members with a white space
 }
}
```

Output: 101 ABC

We can also create multiple objects and store information in it through reference variable.

File: TestStudent.java

```
class Student
{
 int id;
 String name;
}
class TestStudent{
 public static void main(String args[]){
  Student s1=new Student();
  Student s2=new Student();
  s1.id=101;
  s1.name="ABC";
  s2.id=102;
  s2.name="XYZ";
```

```
  System.out.println(s1.id+" "+s1.name);
  System.out.println(s2.id+" "+s2.name);
 }
}
```

Output:

```
101     ABC
102     XYZ
```

2) Object and Class Example: Initialization through method

- Here we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

File: TestStudent.java

```
class Student{
 int rollno;
 String name;
 void insertRecord(int r, String n){
  rollno=r;
  name=n;
 }
 void displayInformation(){System.out.println(rollno+" "+name);}
}
class TestStudent{
 public static void main(String args[]){
  Student s1=new Student();
  Student s2=new Student();
  s1.insertRecord(111,"ABC");
  s2.insertRecord(222,"XYZ");
  s1.displayInformation();
  s2.displayInformation();
 }
}
```

Output:

```
111     ABC
222     XYZ
```

EXAMPLE – 2: Object and Class Example: Rectangle

File: TestRectangle.java

```
class Rectangle{
 int length;
 int width;
 void insert(int l, int w){    length=l;    width=w;   }
 void calculateArea()
{ System.out.println("Area of rectangle is:" + length*width); }
}
class TestRectangle{
 public static void main(String args[]){
  Rectangle r1=new Rectangle();
  Rectangle r2=new Rectangle();
  r1.insert(11,5);
  r2.insert(3,15);
  r1.calculateArea();
```

```
  r2.calculateArea();
}
}
```

Output:

Area of rectangle is: 55
Area of rectangle is: 45

Anonymous object

- Anonymous simply means nameless. An object which has no reference is known as an anonymous object. It can be used at the time of object creation only.
- If you have to use an object only once, an anonymous object is a good approach.
- For example: **new** Calculation();//anonymous object
- Calling method through a reference:

```
Calculation c=new Calculation();
c.fact(5);
```

- Calling method through an anonymous object

```
new Calculation().fact(5);
```

Let's see the full example of an anonymous object in Java.

```
class Calculation{
 void fact(int  n)
{
  int fact=1;
  for(int i=1;i<=n;i++){     fact=fact*i;    }
 System.out.println("factorial is "+fact);
}
public static void main(String args[]){
 new Calculation().fact(5);//calling method with anonymous object
}
}
```

Output: Factorial is 120

LECTURE-3
Encapsulation in Java

- Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. Another way to think about encapsulation is, it is a protective shield that prevents the data from being accessed by the code outside this shield.
- Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of own class in which they are declared.
- As in encapsulation, the data in a class is hidden from other classes, so it is also known as data-hiding. Encapsulation can be achieved by Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.

Example: // Java program to demonstrate encapsulation

```
public class Encapsulate {
// private variables declared and can only be accessed by public methods of class
private String Name;
private int Roll;
private int Age;
 // get method for age to access private variable Age
public int getAge()
{        return Age;     }
 // get method for name to access private variable Name
```

```
    public String getName()
    {       return Name;    }
     // get method for roll to access private variable Roll
    public int getRoll()
    {       return Roll;    }
     // set method for age to access private variable age
    public void setAge(int newAge)
    {       Age = newAge;    }
     // set method for name to access private variable Name
    public void setName(String newName)
    {       Name = newName;    }
     // set method for roll to access private variable Roll
    public void setRoll(int newRoll)
    {       Roll = newRoll;    }
  } // end of class Encapsulate
  class TestEncapsulation {
    public static void main(String[] args)
    {
      Encapsulate obj = new Encapsulate();
      obj.setName("ABC");
      obj.setAge(20);
      obj.setRoll(111);
      System.out.println("'name: " + obj.getName());
      System.out.println("'age: " + obj.getAge());
      System.out.println("'roll: " + obj.getRoll());
       // Direct access of Roll is not possible due to encapsulation
      // System.out.println("'s roll: " + obj.Name); // will generate error
    }
  }
```

Output:

```
name: ABC
age: 20
roll: 111
```

Abstraction in Java

- Data Abstraction is the property by virtue of which only the essential details are displayed to the user. The trivial or the non-essentials units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components.
- Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details. The properties and behaviours of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

Example: // Java program to illustrate the concept of Abstraction

```
  abstract class Shape {
    String color;
     // these are abstract methods
    abstract double area();
    public abstract String toString();
    // abstract class can have a constructor
    public Shape(String color)
    {
      System.out.println("Shape constructor called");
      this.color = color;
```

```
    }
    // this is a concrete method
    public String getColor()
    {       return color;     }
}
class Circle extends Shape {
    double radius;
    public Circle(String color, double radius)
    {
      // calling Shape constructor
        super(color);
        System.out.println("Circle constructor called");
        this.radius = radius;
    }
  // override
    double area()
    {      return Math.PI * Math.pow(radius, 2);
   }
    public String toString()
    {
        return "Circle color is " + super.color  + "and area is : "   + area();
    }
}
 class Rectangle extends Shape {
     double length, width;
     public Rectangle(String color, double length, double width)
    {
         // calling Shape constructor
        super(color);
        System.out.println("Rectangle constructor called");
        this.length = length;
        this.width = width;
    }

  double area()
    {        return length * width;     }
 public String toString()
    {
        return "Rectangle color is "   + super.color   + "and area is : "  + area();
    }
}

public class Test {
    public static void main(String[] args)
    {
        Shape s1 = new Circle("Red", 2.2);
        Shape s2 = new Rectangle("Yellow", 2, 4);
        System.out.println(s1.toString());
        System.out.println(s2.toString());
    }
}
```

Output:

```
Shape constructor called
Circle constructor called
Shape constructor called
Rectangle constructor called
Circle color is Redand area is : 15.205308443374602
Rectangle color is Yellowand area is : 8.0
```

Difference between Abstraction and Encapsulation:

ABSTRACTION	ENCAPSULATION
Abstraction is the process or method of gaining the information.	While encapsulation is the process or method to contain the information.
In abstraction, problems are solved at the design or interface level.	While in encapsulation, problems are solved at the implementation level.
Abstraction is the method of hiding the unwanted information.	Whereas encapsulation is a method to hide the data in a single entity or unit along with a method to protect information from outside.
We can implement abstraction using abstract class and interfaces.	Whereas encapsulation can be implemented using by access modifier i.e. private, protected and public.
In abstraction, implementation complexities are hidden using abstract classes and interfaces.	While in encapsulation, the data is hidden using methods of getters and setters.
The objects that help to perform abstraction are encapsulated.	Whereas the objects that result in encapsulation need not be abstracted.

LECTURE- 4

Method Overloading in Java

- If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**. Here methods have different signatures where the signature can differ by the number of input parameters or type of input parameters or both. Overloading is related to compile-time (or static) polymorphism.
- Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int, int) for two parameters, and b(int,int,int) for three parameters then it may be difficult to understand the behavior of the method because its name differs. Hence we can use a method having same name with different parameter.

Advantage of method overloading

- Method overloading *increases the readability of the program*.
- Different ways to overload the method

There are two ways to overload the method in java

- By changing number of arguments
- By changing the data type

In java, Method Overloading is not possible by changing the return type of the method only.

1) Method Overloading: changing no. of arguments

- In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

```
class Adder{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
class TestOverloading1{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}
}
```

Output:

```
22
33
```

2) Method Overloading: changing data type of arguments

```
// Java program to demonstrate working of method overloading in Java.
  // Overloaded sum(). This sum takes two int parameters
   public int sum(int x, int y)
   {       return (x + y);     }
     // Overloaded sum(). This sum takes three int parameters
   public int sum(int x, int y, int z)
   {       return (x + y + z);     }
     // Overloaded sum(). This sum takes two double parameters
   public double sum(double x, double y)
   {        return (x + y);     }
   public static void main(String args[])
   {
      Sum s = new Sum();
      System.out.println("addition of two integers: " + s.sum(10, 20));
      System.out.println("addition of three integers: " +s.sum(10, 20, 30));
      System.out.println("addition of two double values: " +s.sum(10.5, 20.5));
   }
}
```

Output :

```
addition of two integers: 30
addition of three integers: 60
addition of two double values: 31.0
```

<u>QUESTIONS:</u>

Q) Why Method Overloading is not possible by changing the return type of method only?

ANS: In java, method overloading is not possible by changing the return type of the method only because of ambiguity. Let's see how ambiguity may occur:

```
class Adder{
static int add(int a,int b){return a+b;}
static double add(int a,int b){return a+b;}
}
class TestOverloading3{
```

```
public static void main(String[] args){
System.out.println(Adder.add(11,11));//ambiguity
}
}
```

Output:

Compile Time Error: method add(int,int) is already defined in class Adder

NOTE: System.out.println(Adder.add(11,11)); Here, java compiler can not determine which sum() method should be called.

LECTURE- 5

Constructors in Java

- In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.
- It is a special type of method which is used to initialize the object.
- Every time an object is created using the new() keyword, at least one constructor is called.
- It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

Note: It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

Rules for creating Java constructor

The rules defined for the constructor are:

- Constructor name must be the same as its class name
- A Constructor must have no explicit return type
- A Java constructor cannot be abstract, static, final, and synchronized

Note: We can use **access modifiers** while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.

Types of Java constructors

There are two types of constructors in Java:

- Default constructor (no-arg constructor)
- Parameterized constructor

i. Java Default Constructor:

- A constructor is called "Default Constructor" when it doesn't have any parameter.

Syntax: <class_name>(){}

Example: //Java Program to create and call a default constructor

```
class defaultcon{
//creating a default constructor
defaultcon (){System.out.println("Default Constructor is created");}
public static void main(String args[]){
defaultcon b=new defaultcon ();  //calling a default constructor
}
}
```

Output: Default Constructor is created

NOTE: If there is no constructor in a class, compiler automatically creates a default constructor.

QUESTION:

Q) What is the purpose of a default constructor?

ANSWER: The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

Example: Default constructor that displays the default values

```
class Student {
int id;
String name;
void display (){
System.out.println(id+" "+name);
}
public static void main (String args[]){
Student s1=new Student ();
Student s2=new Student ();
s1.display();
s2.display();
}
}
```

Output:

```
0 null
0 null
```

Explanation: In the above class, we are not creating any constructor so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.

ii. Java Parameterized Constructor

- A constructor which has a specific number of parameters is called a parameterized constructor.
- The parameterized constructor is used to provide different values to distinct objects. However, we can provide the same values also.

Example: //Java Program to demonstrate the use of the parameterized constructor.

```
class Student{
   int id;
   String name;
   //creating a parameterized constructor
   Student (int i, String n){
   id = i;
   name = n;
   }
   //method to display the values
   void display (){ System.out.println (id+" "+name);}
  public static void main (String args[]){
   //creating objects and passing values
   Student s1 = new Student (111,"ABC");
   Student s2 = new Student (222,"MNO");
   //calling method to display the values of object
   s1.display();
   s2.display();
  }
}
```

Output:

```
111 ABC
222 MNO
```

Constructor Overloading in Java

- In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.
- Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.
- Overloaded constructor is called based upon the parameters specified when **"new"** is executed.

Example of Constructor Overloading

```
class Student {
   int id;
   String name;
   int age;
   //creating two arg constructor
   Student (int i, String n){
   id = i;
   name = n;
   }
   //creating three arg constructor
   Student (int i, String n, int a){
   id = i;
   name = n;
   age=a;
   }
   void display(){System.out.println(id+" "+name+" "+age);}
  public static void main(String args[]){
   Student s1 = new Student (111,"ABC");
   Student s2 = new Student (222,"XYZ",25);
   s1.display();
   s2.display();
  }
}
```

Output:

```
111    ABC   0
222    XYZ   25
```

Q) When do we need Constructor Overloading?

ANSWER: Sometimes there is a need of initializing an object in different ways. This can be done using constructor overloading.

Example:

```
class Box
{
   double width, height, depth;
     // constructor used when all dimensions specified
   Box(double w, double h, double d) {
      width = w;
      height = h;
      depth = d;
   }
   // constructor used when no dimensions specified
   Box()
   {       width = height = depth = 0;     }
```

```
    // constructor used when cube is created
    Box(double len)
    {       width = height = depth = len;     }
    // compute and return volume
    double volume()
    {       return width * height * depth;     }
}
public class Test
{
    public static void main(String args[])
    {
        // create boxes using the various constructors
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);
        double vol;
       // get volume of first box
        vol = mybox1.volume();
        System.out.println(" Volume of mybox1 is " + vol);
       // get volume of second box
        vol = mybox2.volume();
        System.out.println(" Volume of mybox2 is " + vol);
       // get volume of cube
        vol = mycube.volume();
        System.out.println(" Volume of mycube is " + vol);
    }
     }
```

Output:

```
Volume of mybox1 is 3000.0
Volume of mybox2 is 0.0
Volume of mycube is 343.0
```

Important points to be taken care while doing Constructor Overloading :

- Constructor calling must be the **first** statement of constructor in Java.
- If we have defined any parameterized constructor, then compiler will not create default constructor. and vice versa if we don't define any constructor, the compiler creates the default constructor(also known as no-arg constructor) by default during compilation
- Recursive constructor calling is invalid in java.

<u>Difference between constructor and method in Java</u>

Java Constructor	Java Method
• A constructor is used to initialize the state of an object.	• A method is used to expose the behavior of an object.
• A constructor must not have a return type.	• A method must have a return type.
• The constructor is invoked implicitly.	• The method is invoked explicitly.
• The Java compiler provides a default constructor if you don't have any constructor in a class.	• The method is not provided by the compiler in any case.
• The constructor name must be same as the class name.	• The method name may or may not be same as the class name.

Java Copy Constructor

- We can copy the values from one object to another using copy constructor.

Example: //Java program to initialize the values from one object to another object.

```
class Student {
   int id;
   String name;
   //constructor to initialize integer and string
   Student (int i,String n){
   id = i;
   name = n;
    }
   //constructor to initialize another object
   Student (Student s){
   id = s.id;
   name =s.name;
   }
   void display (){System.out.println(id+" "+name);}
   public static void main (String args[ ]){
   Student s1 = new Student (111,"ABC");
   Student s2 = new Student (s1);
   s1.display();
   s2.display();
  }
}
```

Output:

```
111 ABC
111 ABC
```

Q) Does constructor return any value?

ANSWER: Yes, it is the current class instance (You cannot use return type yet it returns a value).

Q) Can constructor perform other tasks instead of initialization?

ANSWER: Yes, like object creation, starting a thread, calling a method, etc. You can perform any operation in the constructor as you perform in the method.

Q) What is the purpose of Constructor class?

ANSWER: Java provides a Constructor class which can be used to get the internal information of a constructor in the class. It is found in the java.lang.reflect package.

LECTURE- 6

JAVA ARRAY OF OBJECT

- As defined by its name, it stores an **array of objects**. Unlike a traditional array that store values like string, integer, Boolean, etc an array of objects stores objects. The array elements store the location of the reference variables of the object.
- **Syntax:** Class obj[ ]= new Class[array_length]

  Example: To create Array Of Objects

  Step 1) Copy the following code into an editor

```
class ObjectArray{
  public static void main(String args[]){
    Account obj[] = new Account[2] ;
    //obj[0] = new Account();
    //obj[1] = new Account();
   obj[0].setData(1,2);
   obj[1].setData(3,4);
   System.out.println("For Array Element 0");
```

```
    obj[0].showData();
    System.out.println("For Array Element 1");
     obj[1].showData();
  }
}
class Account{
  int a;
  int b;
 public void setData(int c, int d){
   a=c;
   b=d;
 }
 public void showData(){
   System.out.println("Value of a ="+a);
   System.out.println("Value of b ="+b);
 }
}
```

Step 2) Save , Compile & Run the Code.
Step 3) Error=? Try and debug before proceeding to step 4.
Step 4) The line of code, Account obj[] = new Account[2]; exactly creates an array of two reference variables as shown below
Step 5) Uncomment Line # 4 & 5. This step creates objects and assigns them to the reference variable array as shown below. Your code must run now.
Output:

```
For Array Element 0
Value of a =1
Value of b =2
For Array Element 1
Value of a =3
Value of b =4
```

Access Modifiers in Java

- The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

- **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
- **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
- **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
- **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

NOTE: There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc.

Understanding Java Access Modifiers

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

1) Private: The private access modifier is accessible only within the class.
Example: In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}
public class Simple{
 public static void main(String args[]){
   A obj=new A();
   System.out.println(obj.data);//Compile Time Error
   obj.msg();//Compile Time Error
   }
}
```

2) Default: If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.
Example: In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
//save by A.java
package pack;
class A{
  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();//Compile Time Error
   obj.msg();//Compile Time Error
  }
}
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

3) Protected:

- 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.
- It provides more accessibility than the default modifier.

Example: In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
//save by A.java
package pack;
public class A{
protected void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;
class B extends A{
  public static void main(String args[]){
   B obj = new B();
   obj.msg();
  }
}
```

Output: Hello

4) Public:

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

Example:

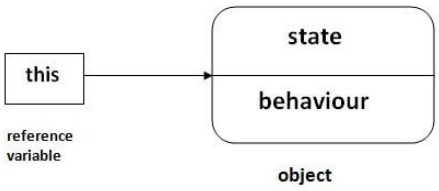
```
//save by A.java
package pack;
public class A{
public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

Output: Hello

LECTURE- 7

"this" keyword in java

There can be a lot of usage of java this keyword. In java, this is a reference variable that refers to the current object.



Usage of java this keyword: Here is given the 6 usage of java this keyword.

- this can be used to refer current class instance variable.
- this can be used to invoke current class method (implicitly)

- this() can be used to invoke current class constructor.
- this can be passed as an argument in the method call.
- this can be passed as argument in the constructor call.
- this can be used to return the current class instance from the method.

1) this: to refer current class instance variable

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

Example: Understanding the problem without this keyword

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno, String name, float fee){
rollno=rollno;
name=name;
fee=fee;
}
void display(){
System.out.println(rollno+" "+name+" "+fee);
}
}
class TestThis1{
public static void main(String args[]){
Student s1=new Student(111,"ABC",5000f);
Student s2=new Student(112,"XYZ",6000f);
s1.display();
s2.display();
}}
```

Output:

```
0 null 0.0
0 null 0.0
```

- In the above example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

Example: Solution of the above problem by this keyword

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}
class TestThis2{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
```

```
s2.display();
}}
```

Output:

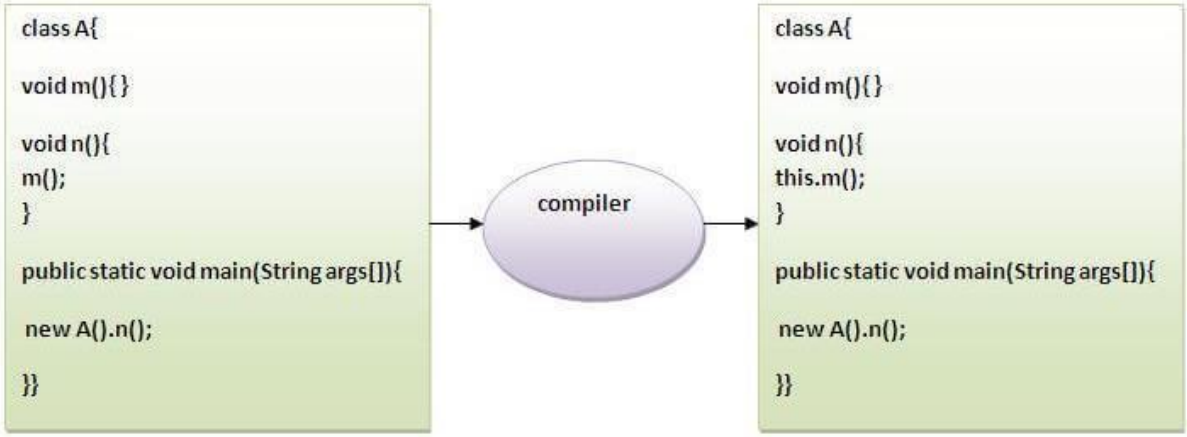
```
111 ABC 5000
112 XYZ 6000
```

- If local variables (formal arguments) and instance variables are different, there is no need to use this keyword like in the following program:

2) this: to invoke current class method

- We may invoke the method of the current class by using the this keyword. If we don't use the "this" keyword, compiler automatically adds this keyword while invoking the method.

Let's see the example



Example:

```
class A{
void m(){System.out.println("hello m");}
void n(){
System.out.println("hello n");
//m();//same as this.m()
this.m();
}
}
class TestThis4{
public static void main(String args[]){
A a=new A();
a.n();
}}
```

Output:

```
hello n
hello m
```

3) this() : to invoke current class constructor

- The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

Example: Calling default constructor from parameterized constructor:

```
class A{
A(){System.out.println("hello a");}
A(int x){
this();
System.out.println(x);
}
}
```

```
class TestThis5{
public static void main(String args[]){
A a=new A(10);
}}
```

Output:

```
hello a
10
```

Using this() in constructor overloading

- this() reference can be used during constructor overloading to call default constructor implicitly from parameterized constructor. “this()” should be the first statement inside a constructor.

Example: // Java program to illustrate role of this() in Constructor Overloading

```
class Box
{
   double width, height, depth;
   int boxNo;
   // constructor used when all dimensions and boxNo specified
   Box(double w, double h, double d, int num) {
      width = w;
      height = h;
      depth = d;
      boxNo = num;
   }
   // constructor used when no dimensions specified
   Box()
   {          width = height = depth = 0;     }
   // constructor used when only boxNo specified
   Box(int num)
   {
     // this() is used for calling the default constructor from parameterized constructor
      this();
      boxNo = num;
   }
   public static void main(String[] args)
   {
      // create box using only boxNo
      Box box1 = new Box(1);
      // getting initial width of box1
      System.out.println(box1.width);
   }
    }
```

Output: 0.0

4) this: to pass as an argument in the method

- The this keyword can also be passed as an argument in the method. It is mainly used in the event handling. Let's see the example:

```
class S2{
  void m(S2 obj){
  System.out.println("method is invoked");
  }
  void p(){
  m(this);
  }
```

```
  public static void main(String args[]){
  S2 s1 = new S2();
  s1.p();
  }
}
```

Output: method is invoked

Application of this that can be passed as an argument: In event handling (or) in a situation where we have to provide reference of a class to another one. It is used to reuse one object in many methods.

5) this: to pass as argument in the constructor call

- We can pass the "this" keyword in the constructor also. It is useful if we have to use one object in multiple classes.
- Example:

```
class B{
  A obj;
  B(A obj){      this.obj=obj;    }
  void display(){
    System.out.println(obj.data); //using data member of A class
  }
}
class A{
  int data=10;
  A(){
   B b=new B(this);
   b.display();
  }
  public static void main(String args[])
        {     A a=new A();    }
}
```

Output:10

6) this keyword can be used to return current class instance

- We can return this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive).
- Syntax of this that can be returned as a statement

```
return_type method_name(){  return this;  }
```

Example of this keyword that you return as a statement from the method

```
class A{
A getA(){  return this;  }
void msg(){System.out.println("Hello java");}
}
class Test1{
public static void main(String args[]){
new A().getA().msg();
}
}
```

Output:

Hello java

LECTURE-8
Java static keyword

- The static keyword in Java is used for memory management mainly. We can apply static keyword with variables, methods, blocks and nested classes. The static keyword belongs to the class than an instance of the class.
- The static can be:
  - Variable (also known as a class variable)
  - Method (also known as a class method)
  - Block
  - Nested class

1) Java static variable

- If you declare any variable as static, it is known as a static variable.
- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.
- Advantages of static variable: It makes the program memory efficient (i.e., it saves memory).

Understanding the problem without static variable

```
class Student{
    int rollno;
    String name;
    String college="ABIT";
}
```

- Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all objects. If we make it static, this field will get the memory only once.
- Java static property is shared to all objects.

Example of static variable

```
//Java Program to demonstrate the use of static variable
class Student{
  int rollno;//instance variable
  String name;
  static String college ="ABIT";//static variable
  //constructor
  Student(int r, String n)
  {    rollno = r;    name = n;    }
  //method to display the values
  void display (){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to show the values of objects
public class TestStaticVariable{
 public static void main(String args[]){
 Student s1 = new Student(111,"ABC");
 Student s2 = new Student(222,"XYZ");
 //we can change the college of all objects by the single line of code Student.college="IT"
 s1.display();
 s2.display();
 }
}
```

Output:

```
111    ABC    ABIT
222    XYZ    ABIT
```

Program of the counter 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 other objects. So each object will have the value 1 in the count variable.

```
/*Java Program to demonstrate the use of an instance variable which get memory each time
when we create an object of the class.  */
class Counter{
int count=0;//will get memory each time when the instance is created
Counter(){
count++;//incrementing value
System.out.println(count);
}
public static void main(String args[]){
//Creating objects
Counter c1=new Counter();
Counter c2=new Counter();
Counter c3=new Counter();
}
}
```

Output:

```
1
1
1
```

Program of counter by static variable

- We know that static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

```
//Java Program to illustrate the use of static variable which is shared with all objects.
class Counter2{
static int count=0;//will get memory only once and retain its value
Counter2(){
count++;//incrementing the value of static variable
System.out.println(count);
}
public static void main(String args[]){
//creating objects
Counter2 c1=new Counter2();
Counter2 c2=new Counter2();
Counter2 c3=new Counter2();
}
}
```

Output:

```
1
2
3
```

2) Java 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 the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

Example: //Java Program to get the cube of a given number using the static method

```
class Calculate{
static int cube(int x){
return x*x*x;
}
public static void main(String args[]){
int result=Calculate.cube(5);
System.out.println(result);
}
}
```

Output:125

Restrictions for the static method

There are two main restrictions for the static method. They are:

- The static method can not use non static data member or call non-static method directly.
- "this" and "super" cannot be used in static context.

```
class A{
 int a=40;//non static
   public static void main(String args[]){
  System.out.println(a);
 }
}
```

Output: Compile Time Error

Q) Why is the Java main method static?

Ans) It is because the object is not required to call a static method. If it were a non-static method, JVM creates an object first then call main() method that will lead the problem of extra memory allocation.

3) Java static block

- Is used to initialize the static data member.
- It is executed before the main method at the time of classloading.

Example of static block

```
class A2{
  static{System.out.println("static block is invoked");}
  public static void main(String args[]){
   System.out.println("Hello main");
  }
}
```

Output: static block is invoked
Hello main

CHAPTER-2
LECTURE - 9
Inheritance in Java

- **Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).
- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.
- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

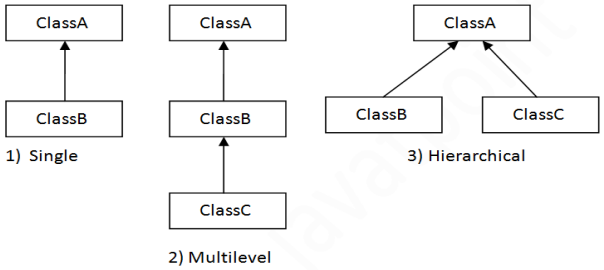
The syntax of Java Inheritance

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

- The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.
- In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

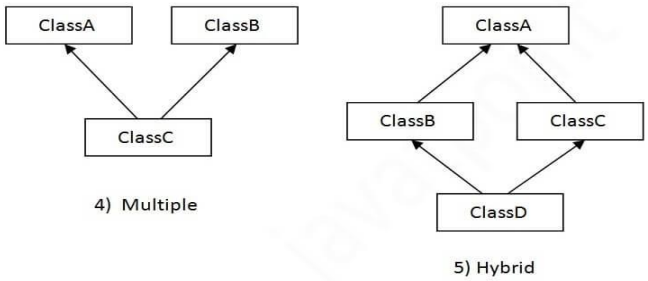
Types of inheritance in java

- On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.
  - In java programming, multiple and hybrid inheritance is supported through interface only.



Note: Multiple inheritance is not supported in Java through class.

- When one class inherits multiple classes, it is known as multiple inheritance. For Example:



1. Single Inheritance Example

- When a class inherits another class, it is known as a *single inheritance*.

```
Class A
{
  public void methodA()
  {     System.out.println("Base class method");   }
}
Class B extends A
{
  public void methodB()
  {     System.out.println("Child class method");   }
  public static void main(String args[])
  {
    B obj = new B();
    obj.methodA(); //calling super class method
    obj.methodB(); //calling local method
  }
}
```

OUTPUT:

```
Base class method
Child class method
```

2. Multilevel Inheritance Example

When there is a chain of inheritance, it is known as *multilevel inheritance*.

```
Class X
{
  public void methodX()
  {     System.out.println("Class X method");   }
}
Class Y extends X
{
public void methodY()
{         System.out.println("class Y method");          }
}
Class Z extends Y
{
  public void methodZ()
  {     System.out.println("class Z method");   }
  public static void main(String args[])
  {
    Z obj = new Z();
    obj.methodX(); //calling grand parent class method
    obj.methodY(); //calling parent class method
    obj.methodZ(); //calling local method
  }
}
```

OUTPUT:

```
Class X method
Class Y method
Class Z method
```

3. Hierarchical Inheritance Example

- When two or more classes inherits a single class, it is known as hierarchical inheritance.

```
class A
{
  public void methodA()
  {     System.out.println("method of Class A");   }
}
class B extends A
{
  public void methodB()
  {     System.out.println("method of Class B");   }
}
class C extends A
{
 public void methodC()
 {     System.out.println("method of Class C");  }
}
class D extends A
{
 public void methodD()
 {     System.out.println("method of Class D");  }
}
class JavaExample
{
 public static void main(String args[])
 {
   B obj1 = new B();
   C obj2 = new C();
   D obj3 = new D();
   //All classes can access the method of class A
   obj1.methodA();
   obj2.methodA();
   obj3.methodA();
 }
}
```

Output:

```
method of Class A
method of Class A
method of Class A
```

Q) Why multiple inheritance is not supported in java?

- To reduce the complexity and simplify the language, multiple inheritance is not supported in java.
- Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.
- Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
```

```
}
class C extends A, B{
 public static void main(String args[]){
   C obj=new C();
   obj.msg();//Now which msg() method would be invoked?
}
}
```

OUTPUT: Compile Time Error

LECTURE-9

Inheritance and constructors in Java

In Java, constructor of base class with no argument gets automatically called in derived class constructor.

Example:

```
class Base {
  Base() {
    System.out.println("Base Class Constructor Called ");
  }
}
class Derived extends Base {
  Derived() {
    System.out.println("Derived Class Constructor Called ");
  }
}
public class Main {
  public static void main(String[] args) {
    Derived d = new Derived();
  }
}
```

Output:

```
Base Class Constructor Called
Derived Class Constructor Called
```

Super Keyword in Java

- The super keyword in java is a reference variable that is used to refer parent class objects. The keyword "super" came into the picture with the concept of Inheritance. It is majorly used in the following contexts:

1. Use of super with variables:

This scenario occurs when a derived class and base class has same data members. In that case there is a possibility of ambiguity for the JVM. We can understand it more clearly using this code snippet:

```
/* Base class vehicle */
class Vehicle
{     int maxSpeed = 120;    }
 /* sub class Car extending vehicle */
class Car extends Vehicle
{
   int maxSpeed = 180;
   void display()
   {
      /* print maxSpeed of base class (vehicle) */
      System.out.println("Maximum Speed: " + super.maxSpeed);
   }
}
```

```
/* Driver program to test */
class Test
{
   public static void main(String[] args)
   {
      Car small = new Car();
      small.display();
   }
}
```

Output: Maximum Speed: 120

- In the above example, both base class and subclass have a member maxSpeed. We could access maxSpeed of base class in subclass using super keyword.

2. Use of super with methods:

This is used when we want to call parent class method. So whenever a parent and child class have same named methods then to resolve ambiguity we use super keyword. This code snippet helps to understand the said usage of super keyword.

```
/* Base class Person */
class Person
{
   void message()
   {        System.out.println("This is person class");     }
}
/* Subclass Student */
class Student extends Person
{
   void message()
   {       System.out.println("This is student class");     }
   // Note that display() is only in Student class
   void display()
   {
      // will invoke or call current class message() method
      message();
      // will invoke or call parent class message() method
      super.message();
   }
}
/* Driver program to test */
class Test
{
   public static void main(String args[])
   {
      Student s = new Student();
     // calling display() of Student
      s.display();
   }
}
```

Output:

```
This is student class
This is person class
```

- In the above example, we have seen that if we only call method message() then, the current class message() is invoked but with the use of super keyword, message() of superclass could also be invoked.

3. Use of super with constructors:

super keyword can also be used to access the parent class constructor. One more important thing is that, 'super' can call both parametric as well as non parametric constructors depending upon the situation. Following is the code snippet to explain the above concept:

```
/* superclass Person */
class Person
{
   Person()
   {
      System.out.println("Person class Constructor");
   }
}
/* subclass Student extending the Person class */
class Student extends Person
{
   Student()
   {
      // invoke or call parent class constructor
      super();
      System.out.println("Student class Constructor");
   }
}
/* Driver program to test*/
class Test
{
   public static void main(String[] args)
   {
      Student s = new Student();
   }
}
```

Output:

```
Person class Constructor
Student class Constructor
```

In the above example we have called the superclass constructor using keyword 'super' via subclass constructor.

Other Important points:

- Call to super() must be first statement in Derived(Student) Class constructor.
- If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the superclass does not have a no-argument constructor, you will get a compile-time error. Object *does* have such a constructor, so if Object is the only superclass, there is no problem.
- If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that a whole chain of constructors called, all the way back to the constructor of Object. This, in fact, is the case. It is called *constructor chaining*.
- The super keyword in Java is a reference variable which is used to refer immediate parent class object.

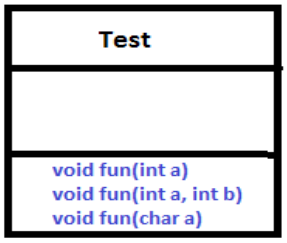
- Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

LECTURE-10
Polymorphism in Java

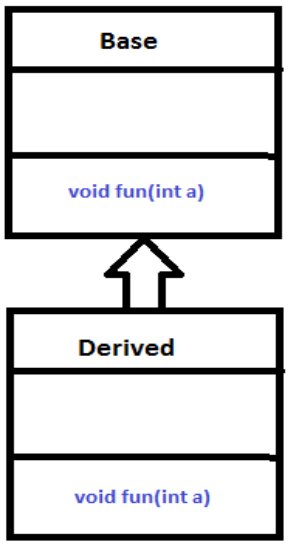
- **Polymorphism in Java** is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
- Real life example of polymorphism: A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person possess different behaviour in different situations. This is called polymorphism.
- Polymorphism is considered as one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations.
- **In Java polymorphism is mainly divided into two types:**
    - Compile time Polymorph0ism (method overloading and operator overloading)
    - Runtime Polymorphism (method overriding)

1. **Compile time polymorphism**: It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading.
2. **Runtime polymorphism**: It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding.

EXAMPLE:



Overloading



Overriding

<u>i. Method Overloading</u>: When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by **change in number of arguments** or/and **change in type of arguments**.

Example: By using different types of arguments

```
// Java program for Method overloading
class MultiplyFun {
   // Method with 2 parameter
   static int Multiply(int a, int b)
   {        return a * b;     }
  // Method with the same name but 2 double parameter
```

```
    static double Multiply(double a, double b)
    {       return a * b;     }
}
 class Main {
    public static void main(String[ ] args)
    {
        System.out.println(MultiplyFun.Multiply(2, 4));
        System.out.println(MultiplyFun.Multiply(5.5, 6.3));
    }
}
```

Output:

```
8
34.65
```

ii. Operator Overloading: Java also provide option to overload operators. For example, we can make the operator ('+') for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So a single operator '+' when placed between integer operands, adds them and when placed between string operands, concatenates them.

- In java, Only "+" operator can be overloaded:
- To add integers
- To concatenate strings

Example:

```
// Java program for Operator overloading
class OperatorOVERDDN {
    void operator(String str1, String str2)
    {
        String s = str1 + str2;
        System.out.println("Concatinated String - " + s);
    }
    void operator(int a, int b)
    {
        int c = a + b;
        System.out.println("Sum = " + c);
    }
}
class Main {
    public static void main(String[] args)
    {
        OperatorOVERDDN obj = new OperatorOVERDDN();
        obj.operator(2, 3);
        obj.operator("java ", "program");
    }
}
```

Output:

Sum = 5

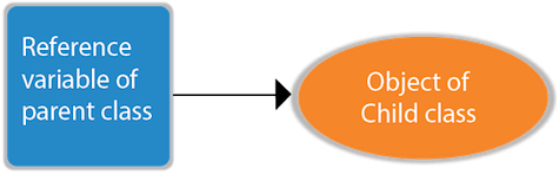
Concatinated String – java program

Runtime Polymorphism in Java

- **Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

- In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Upcasting: If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:




```
class A{}
class B extends A{}
A a=new B();//upcasting
```

For upcasting, we can use the reference variable of class type or an interface type. For Example:

```
interface I{}
class A{}
class B extends A implements I{}
```

Here, the relationship of B class would be:

```
B IS-A A
B IS-A I
B IS-A Object
```

Since Object is the root class of all classes in Java, so we can write B IS-A Object.

Example of Java Runtime Polymorphism

- In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.
- Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

```
class Bike{
  void run(){System.out.println("running");}
}
class Splendor extends Bike{
  void run(){System.out.println("running safely with 60km");}

  public static void main(String args[]){
    Bike b = new Splendor();//upcasting
    b.run();
  }
}
```

Output: running safely with 60km.

- **Method overriding** occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

Example:

```
// Java program for Method overridding
class Parent {
    void Print()
  {     System.out.println("parent class");    }
}
```

```
class subclass1 extends Parent {
   void Print()
   {
      System.out.println("subclass1");
   }
}
class subclass2 extends Parent {
   void Print()
   {        System.out.println("subclass2");     }
}
class TestPolymorphism3 {
   public static void main(String[] args)
   {
     Parent a;
      a = new subclass1();
      a.Print();
      a = new subclass2();
      a.Print();
   }
}
```

Output:

```
subclass1
subclass2
```

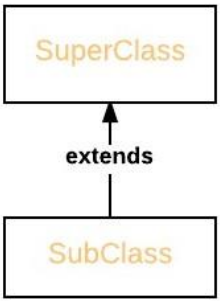
LECTURE-11

Dynamic Method Dispatch or Runtime Polymorphism in Java

- Method overriding is one of the ways in which Java supports Runtime Polymorphism. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.
- At run-time, it depends on 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
- A superclass reference variable can refer to a subclass object. This is also known as upcasting. Java uses this fact to resolve calls to overridden methods at run time.

Upcasting

SuperClass obj = new SubClass



- Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed. Here is an example that illustrates dynamic method dispatch:

Example: // A Java program to illustrate Dynamic Method Dispatch using hierarchical inheritance

```
class A
```

```
{
   void m1()
   {        System.out.println("Inside A's m1 method");     }
}
class B extends A
{
   // overriding m1()
   void m1()
   {        System.out.println("Inside B's m1 method");     }
}
class C extends A
{
   // overriding m1()
   void m1()
   {       System.out.println("Inside C's m1 method");     }
}
 // Driver class
class Dispatch
{
   public static void main(String args[])
   {
      // object of type A
      A a = new A();
     // object of type B
      B b = new B();
     // object of type C
      C c = new C();
     // obtain a reference of type A
      A ref;
     // ref refers to an A object
      ref = a;
     // calling A's version of m1()
      ref.m1();
     // now ref refers to a B object
      ref = b;
      // calling B's version of m1()
      ref.m1();
      // now ref refers to a C object
      ref = c;
      // calling C's version of m1()
      ref.m1();
   }
     }
```

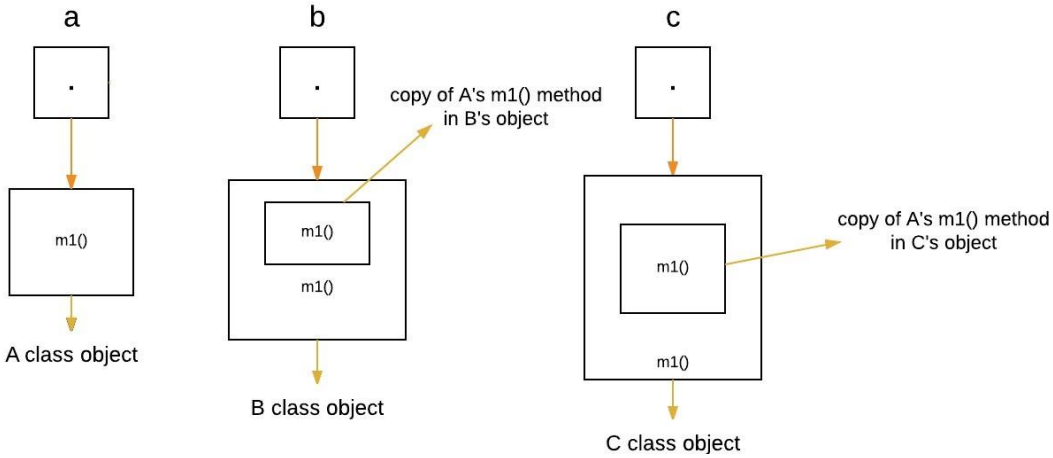
Output:

```
Inside A's m1 method
Inside B's m1 method
Inside C's m1 method
```

Explanation:

- The above program creates one superclass called A and it’s two subclasses B and C. These subclasses overrides m1( ) method.
- Inside the main() method in Dispatch class, initially objects of type A, B, and C are declared.

A a = new A(); // object of type A
B b = new B(); // object of type B
C c = new C(); // object of type C

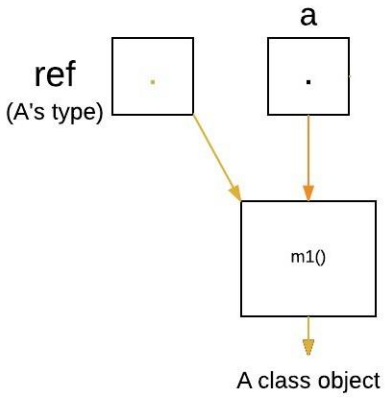


Now a reference of type A, called ref, is also declared, initially it will point to null.

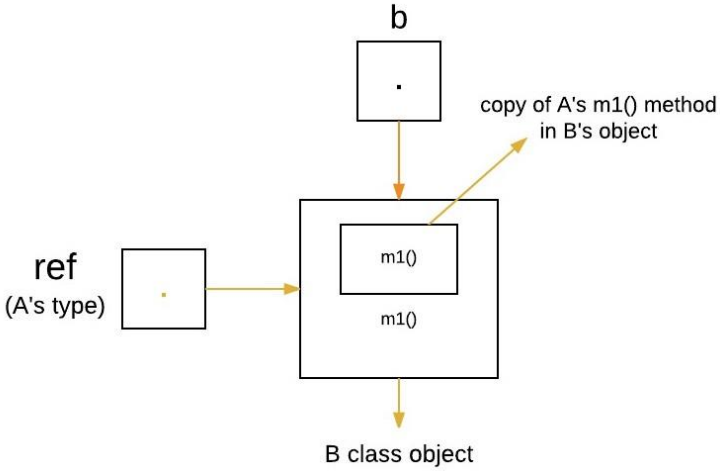
A ref; // obtain a reference of type A

Now we are assigning a reference to each **type of object** (either A's or B's or C's) to *ref*, one-by-one, and uses that reference to invoke m1( ). As the output shows, the version of m1( ) executed is determined **by the type of object being referred to at the time of the call.**

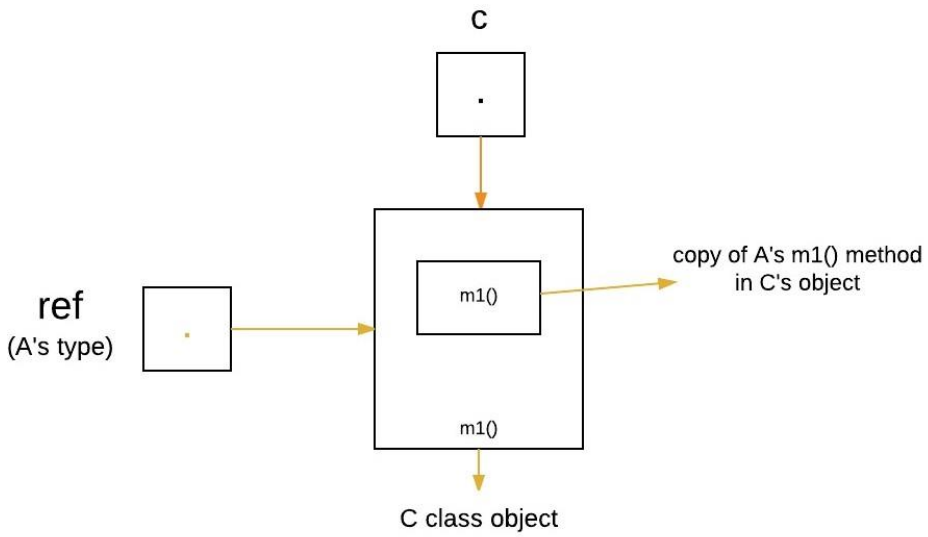
ref = a; // r refers to an A object
ref.m1(); // calling A's version of m1()



ref = b; // now r refers to a B object
ref.m1(); // calling B's version of m1()



ref = c; // now r refers to a C object
ref.m1(); // calling C's version of m1()



Runtime Polymorphism with Data Members

- In Java, we can override methods only, not the variables(data members), so **runtime polymorphism cannot be achieved by data members.**

Example: /* Java program to illustrate the fact that runtime polymorphism cannot be achieved by data members */

```
// class A
class A
{     int x = 10; }

// class B
class B extends A
{     int x = 20; }
// Driver class
public class Test
{
   public static void main(String args[])
   {
      A a = new B(); // object of type B
      // Data member of class A will be accessed
      System.out.println(a.x);
   }
     }
```

Output: 10

Explanation :

- In above program, both the class A(super class) and B(sub class) have a common variable 'x'. Now we make object of class B, referred by 'a' which is of type of class A. Since variables are not overridden, so the statement "a.x" will **always** refer to data member of super class.

<u>Advantages of Dynamic Method Dispatch</u>

- Dynamic method dispatch allow Java to support overriding of methods which is central for run-time polymorphism.
- It allows a 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.
- It also allow subclasses to add its specific methods subclasses to define the specific implementation of some.

CHAPTER-3
STRING MANIPULATION
LECTURE-12
Java - Strings Class

- Strings, which are widely used in Java programming, are a sequence of characters.
- In Java programming language, strings are treated as objects.
- The Java platform provides the String class to create and manipulate strings.

Creating Strings

- The most direct way to create a string is to write −
  String greeting = "Hello world!";
- Whenever it encounters a string literal in your code, the compiler creates a String object with its value in this case, "Hello world!'.
- As with any other object, you can create String objects by using the new keyword and a constructor. The String class has 11 constructors that allow you to provide the initial value of the string using different sources, such as an array of characters.

Example

```
public class StringDemo {
   public static void main(String args[]) {
      char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };
      String helloString = new String(helloArray);
      System.out.println( helloString );
   }
}
```

Output: hello.

Note − The String class is immutable, so that once it is created a String object cannot be changed. If there is a necessity to make a lot of modifications to Strings of characters, then we should use String Buffer & String Builder Classes.

String Length:

Methods used to obtain information about an object are known as **accessor methods**. One accessor method that you can use with strings is the **length()** method, which returns the number of characters contained in the string object.

Example:

```
public class StringDemo {
   public static void main(String args[]) {
      String palindrome = "Dot saw I was Tod";
      int len = palindrome.length();
      System.out.println( "String Length is : " + len );
   }
}
```

Output: String Length is : 17

Concatenating Strings

- The String class includes a method for concatenating two strings −
  `string1.concat(string2);`
- This returns a new string that is string1 with string2 added to it at the end. You can also use the concat() method with string literals, as in − "My name is ".concat("ABC");
- Strings are more commonly concatenated with the + operator, as in −
  "Hello," + " world" + "!"
- which results in − "Hello, world!"

Example:

```
public class StringDemo {
   public static void main(String args[]) {
      String string1 = "JAVA";
```

```
        String string2 = "PROGRAMMING";
        System.out.println("string1 + string2");
        }
    }
```

Output: JAVA PROGRAMMING

Creating Format Strings

- You have printf() and format() methods to print output with formatted numbers. The String class has an equivalent class method, format(), that returns a String object rather than a PrintStream object.
- Using String's static format() method allows you to create a formatted string that you can reuse, as opposed to a one-time print statement.
- For example, instead of −

```
System.out.printf("The value of the float variable is " +
        "%f, while the value of the integer " +
        "variable is %d, and the string " +
        "is %s", floatVar, intVar, stringVar);
```

- We can write −

```
String fs;
fs = String.format("The value of the float variable is " +
        "%f, while the value of the integer " +
        "variable is %d, and the string " +
        "is %s", floatVar, intVar, stringVar);
System.out.println(fs);
```

String Methods:

Java - String charAt() Method

- This method returns the character located at the String's specified index. The string indexes start from zero.
- **Syntax:** public char charAt(int index)
- **Parameters:** Here is the detail of parameters −
  index − Index of the character to be returned.
- **Return Value:** This method returns a char at the specified index.

Example

```
public class Test {
  public static void main(String args[]) {
    String s = "Strings are immutable";
    char result = s.charAt(8);
    System.out.println(result);
  }
}
```

Output: a

Java - String compareTo() Method

- This method compares this String to another Object.
- **Syntax:** int compareTo(Object o)
- **Parameters:** O − the Object to be compared.
- **Return Value:** The value 0 if the argument is a string lexicographically equal to this string; a value less than 0 if the argument is a string lexicographically greater than this string; and a value greater than 0 if the argument is a string lexicographically less than this string.

Example

```
public class Test {
  public static void main(String args[]) {
    String str1 = "Strings are immutable";
    String str2 = new String("Strings are immutable");
    String str3 = new String("Integers are not immutable");
    int result = str1.compareTo( str2 );
    System.out.println(result);
    result = str2.compareTo( str3 );
    System.out.println(result);
  }
}
```

Output: 0
10

String compareTo(String anotherString)

- This method compares two strings lexicographically.
- **Syntax:** int compareTo(String anotherString)
- **Parameters:** anotherString − the String to be compared.
- **Return Value:** The value 0 if the argument is a string lexicographically equal to this string; a value less than 0 if the argument is a string lexicographically greater than this string; and a value greater than 0 if the argument is a string lexicographically less than this string.

Example:

```
public class Test {
  public static void main(String args[]) {
    String str1 = "Strings are immutable";
    String str2 = "Strings are immutable";
    String str3 = "Integers are not immutable";
    int result = str1.compareTo( str2 );
    System.out.println(result);
    result = str2.compareTo( str3 );
    System.out.println(result);
    result = str3.compareTo( str1 );
    System.out.println(result);
  }
}
```

Output: 0
10
-10

Java - String compareToIgnoreCase() Method

- This method compares two strings lexicographically, ignoring case differences.
- **Syntax:** int compareToIgnoreCase(String str)
- **Parameters:** str − the String to be compared.
- **Return Value:** This method returns a negative integer, zero, or a positive integer as the specified String is greater than, equal to, or less than this String, ignoring case considerations.

Example

```
public class Test {
  public static void main(String args[]) {
    String str1 = "Strings are immutable";
    String str2 = "Strings are immutable";
    String str3 = "Integers are not immutable";
    int result = str1.compareToIgnoreCase( str2 );
    System.out.println(result);
```

```
        result = str2.compareToIgnoreCase( str3 );
        System.out.println(result);
        result = str3.compareToIgnoreCase( str1 );
        System.out.println(result);
      }
    }
```

Output:

```
0
10
-10
```

Java - String concat() Method

- This method appends one String to the end of another. The method returns a String with the value of the String passed into the method, appended to the end of the String, used to invoke this method.
- **Syntax:** public String concat(String s)
- **Parameters:** s − the String that is concatenated to the end of this String.
- **Return Value:** This method returns a string that represents the concatenation of this object's characters followed by the string argument's characters.

Example

```
public class Test {
  public static void main(String args[]) {
    String s = "Strings are immutable";
    s = s.concat(" all the time");
    System.out.println(s);
  }
}
```

Output: Strings are immutable all the time

Java - String contentEquals() Method

- This method returns true if and only if this String represents the same sequence of characters as specified in StringBuffer.
- **Syntax:** public boolean contentEquals(StringBuffer sb)
- **Parameters:** sb − the StringBuffer to compare.
- **Return Value:** This method returns true if and only if this String represents the same sequence of characters as the specified in StringBuffer, otherwise false.

Example:

```
public class Test {
  public static void main(String args[]) {
    String str1 = "Not immutable";
    String str2 = "Strings are immutable";
    StringBuffer str3 = new StringBuffer( "Not immutable");
    boolean  result = str1.contentEquals( str3 );
    System.out.println(result);
    result = str2.contentEquals( str3 );
    System.out.println(result);
  }
}
```

Output

```
true
false
```

Java - String copyValueOf() Method

- This method returns a String that represents the character sequence in the array specified.
- **Syntax:** public static String copyValueOf(char[] data)
- **Parameters:** data − the character array.

- **Return Value:** This method returns a String that contains the characters of the character array.

Example:
```
public class Test {
  public static void main(String args[]) {
    char[] Str1 = {'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd'};
    String Str2 = "";
    Str2 = Str2.copyValueOf( Str1 );
    System.out.println("Returned String: " + Str2);
  }
}
```

Output: Returned String: hello world

Java - String copyValueOf(data, offset, count)

- This returns a String that represents the character sequence in the array specified.
- **Syntax:** public static String copyValueOf(char[] data, int offset, int count)
- **Parameters:**
  data − the character array.
  offset − initial offset of the subarray.
  count − length of the subarray.
- **Return Value:** This method returns a String that contains the characters of the character array.

Example:
```
public class Test {
  public static void main(String args[]) {
    char[] Str1 = {'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd'};
    String Str2 = "";
    Str2 = Str2.copyValueOf( Str1, 2, 6 );
    System.out.println("Returned String: " + Str2);
  }
}
```

Output: Returned String: llo wo

Java - String endsWith() Method

- This method tests if this string ends with the specified suffix.
- **Syntax:** public boolean endsWith(String suffix)
- **Parameters:** suffix − the suffix.
- **Return Value:** This method returns true if the character sequence represented by the argument is a suffix of the character sequence represented by this object; false otherwise. Note that the result will be true if the argument is the empty string or is equal to this String object as determined by the equals (Object) method.

Example
```
public class Test {
  public static void main(String args[]) {
    String Str = new String("This is really not immutable!!");
    boolean retVal;
    retVal = Str.endsWith( "immutable!!" );
    System.out.println("Returned Value = " + retVal );
    retVal = Str.endsWith( "immu" );
    System.out.println("Returned Value = " + retVal );
  }
}
```

Output:
```
Returned Value = true
Returned Value = false
```

Java - String equals() Method

- This method compares this string to the specified object. The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as this object.
- **Syntax:** public boolean equals(Object anObject)
- **Parameters:** anObject − the object to compare this String against.
- **Return Value:** This method returns true if the String are equal; false otherwise.

Example

```
public class Test {
  public static void main(String args[]) {
    String Str1 = new String("This is really not immutable!!");
    String Str2 = Str1;
    String Str3 = new String("This is really not immutable!!");
    boolean retVal;
    retVal = Str1.equals( Str2 );
    System.out.println("Returned Value = " + retVal );
    retVal = Str1.equals( Str3 );
    System.out.println("Returned Value = " + retVal );
  }
}
```

Output:

```
Returned Value = true
Returned Value = true
```

Java - String equalsIgnoreCase() Method

- This method compares this String to another String, ignoring case considerations. Two strings are considered equal ignoring case, if they are of the same length, and corresponding characters in the two strings are equal ignoring case.
- **Syntax:** public boolean equalsIgnoreCase(String anotherString)
- **Parameters:** anotherString − the String to compare this String against.
- **Return Value:** This method returns true if the argument is not null and the Strings are equal, ignoring case; false otherwise.

Example:

```
public class Test {
  public static void main(String args[]) {
    String Str1 = new String("This is really not immutable!!");
    String Str2 = Str1;
    String Str3 = new String("This is really not immutable!!");
    String Str4 = new String("This IS REALLY NOT IMMUTABLE!!");
    boolean retVal;
    retVal = Str1.equals( Str2 );
    System.out.println("Returned Value = " + retVal );
    retVal = Str1.equals( Str3 );
    System.out.println("Returned Value = " + retVal );
    retVal = Str1.equalsIgnoreCase( Str4 );
    System.out.println("Returned Value = " + retVal );
  }
}
```

Output:

```
Returned Value = true
Returned Value = true
Returned Value = true
```

Java – String indexOf() Method

- **Description:** This method returns the index within this string of the first occurrence of the specified character or -1, if the character does not occur.
- **Syntax:** public int indexOf(char ch)

- **Parameter:** ch − a character.

Example:
```
public class Test {
   public static void main(String args[]) {
      String Str = new String("Welcome to Java programming");
      System.out.print("Found Index :" );
      System.out.println(Str.indexOf( 'o' ));
   }
}
```
Output: Found Index :4

Java - String indexOf(int ch, int fromIndex)

- This method returns the index within this string of the first occurrence of the specified character, starting the search at the specified index or -1, if the character does not occur.
- **Syntax:** public in indexOf(char ch, int fromIndex)
- **Parameters:** Here is the detail of parameters −
  ch − a character.
  fromIndex − the index to start the search from.

Example:
```
import java.io.*;
public class Test {
   public static void main(String args[]) {
      String Str = new String("Welcome to Java programming");
      System.out.print("Found Index :" );
      System.out.println(Str.indexOf( 'o', 5 ));
   }
}
```
Output: Found Index :9

Java – String indexOf(String str) Method

- **Description:** This method returns the index within this string of the first occurrence of the specified substring. If it does not occur as a substring, -1 is returned.
- **Syntax:** int indexOf(String str)
- **Parameters:** str − a string.

Example:
```
import java.io.*;
public class Test {
   public static void main(String args[]) {
      String Str = new String("Welcome to Java programming");
      String SubStr1 = new String("to");
      System.out.println("Found Index :" + Str.indexOf( SubStr1 ));
   }
}
```
Output: Found Index :8

Java – String lastIndexOf() Method

- **Description:** This method returns the index of the last occurrence of the character in the character sequence represented by this object that is less than or equal to fromIndex, or -1 if the character does not occur before that point.
- **Syntax**: int lastIndexOf(int ch)
- **Parameters:** ch − a character.
- **Return Value:** This method returns the index.

Example:
```
import java.io.*;
public class Test {
   public static void main(String args[]) {
      String Str = new String("Welcome to Java programming");
      System.out.print("Found Last Index :" );
```

```
        System.out.println(Str.lastIndexOf( 'o' ));
      }
    }
```

Output: Found Last Index :18

Java - String length() Method

- **Description:** This method returns the length of this string. The length is equal to the number of 16-bit Unicode characters in the string.
- **Syntax:** public int length()
- **Return Value:** This method returns the the length of the sequence of characters represented by this object.

Example:

```
import java.io.*;
public class Test {
  public static void main(String args[]) {
    String Str1 = new String("Welcome to Java programming");
    String Str2 = new String("Tutorials" );
    System.out.print("String Length :" );
    System.out.println(Str1.length());
    System.out.print("String Length :" );
    System.out.println(Str2.length());
  }
}
```

Output:

```
String Length :27
String Length :9
```

Java - String startsWith() Method

- **Description:** This method has two variants and tests if a string starts with the specified prefix beginning a specified index or by default at the beginning.
- **Syntax:** public boolean startsWith (String prefix)
- **Parameters**: prefix − the prefix to be matched.
- **Return Value:** It returns true if the character sequence represented by the argument is a prefix of the character sequence represented by this string; false otherwise.

Example:

```
public class Test {
  public static void main(String args[]) {
    String Str = new String("Welcome to Java programming");
    System.out.print("Return Value :" );
    System.out.println(Str.startsWith("Welcome") );
    System.out.print("Return Value :" );
    System.out.println(Str.startsWith("Tutorials") );
  }
}
```

Output:

```
Return Value :true
Return Value :false
```

Java - String substring() Method

- **Description:** This method has two variants and returns a new string that is a substring of this string. The substring begins with the character at the specified index and extends to the end of this string or up to endIndex – 1, if the second argument is given.
- **Syntax:** public String substring(int beginIndex)
- **Parameters:** beginIndex − the begin index, inclusive.
- **Return Value:** The specified substring.

Example:

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

```
        String Str = new String("Welcome to Java programming");
        System.out.print("Return Value :" );
        System.out.println(Str.substring(10) );
    }
}
```

Output: Return Value : Java programming

Java - String toLowerCase() Method

- **Description:** This method has two variants. The first variant converts all of the characters in this String to lower case using the rules of the given Locale. This is equivalent to calling toLowerCase(Locale.getDefault()). The second variant takes locale as an argument to be used while converting into lower case.
- **Syntax:** public String toLowerCase()
- **Return Value:** It returns the String, converted to lowercase.

Example:

```
import java.io.*;
public class Test {
    public static void main(String args[]) {
        String Str = new String("Welcome to Java programming");
        System.out.print("Return Value :");
        System.out.println(Str.toLowerCase());
    }
}
```

Output: Return Value :welcome to java programming

Java - String toUpperCase() Method

- **Description:** This method has two variants. The first variant converts all of the characters in this String to upper case using the rules of the given Locale. This is equivalent to calling toUpperCase(Locale.getDefault()). The second variant takes locale as an argument to be used while converting into upper case.
- **Syntax:** public String toUpperCase()
- **Return Value:** It returns the String, converted to uppercase.

Example:

```
import java.io.*;
public class Test {
    public static void main(String args[]) {
        String Str = new String("Welcome to Java programming");
        System.out.print("Return Value :" );
        System.out.println(Str.toUpperCase() );
    }
}
```

Output: Return Value : WELCOME TO JAVA PROGRAMMING

Java - String trim() Method

- This method returns a copy of the string, with leading and trailing whitespace omitted.
- **Syntax:** public String trim()
- **Return Value:** It returns a copy of this string with leading and trailing white space removed, or this string if it has no leading or trailing white space.

Example:

```
import java.io.*;
public class Test {
    public static void main(String args[]) {
        String Str = new String("   Welcome to Java programming   ");
        System.out.print("Return Value :" );
        System.out.println(Str.trim() );
    }
}
```

Output: Return Value :Welcome to Java programming

LECTURE - 13

String Buffer and String Builder Classes

- The **StringBuffer** and **StringBuilder** classes are used when there is a necessity to make a lot of modifications to Strings of characters.
- Unlike Strings, objects of type StringBuffer and String builder can be modified over and over again without leaving behind a lot of new unused objects.
- The StringBuilder class was introduced as of Java 5 and the main difference between the StringBuffer and StringBuilder is that StringBuilders methods are not thread safe (not synchronised).
- It is recommended to use **StringBuilder** whenever possible because it is faster than StringBuffer. However, if the thread safety is necessary, the best option is StringBuffer objects.

Example:
```
public class Test {
  public static void main(String args[]) {
    StringBuffer sBuffer = new StringBuffer("test");
    sBuffer.append(" String Buffer");
    System.out.println(sBuffer);
  }
}
```

Output: test String Buffer

Java StringBuffer class

- Java StringBuffer class is used to create mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.
- Note: Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.

Important methods of StringBuffer class

1) StringBuffer append() method

- The append() method concatenates the given argument with this string.

```
class StringBufferExample{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello ");
sb.append("Java");//now original string is changed
System.out.println(sb);//prints Hello Java
}
}
```

2) StringBuffer insert() method

- The insert() method inserts the given string with this string at the given position.

```
class StringBufferExample2{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello ");
sb.insert(1,"Java");//now original string is changed
System.out.println(sb);//prints HJavaello
}
}
```

3) StringBuffer replace() method

- The replace() method replaces the given string from the specified beginIndex and endIndex.

```
class StringBufferExample3{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
```

```
sb.replace(1,3,"Java");
System.out.println(sb);//prints HJavalo
}
}
```

4) StringBuffer delete() method

- This method of StringBuffer class deletes the string from the specified beginIndex to endIndex.

```
class StringBufferExample4{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.delete(1,3);
System.out.println(sb);//prints Hlo
}
}
```

5) StringBuffer reverse() method

- The reverse() method of StringBuilder class reverses the current string.

```
class StringBufferExample5{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.reverse();
System.out.println(sb);//prints olleH
}
}
```

6) StringBuffer capacity() method

- The capacity() method of StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity*2)+2.
- For example if your current capacity is 16, it will be (16*2)+2=34.

```
class StringBufferExample6{
public static void main(String args[]){
StringBuffer sb=new StringBuffer();
System.out.println(sb.capacity());//default 16
sb.append("Hello");
System.out.println(sb.capacity());//now 16
sb.append("java is my favourite language");
System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
}
}
```

7) StringBuffer ensureCapacity() method

- The ensureCapacity() method of StringBuffer class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

```
class StringBufferExample7{
public static void main(String args[]){
StringBuffer sb=new StringBuffer();
System.out.println(sb.capacity());//default 16
sb.append("Hello");
System.out.println(sb.capacity());//now 16
sb.append("java is my favourite language");
System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
sb.ensureCapacity(10);//now no change
```

```
System.out.println(sb.capacity());//now 34
sb.ensureCapacity(50);//now (34*2)+2
System.out.println(sb.capacity());//now 70
}
}
```

Java StringBuilder class

- Java StringBuilder class is used to create mutable (modifiable) string. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK 1.5.

Java StringBuilder Examples

1) StringBuilder append() method

- The StringBuilder append() method concatenates the given argument with this string.

```
class StringBuilderExample{
public static void main(String args[]){
StringBuilder sb=new StringBuilder("Hello ");
sb.append("Java");//now original string is changed
System.out.println(sb);//prints Hello Java
}
}
```

2) StringBuilder insert() method

- The StringBuilder insert() method inserts the given string with this string at the given position.

```
class StringBuilderExample2{
public static void main(String args[]){
StringBuilder sb=new StringBuilder("Hello ");
sb.insert(1,"Java");//now original string is changed
System.out.println(sb);//prints HJavaello
}
}
```

3) StringBuilder replace() method

- The StringBuilder replace() method replaces the given string from the specified beginIndex and endIndex.

```
class StringBuilderExample3{
public static void main(String args[]){
StringBuilder sb=new StringBuilder("Hello");
sb.replace(1,3,"Java");
System.out.println(sb);//prints HJavalo
}
}
```

4) StringBuilder delete() method

- This method of StringBuilder class deletes the string from the specified beginIndex to endIndex.

```
class StringBuilderExample4{
public static void main(String args[]){
StringBuilder sb=new StringBuilder("Hello");
sb.delete(1,3);
System.out.println(sb);//prints Hlo
}
}
```

5) StringBuilder reverse() method

- The reverse() method of StringBuilder class reverses the current string.

```
class StringBuilderExample5{
public static void main(String args[]){
StringBuilder sb=new StringBuilder("Hello");
```

```
sb.reverse();
System.out.println(sb);//prints olleH
}
}
```

6) StringBuilder capacity() method

- The capacity() method of StringBuilder class returns the current capacity of the Builder. The default capacity of the Builder is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity*2)+2.
- For example if your current capacity is 16, it will be (16*2)+2=34.

```
class StringBuilderExample6{
public static void main(String args[]){
StringBuilder sb=new StringBuilder();
System.out.println(sb.capacity());//default 16
sb.append("Hello");
System.out.println(sb.capacity());//now 16
sb.append("java is my favourite language");
System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
}
}
```

7) StringBuilder ensureCapacity() method

- The ensureCapacity() method of StringBuilder class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity*2)+2.
- For example if your current capacity is 16, it will be (16*2)+2=34.

```
class StringBuilderExample7{
public static void main(String args[]){
StringBuilder sb=new StringBuilder();
System.out.println(sb.capacity());//default 16
sb.append("Hello");
System.out.println(sb.capacity());//now 16
sb.append("java is my favourite language");
System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
sb.ensureCapacity(10);//now no change
System.out.println(sb.capacity());//now 34
sb.ensureCapacity(50);//now (34*2)+2
System.out.println(sb.capacity());//now 70
}
}
```

Q. What is mutable string?

ANS: A string that can be modified or changed is known as mutable string. StringBuffer and StringBuilder classes are used for creating mutable string.

<u>Difference between String and StringBuffer</u>

There are many differences between String and StringBuffer. A list of differences between String and StringBuffer are given below:

String	StringBuffer
String class is immutable.	StringBuffer class is mutable.
String is slow and consumes more memory when you concat too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when you cancat strings.

String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.

Difference between StringBuffer and StringBuilder

- Java provides three classes to represent a sequence of characters: String, StringBuffer, and StringBuilder. The String class is an immutable class whereas StringBuffer and StringBuilder classes are mutable. There are many differences between StringBuffer and StringBuilder. The StringBuilder class is introduced since JDK 1.5.
- A list of differences between StringBuffer and StringBuilder are given below:

StringBuffer	StringBuilder
StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
StringBuffer is *less efficient* than StringBuilder.	StringBuilder is *more efficient* than StringBuffer.

Wrapper Classes in Java

A Wrapper class is a class whose object wraps or contains a primitive data type. When we create an object to a wrapper class, it contains a field and, in this field, we can store a primitive data types. In other words, we can wrap a primitive value into a wrapper class object.

Need of Wrapper Classes

- They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
- The classes in java.util package handles only objects and hence wrapper classes help in this case also. Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.

Primitive Data types and their Corresponding Wrapper class

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
long	Integer
float	Float
double	Double
boolean	Boolean

Autoboxing and Unboxing

1. **Autoboxing:** Automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example: conversion of int to Integer, long to Long, double to Double etc.
2. **Unboxing:** It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing. For example: conversion of Integer to int, Long to long, Double to double etc.

Example: //Java program to demonstrate Wrapping and UnWrapping in Java Classes

```
class WrappingUnwrapping
{
   public static void main(String args[])
   {
      byte a = 1; //  byte data type
```

```
        Byte byteobj = new Byte(a);        // wrapping around Byte object
        int b = 10;        // int data type
        Integer intobj = new Integer(b);        //wrapping around Integer object
        float c = 18.6f;        // float data type
        Float floatobj = new Float(c);        // wrapping around Float object
        double d = 250.5;        // double data type
        Double doubleobj = new Double(d);        // Wrapping around Double object
        char e='a';        // char data type
        Character charobj=e;        // wrapping around Character object
        //  printing the values from objects
        System.out.println("Values of Wrapper objects (printing as objects)");
        System.out.println("Byte object byteobj:  " + byteobj);
        System.out.println("Integer object intobj:  " + intobj);
        System.out.println("Float object floatobj:  " + floatobj);
        System.out.println("Double object doubleobj:  " + doubleobj);
        System.out.println("Character object charobj:  " + charobj);
        // objects to data types (retrieving data types from objects)
        // unwrapping objects to primitive data types
        byte bv = byteobj;
        int iv = intobj;
        float fv = floatobj;
        double dv = doubleobj;
        char cv = charobj;
        // printing the values from data types
        System.out.println("Unwrapped values (printing as data types)");
        System.out.println("byte value, bv: " + bv);
        System.out.println("int value, iv: " + iv);
        System.out.println("float value, fv: " + fv);
        System.out.println("double value, dv: " + dv);
        System.out.println("char value, cv: " + cv);
    }
}
```

Output:

```
Values of Wrapper objects (printing as objects)
Byte object byteobj:  1
Integer object intobj:  10
Float object floatobj:  18.6
Double object doubleobj:  250.5
Character object charobj: a
Unwrapped values (printing as data types)
byte value, bv: 1
int value, iv: 10
float value, fv: 18.6
double value, dv: 250.5
char value, cv: a
```

MODULE-III
INTERFACE

- Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).
- Interfaces specify what a class must do and not how. It is the blueprint of the class.
- An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So, it specifies a set of methods that the class has to implement.
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.

Syntax:

```
interface <interface_name> {
  // declare constant fields
    // declare methods that abstract by default.
}
```

- To declare an interface, use interface keyword. It is used to provide total abstraction. That means all the methods in an interface are declared with an empty body and are public and all fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface. To implement interface use implements keyword.

Why do we use interface?

- It is used to achieve total abstraction.
- Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance.
- It is also used to achieve loose coupling.
- Interfaces are used to implement abstraction. So, the question arises why use interfaces when we have abstract classes? The reason is, abstract classes may contain non-final variables, whereas variables in interface are final, public and static.

```
// A simple interface
interface Player
{
    final int id = 10;
    int move();//abstract method
}
```

- To implement an interface we use keyword: implements

Example: // Java program to demonstrate working of interface.

```
import java.io.*;
  // A simple interface
interface In1
{
    // public, static and final
    final int a = 10;
      // public and abstract
    void display();
}
// A class that implements the interface.
class TestClass implements In1
{
    // Implementing the capabilities of interface.
    public void display()
    {        System.out.println("JAVA");     }
```

```
    // Driver Code
    public static void main (String[] args)
    {   TestClass t = new  TestClass();
        t.display();
        System.out.println(a);
    }
}
```

Output: JAVA
10

New features added in interfaces in JDK 8

- Prior to JDK 8, interface could not define implementation. We can now add default implementation for interface methods. This default implementation has special use and does not affect the intention behind interfaces. Suppose we need to add a new function in an existing interface.
- Obviously, the old code will not work as the classes have not implemented those new functions. So with the help of default implementation, we will give a default body for the newly added functions. Then the old codes will still work.

// An example to show that interfaces can have methods from JDK 1.8 onwards

```
interface In1
{
    final int a = 10;
    default void display()
    {
        System.out.println("hello");
        }
}
  // A class that implements the interface.
class TestClass implements In1
{
    // Driver Code
    public static void main (String[] args)
    {
        TestClass t = new TestClass();
        t.display();
    }
}
```

Output: hello

- Another feature that was added in JDK 8 is that we can now define static methods in interfaces which can be called independently without an object.
- Note: these methods are not inherited.

// An example to show that interfaces can have methods from JDK 1.8 onwards

```
interface In1
{
    final int a = 10;
    static void display()
    {
        System.out.println(a);
         System.out.println("hello");
     }
}
```

```
// A class that implements the interface.
class TestClass implements In1
{
   // Driver Code
   public static void main (String[] args)
   {
         In1.display();
   }
}
Output: 10
         hello
```

Important points about interface:

- We can't create instance (interface can't be instantiated) of interface but we can make reference of it that refers to the Object of its implementing class.
- A class can implement more than one interface.
- An interface can extend another interface or interfaces (more than one interface).
- A class that implements interface must implements all the methods in interface.
- All the methods are public and abstract. And all the fields are public, static, and final.
- It is used to achieve multiple inheritance.

<u>Nested Interface in Java</u>

- We can declare interfaces as member of a class or another interface. Such an interface is called as member interface or nested interface.

i. Interface in a class:

- Interfaces (or classes) can have only public and default access specifiers when declared outside any other class.
- This interface declared in a class can either be default, public, protected not private. While implementing the interface, we mention the interface as c_name.i_name where c_name is the name of the class in which it is nested and i_name is the name of the interface itself.

Let us have a look at the following code:

// Java program to demonstrate working of interface inside a class.

```
import java.util.*;
class Test
{
   interface Yes
   {
         void show();
    }
}

class Testing implements Test.Yes
{
   public void show()
   {
         System.out.println("show method of interface");
    }
}
  class A
{
   public static void main(String[] args)
   {
      Test.Yes obj; //reference object
```

```
        Testing t = new Testing();
        obj=t;
        obj.show();
    }
}
```

Output: show method of interface

ii. Interface in another Interface:

- An interface can be declared inside another interface also.
- We mention the interface as i_name1.i_name2 where i_name1 is the name of the interface in which it is nested and i_name2 is the name of the interface to be implemented.

// Java program to demonstrate working of interface inside another interface.

```
import java.util.*;
interface Test {
  interface Yes {
      void show();
  }
}
class Testing implements Test.Yes {
  public void show() {
      System.out.println("show method of interface");
   }
}
class A
{
  public static void main(String[] args)
  {
    Test.Yes obj;
    Testing t = new Testing();
    obj = t;
    obj.show();
  }
}
```

Output: show method of interface

iii. Interface in another Interface:

- An interface can be declared inside another interface also.
- We mention the interface as i_name1.i_name2 where i_name1 is the name of the interface in which it is nested and i_name2 is the name of the interface to be implemented.

// Java program to demonstrate working of interface inside another interface.

```
import java.util.*;
interface Test {
  interface Yes  {
    void show();
  }
}
class Testing implements Test.Yes
{
  public void show()
  {      System.out.println("show method of interface");    }
}
class A
{
```

```
  public static void main(String[] args)
  {
    Test.Yes obj;
    Testing t = new Testing();
    obj = t;
    obj.show();
  }
}
```

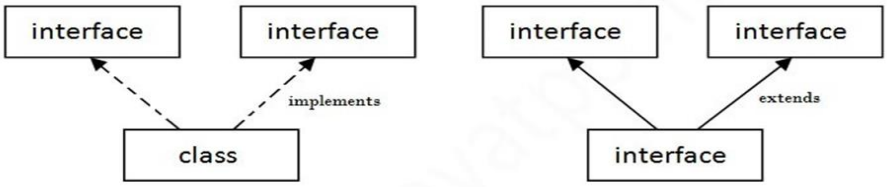
Output: show method of interface

```
class A
{
   public static void main(String[] args)
   {
      Test.Yes obj;
      Testing t = new Testing();
      obj = t;
      obj.show();
   }
}
```

Output: illegal combination of modifiers: public and protected
protected interface Yes

iv. Multiple inheritance in Java by interface

- If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple Inheritance in Java

Example:

```
interface Printable{
void print();  //abstract method
}
interface Showable{
void show();  //abstract method
}
class A implements Printable, Showable
{
public void print(){
System.out.println("Hello");}
public void show(){
System.out.println("Welcome");
}
public static void main(String args[]){
A obj = new A();
obj.print();
obj.show();
 }
```

```
}
Output:
        Hello
        Welcome
```

- As we have explained in the inheritance, multiple inheritance is not supported in the case of class because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementing class.

Example:

```
interface Printable{
void print();
}
interface Showable{
void print();
}
class TestInterface implements Printable, Showable{
public void print(){
System.out.println("Hello");
}
public static void main(String args[]){
TestInterface obj = new TestInterface();
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 TestInterface, so there is no ambiguity.

<u>Interface inheritance</u>

- A class implements an interface, but one interface extends another interface.

Example:

```
interface Printable{
void print();
}
interface Showable extends Printable{
void show();
}
class TestInterface implements Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}
public static void main(String args[]){
TestInterface obj = new TestInterface();
obj.print();
obj.show();
 }
}
Output:
Hello
Welcome
```

<u>Abstract Classes in Java</u>

1) In Java, a separate keyword abstract is used to make a class abstract. An abstract class contains an abstract method.

```
// An example abstract class in Java
abstract class Shape {
   int color;
     // An abstract function
   abstract void draw();
}
```

Example:

```
abstract class Base {
   abstract void fun(); //to be redefined(overridden) in its subclass
}
class Derived extends Base {
   void fun() { System.out.println("Derived fun() called"); }
}
class Main {
   public static void main(String args[]) {
 /*Uncommenting the following line will cause compiler error as it tries to create
an instance of abstract class.*/
      //Base b = new Base();
      // But, We can have references of Base type.
       Base b = new Derived();
       b.fun();
   }
}
```

Output: Derived fun() called

2) An abstract class can contain constructors in Java. And a constructor of abstract class is called when an instance of a inherited class is created. For example, the following is a valid Java program.

```
// An abstract class with constructor
abstract class Base {
   Base() { System.out.println("Base Constructor Called"); }
   abstract void fun();
}
class Derived extends Base {
   Derived() { System.out.println("Derived Constructor Called"); }
   void fun() { System.out.println("Derived fun() called"); }
}
class Main {
   public static void main(String args[]) {
      Derived d = new Derived();
      d. fun();
   }
}
```

Output:

```
Base Constructor Called
Derived Constructor Called
Derived fun() called
```

3) In Java, we can have an abstract class without any abstract method. This allows us to create classes that cannot be instantiated, but can only be inherited.

```
// An abstract class without any abstract method
abstract class Base {
   void fun() {
         System.out.println("Base fun() called"); }
}
```

```
class Derived extends Base { }
class Main {
 public static void main(String args[]) {
    Derived d = new Derived();
    d.fun();
 }
}
```

Output:

```
Base fun() called
```

4) Abstract classes can also have final methods (methods that cannot be overridden). For example, the following program compiles and runs fine.

Example: // An abstract class with a final method

```
abstract class Base {
   final void fun() {
         System.out.println("Derived fun() called");
         }
}
  class Derived extends Base {}
  class Main {
   public static void main(String args[]) {
      Base b = new Derived(); //reference object of an abstract class
      b.fun();
   }
}
```

Output:

```
Derived fun() called
```

5) For any abstract java class we are not allowed to create an object i.e., for abstract class instantiation is not possible.

Example: // An abstract class example

```
abstract class Test
{
     public static void main(String args[])
    {
         // Try to create an object
         Test t=new Test();
    }
}
```

Output:

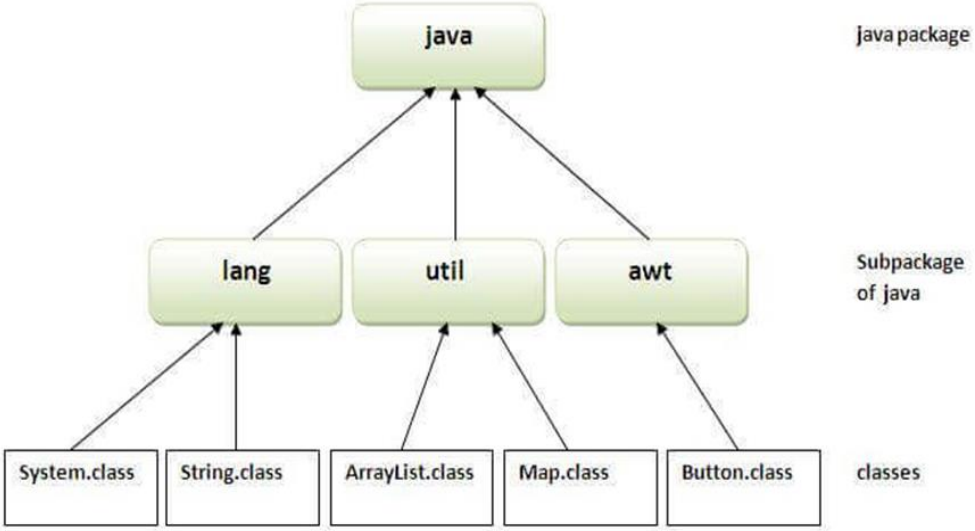
```
Compile time error. Test is abstract;
cannot be instantiated Test t=new Test();
```

Java Package

- 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.



<u>Use of packages</u>

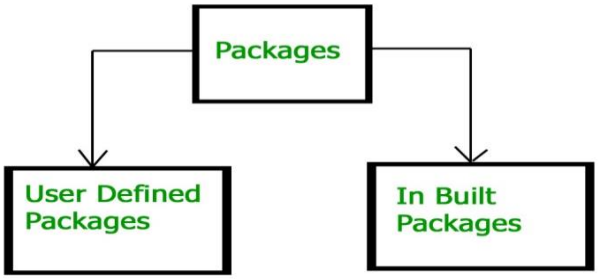
Packages are used for:

- Preventing naming conflicts. For example there can be two classes with name Employee in two different packages, college.staff.cse.Employee and college.staff.ee.Employee
- Making searching/locating and usage of classes, interfaces and annotations easier
- Providing controlled access: protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.
- Packages can be considered as data encapsulation (or data-hiding).

NOTE:

- All we need to do is put related classes into packages. After that, we can simply write an import class from existing packages and use it in our program.
- A package is a container of a group of related classes where some of the classes are accessible are exposed and others are kept for internal purpose.
- We can reuse existing classes from the packages as many time as we need it in our program.

<u>Types of packages</u>



Built-in Packages:

These packages consist of a large number of classes which are a part of Java API. Some of the commonly used built-in packages are:

1) java.lang: Contains language support classes(e.g classed which defines primitive data types, math operations). This package is automatically imported.
2) java.io: Contains classed for supporting input / output operations.
3) java.util: Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.
4) java.applet: Contains classes for creating Applets.
5) java.awt: Contain classes for implementing the components for graphical user interfaces (like button, menus etc).
6) java.net: Contain classes for supporting networking operations.

User-defined packages:

- These are the packages that are defined by the user.

How packages work?

- Package names and directory structure are closely related. For example, if a package name is college.staff.cse, then there are three directories, college, staff and cse such that cse is present in staff and staff is present college. Also, the directory college is accessible through CLASSPATH variable, i.e., path of parent directory of college is present in CLASSPATH. The idea is to make sure that classes are easy to locate.
- Package naming conventions: Packages are named in reverse order of domain names, i.e., org.abit.practice. For example, in a college, the recommended convention is college.tech.cse, college.tech.ee, college.art.history, etc.
- Adding a class to a Package: We can add more classes to a created package by using package name at the top of the program and saving it in the package directory. We need a new java file to define a public class, otherwise we can add the new class to an existing .java file and recompile it.
- Subpackages: Packages that are inside another package are the subpackages. These are not imported by default; they have to imported explicitly. Also, members of a subpackage have no access privileges, i.e., they are considered as different package for protected and default access specifiers.

Example:

```
import java.util.*;
//util is a subpackage created inside java package.
 //Accessing classes inside a package
```

Consider following two statements:

```
// import the Vector class from util package.
        import java.util.vector;
// import all the classes from util package
        import java.util.*;
```

i. First Statement is used to import Vector class from util package which is contained inside java.
ii. Second statement imports all the classes from util package.

```
// All the classes and interfaces of this package will be accessible but not subpackages.
        import package.*;
// Only mentioned class of this package will be accessible.
        import package.classname;
// Class name is generally used when two packages have the same class name.
```

- For example, in below code both packages have date class so using a fully qualified name to avoid conflict

```
        import java.util.Date;
        import my.package.Date;
```

Simple example of java package

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

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

How to compile java package?

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

```
        javac -d directory javafilename
```

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

How to run java package program?

- You need to use fully qualified name e.g. mypack.Simple etc to run the class.
- To Compile: javac -d . Simple.java
- To Run: java mypack.Simple

Output:

```
Welcome to package
```

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

How to access package from another package?

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

```
        import package.*;
        import package.classname;
```

i) fully qualified name: 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

ii) 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

iii) 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

Subpackage in java

- Package inside the package is called the subpackage. It should be created to categorize the package further.
- Let's take an example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on.
- So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.
- The standard of defining package is domain.company.package e.g. com.abit.bean or org.sssit.dao.

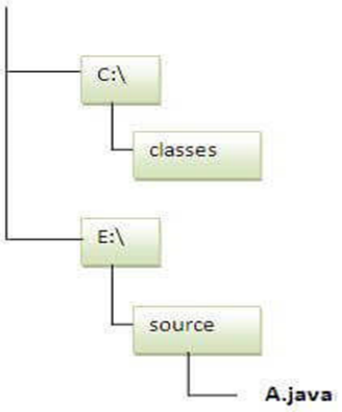
Example of Subpackage

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

How to send the class file to another directory or drive?

There is a scenario, where we want to put the class file of A.java source file in classes folder of c: drive.

For example:



Example

```
//save as Simple.java
package mypack;
public class Simple{
 public static void main(String args[]){
   System.out.println("Welcome to package");
  }
}
To Compile:
e:\sources> javac -d c:\classes Simple.java
To Run:
 To run this program from e:\source directory, you need to set classpath of the directory where the cl
 e:\sources> set classpath=c:\classes;.;
 e:\sources> java mypack.Simple
```

EXCEPTION HANDLING

- An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.
- In java, an exception is an object that wraps an error event that occurred within a method and it contains the followings:
  - i) Information about the error including its type
  - ii) The state of the program when the error occurred
  - iii) Optionally, other custom information

Exceptions are used to indicate many different types of error conditions:

I. JVM Errors:

- OutOfMemoryError
- StackOverflowError
- LinkageError

II. System Errors:

- FileNotFoundException
- IOException
- SocketTimeoutException

III. Programming Errors:

- NullPointerException
- ArrayIndexOutOfBoundsException
- ArithmeticException

Why we use Exceptions:

We use the above Exceptions because of following reasons:

- Exceptions separate error handling codes from regular code.
- Exceptions propagate errors up the call stack. i.e, for nested methods do not have to explicitly catch and Forward errors
- Exception classes the group and differentiate error type
- Exceptions standardize error handling

TRY, CATCH&MULTIPLECATCH IN EXCEPTION HANDLING

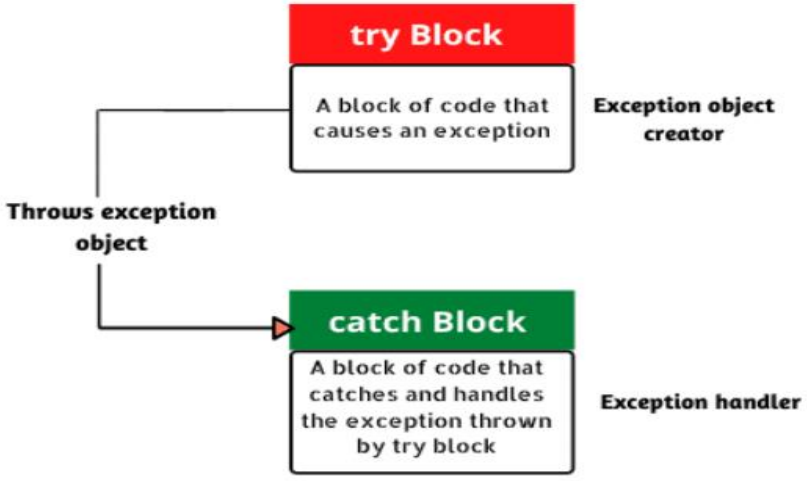


Fig: Exception handling mechanism

General form of Exception Handling

```
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// … finally {
// block of code to be executed before try block ends
}
```

By using exception to managing errors, Java programs have the following advantage over traditional error management techniques:

- Separating Error handling code from regular code.
- Propagating error up the call stack.
- Grouping error types and error differentiation.

For Example:

```
class Exc0 {
public static void main (String args[]) {
int d = 0;
int a = 42 / d;
}
}
```

- When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception. This causes the execution of Exc0 to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately.
- In this example, we haven't supp the exception is caught by the default handler provided by the Java run-time system.
- Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program. Here is the output generated when this example is executed.

  java.lang.ArithmeticException: / by zero at Exc0.main(Exc0.java:4)
- Notice how the class name, Exc0; the method name, main; the filename, Exc0.java; and the line number, 4

Try Block:

- If we don't want to prevent the programapthe to trap the exception using the try block. So we can place the statements that may causes an exception in the try block.

  try{
  //statement }
- If an exception occurs with in the try block, the appropriate exception handler that is associated with the try block handles the exception immediately following the try block, include a catch clause specifies the exception type we wish to catch.
- A try block must have at least one catch block or finally that allows it immediately.

Catch block:

- The catch block is used to process the exception raised. A try block can be one or more catch blocks can handle a try block.
- Catch handles are placed immediately after the try block.

```
Catch(exceptiontype e)
{
//Error handle routine is placed here for handling exception
}
```

Program 1:

```
class trycatch
{
Public static void main (String args[])
{
int[] no={1,2,3};
try
{
System.out.println(no[3]);
}
catch (ArrayIndexOutOfBoundsException e)
{
System.out.println("Out of bounds");
}
System.out.println("Quit");
}
}
```

Output: Out of the Range Quit

Program 2:

```
class ArithExce
{
Public static void main (String args[])
{
int a=10, b=0;
try
{
a=a/b;
System.out.println("Won't Print");
}
catch (ArithmeticException e)
{
System.out.println("Division by Zero error");
System.out.println("Change the b value");
}
System.out.println("Quit");
}
}
```

Output: Division By zero error Please change the B value Quit

Note:

- A try and its catch statement form a unit.
- We cannot use try block alone.
- The compiler does not allow any statement between try block and its associated catch block.

Displaying description of an Exception

- Throwable overrides the toString() method (defined by Object) so that it returns a string containing a description of the exception.
- We can display this description in a println statement by simply passing the exception as an argument.

```
catch (ArithmeticException e)
{
System.out.println("Exception: " + e);
a = 0; // set a to zero and continue
}
```

- When this version is substituted in the program, and the program is run, each divide-byzero error displays the following message:

```
Exception: java.lang.ArithmeticException: / by zero
```

Multiple Catch Blocks

- In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception. When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed. After one catch statement executes, the others are bypassed, and execution continues after the try/catch block. The following example traps two different exception types:

// Demonstrate multiple catch statements.

```
class MultiCatch {
public static void main (String args[]) {
try {
int a = args.length;
System.out.println("a = " + a);
int b = 42 / a;
int c[] = { 1 };
c[42] = 99;
}
catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
}
catch(ArrayIndexOutOfBoundsException e){
System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
}
}
```

- This program will cause a division-by-zero exception if it is started with no command line parameters, since a will equal zero.
- It will survive the division if you provide a command-line argument, setting a to something larger than zero. But it will cause a parameter, since a will equal zero. It will survive the division if you provide a command-line argument, setting a to something larger than zero. But it will cause an ArrayIndexOutOfBoundsException.

USE OF THROW &THROWS AND FINALLY KEYWORDS

1. Throw Keyword

- We have only been catching exceptions that are thrown by the Java Run-Time systems. However, it is possible for our program to throw an exception explicitly, using the throw statement.
  Throw throwableInstance
- Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable. Simple types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions
- There are two ways you can obtain a Throwable object:

i. using a parameter into a catch clause
ii. creating one with the new operator.

- The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of the exception.
- If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

```
// Demonstrate throw
class ThrowDemo {
static void demoproc()
{
try
{       throw new NullPointerException("demo");
}
catch(NullPointerException e)
{
System.out.println("Caught inside demoproc.");
throw e; // rethrow the exception
}
}
public static void main(String args[])
{
try
{       demoproc();
}
catch(NullPointerException e) {
System.out.println("Recaught: " + e);
}
}
}
```

- This program gets two chances to deal with the same error. First, main( ) sets up an exception context and then calls demoproc( ). The demoproc( ) method then sets up another exception-handling context and immediately throws a new instance of NullPointerException, which is caught on the next line. The exception is then rethrown.
- Here is the resulting output:
  Caught inside demoproc.
  Recaught: java.lang.NullPointerException: demo
- The program also illustrates how to create one of J close attention to this line:
  throw new NullPointerException("demo");
- Here, new is used to construct an instance of NullPointerException.

- All of Java's run-time exceptions have at least two constructors: one with no parameter and one that takes a string parameter. When the second form is used, the argument specifies a string that describes the exception.
- This string is displayed when the object is used as an argument to print( ) or println( ). It can also be obtained by a call to getMessage( ), which is defined by Throwable.

Throws Keyword

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a throws clause in the method's throws clause lists declare the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses. All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result.
- This is the general form of a method declaration that includes a throws clause:

```
Return-type method-name (parameter-list) throws exception-list
{
// bodyof method
}
```

- Here, exception-list is a comma-separated list of the exceptions that a method can throw.

Program

```
class ThrowsDemo {
static void throwOne() throws IllegalAccessException {
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(Stringargs[]) {
try{
throwOne();
}
catch (IllegalAccessException e){
System.out.println("Caught " + e);
}
}
}
```

Here is the output generated by running this example program:

```
inside throwOne
caught java.lang.IllegalAccessException
```

Difference between Throw & Throws keyword:-

Throw	Throws
1. Java throw keyword is used to explicitly throw an exception.	1. Java throws keyword is used to declared an exception.
2. Checked exception cannot be propagated throws.	2.Checked exception can be propagated with using throws only.
3. Throw is followed by an instance.	3.Throws is followed by class
4. Throw is used within the method.	4.Throws is used with the method signature.
5.You can't throw multiple exceptions.	5. You can declare multiple exceptions e.g. public void method() throws IOException,SQLException.

Finally Block:

- When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method.
- Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods.
- For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception- handling mechanism.
- The finally keyword is designed to address this contingency. finally creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block.
- The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception.
- Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns.
- This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The finally clause is optional. However, each try statement requires at least one catch or a finally clause.

```
// Demonstrate finally.
class FinallyDemo {
// Through an exception out of the method.
static void procA(){
try {
System.out.println("inside procA");
throw new RuntimeException("demo");
}
finally {        System.out.println("procA's finally");        }
}
// Return from within a try block.
static void procB() {
try {
System.out.println("inside procB");
return;
}
finally {        System.out.println("procB's  finally”);        }
}
// Execute a try block normally.
static void procC() {
try {
System.out.println("inside procC");
}
finally {        System.out.println("procC's finally");        }
}
public static void main (String args[])
{
try {
procA();
}
catch (Exception e) {
System.out.println("Exception caught");
}
```

```
procB();
procC();
}
}
```

- In this example In this example, procA( ) prematurely breaks out of the try by throwing an exception. The finally clause is executed on the way out. procB( )'stry statement is exited via a return statement. The finally clause is executed before procB( ) returns. In procC( ), the try statement executes normally, without error. However, the finally block is still executed. If a finally block is associated with a try, the finally block will be executed upon conclusion of thetry.
- Here is the output generated by the preceding program:
  inside procA
  procA's finally
  Exception caught
  inside procB
  procB's finally
  inside procC
  procC's finally

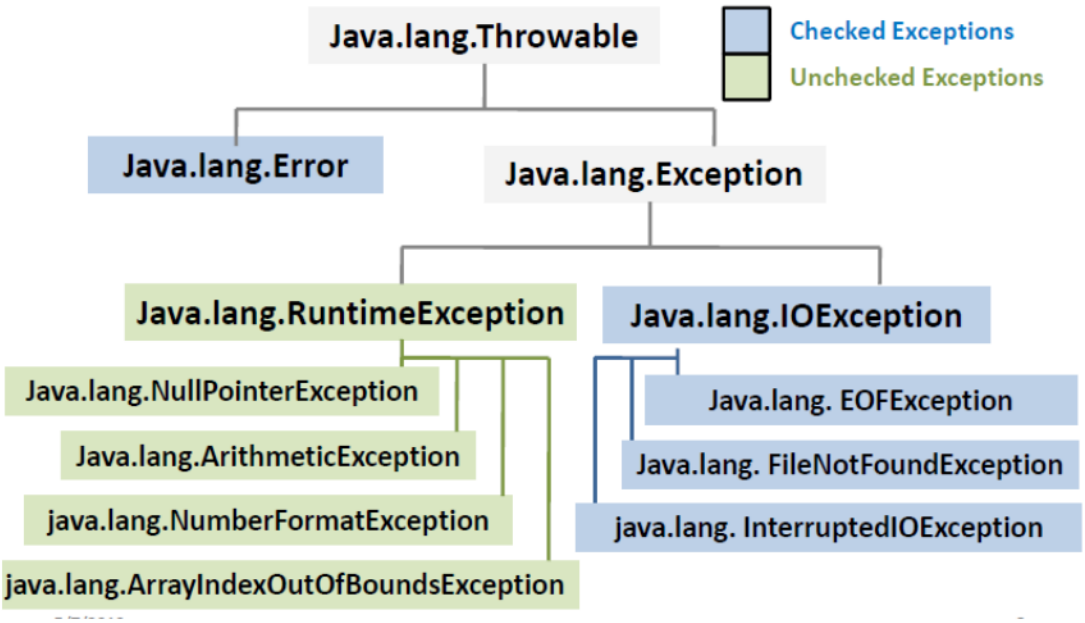
Difference between final, finally and finalize:

Final	Finally	Finalize

1. Final is used to apply restrictions on class, method & variables.
2. Final class can't be inherited.
3. Final method can't be executed overridden.
4. Final variable value can't be changed.
5.Final is a keyword.
EXAMPLE:
class FinalExample
{
Public static void main (String args[])
{
Final int x=100;
X=200;//compile time error
}
}
	1.Finally is used to place important code, it will perform whether processing exception is handled or object is not.
	2.Finally is a block.
	EXAMPLE:
	class FinallyExample
	{
	Public static void main (String args[])
	{
	try
	{
	int x=300;
	}
	catch (Exception e){
	System.out.println(e);}
	finally{
	System.out.println("finally block is executed");}
	}
	}
		1.Finalize is used to clean up just before garbage collected.
		2.Finalize is a method.
		EXAMPLE:
		Class FinalizeExample{
		Public void finalize() {
		System.out.println("finalize called");}
		Public static void main(String[] args){
		FinalizeExample f1=new FinalizeExample();
		FinalizeExample f2=new FinalizeExample();
		f1=null;
		f2=null;
		System.gc();
		}
		}

INTRODUCTION TO JAVA EXCEPTION CLASSES

Hierarchy of Java Exception classes:



Exception Hierarchy

Java Built-In Exception:-

Many exceptions and errors are automatically generated by the java virtual machine. Errors generally abnormal situations in the JVM, such as:

- Running out of memory
- Infinite recursion
- Inability to link to another class

1. Runtime exceptions: generally, a result of programming errors, such as:

- Dereferencing a null reference
- Trying to read outside the bounds of an array
- Dividing an integer value by zero

Example:

```
public class ExceptionDemo {
public static void main (String args[]) {
System.out.println(divideArray(args));
}
private static int divideArray(String array[]) {
String s1=array[0];
String  s2=array[1];
return divideString(s1,s2);
}
private static int divideString(String s1,String s2) {
int  i1=Integer.parseInt(s1);
int i2=Integer.parseInt(s2);
return divideInts(i1,i2);
}
private static int  divideInts(int i1,int i2) {
return i1/i2;
}
}
```

OUTPUT:

Java ExceptionDemo 100 4
Java ExceptionDemo 100 0
Java ExceptionDemo 100 four
Java ExceptionDemo 100

Java unchecked Exception:

- Errors and Runtime Exceptions are unchecked exceptions that is, the compiler does not check that you handle them explicitly.
- Methods do not have to declare that they throw them.
- It is assumed that theapplication can't do anything to recover from these exceptions.
- Unchecked exceptions give programmers the power to ignore exceptions that they can't recover from, and only handle the ones they can.
- This leads to less clutter. However, many programmers simply ignore unchecked exceptions because they are by default unrecoverable. Turning all exceptions into the unchecked type would likely lead to poor overall error handling.

List of Unchecked Exceptions

EXCEPTION	MEANING
1.ArithmeticException	Arithmetic error, such as divide_by_zero
2.ArrayIndexOutOfBoundsException	Array index is out of bounds
3.ArrayStroreException	Assignment to an array element of an incompatible type
4.ClassCastException	Invalid cast
5.IllegalArgumentException	Illegal argument used to invoke a method.
6.IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
7.IllegalStateException	Environment or application is in incorrect state.
8.IllegalThreadStateException	Requested operation not compatible with current thread state.
9.IndexOutOfBoundsException	Some type of index is out of bounds.
10.NegativeArraySizeException	Array created with a negative size.
11.NullPointerException	Invalid use of a null reference.
12.NumberFormatException	Invalid conversion of a string to a numeric format.
13.SecurityException	Attempt to violate security.
14.StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
15.UnsupportedOperationException	An unsupported operation was encountered.

Java checked Exception:

- All other exceptions except unchecked exceptions are called checked exception, that is, the compiler enforces that you handle them explicitly.
- Methods that generate checked exceptions must declare that they throw them.
- Methods that invoke other methods that throw checked exceptions must either handle them or let them propagate by declaring that they throw them.
- Checked exceptions give API designers the power to force programmers to deal with the exceptions.
- API designers except programmers to be able to reasonably recover from those exceptions, even if that just means logging the exceptions or returning error messages to the users.

List of Unchecked Exceptions

EXCEPTION	MEANING
1.ClassNotFoundException	Class not found.
2.CloneNotSupportedException	Attempt to clone an object that does implement the cloneable interface.

3.IllegalAccessException	Access to a class is denied.
4.InstantiationException	Attempt to create an object of an abstract class or interface.
5.InterruptedException	One thread has been interrupted by another thread.
6.NoSuchFieldException	A requested field does not exist.
7.NoSuchMethodException	A requested method does not exist.

User defined exceptions

- We can create our own exception by extending exception class.
- The throw and throws keywords are used while implementing user defined exceptions

Example:

```
Class ownExcepion extends Exception {
ownException(String msg){
Super(msg);
}
}
Class test
{
public static void main(String args[]){
int mark=101;
try {
if(mark>100)
{
throw new ownException("Marks is >100");
}
}
catch (ownException e) {
System.out.println ("Exception caught");
System.out.println.(e.getMessage());
}
finally
{
System.out.println("End of program");
}
}
}
```

Output:

Exception caught
Marks is > 100
End of program

CONCEPT OF MULTITHREADING

MULTITHREADING:

- Java provides a built-in support for multithreaded programming.
- A multithreaded program contains two or more parts that can run concurrently.

- Each part of such a program called thread.
- Each thread defines a separate path of execution.
- Thus, multi thread is a specialized form of multitasking.
- Multitasking is supported by OS.
- There are two distinct types of multitasking:

I. Process based multitasking:

- Process is a program that is executing.
- In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.
- Process based multitasking is a feature that allows computer to run two or more programs concurrently
- For example: This tasking enables us to run the Java compiler and texteditor at the same time.

II. Thread based multitasking:

- Thread is a smallest unit of dispatchable code
- The single program can perform two or more tasks simultaneously.
- For example: A text editor can format text at the same time that is printing as long as these two actions are performed by two separate threads.
- Multitasking threads require less overhead than multitasking processes.

Thread concept:

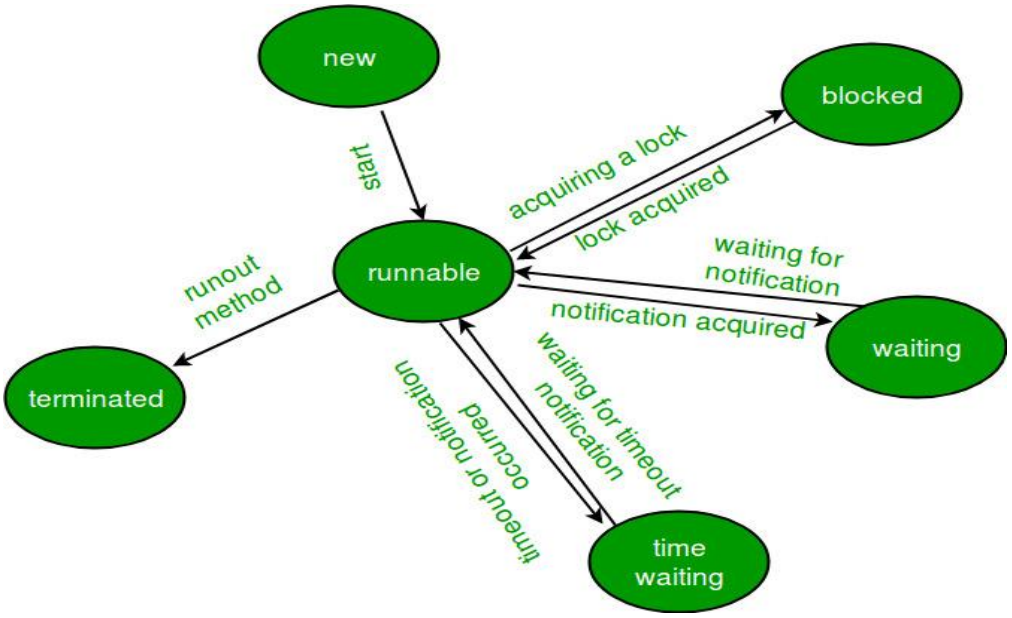
- One thread can pause without stopping other parts of your program.
- For example, the idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere.
- Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause.
- When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run.

Thread States:

- Threads exist in several states.
- A thread can be running. It can be ready to run as soon as it gets CPU time.
- A running thread can be suspended, which temporarily suspends its activity.
- A suspended thread can then be resumed, allowing it to pick up where it left off.
- A thread can be blocked when waiting for a resource.
- At any time, a thread can be terminated, which halts its execution immediately.
- Once terminated, a thread cannot be resumed
- A thread goes through various stages in its life cycle.
- For example: A thread is born, started, runs and then dies.

The following diagram shows the complete life cycle of a thread.

LIFE CYCLE OF A THREAD:



STAGES OF LIFE CYCLE:

i. New: A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.

ii. Runnable: After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.

iii. Running:

- Running means that the processor has given its time to the thread for its execution.
- The thread runs until it is pre-empted or releases its control to a highest priority thread.
- A running thread may release its control by one of the following methods.
  - **Suspended:** It has been suspended by using suspend () go to the block state & again retrieve using the method resume ().
  - **Sleep:** The 2nd possibility is the thread is mode to sleep. We can put a thread to sleep for a specify time period using the method sleep(time).
  - **After:** Again, to make it activated after that period of time we have to call after(time).

iv. Wait: The thread may wait until some event occurs. This is done by using the wait (). The thread can be scheduled again by using the notify ().

v. Blocked State:

- A thread is said to be inn blocked state when it is prevented from entering into the runnable state or running state.
- This happens when a thread is suspended/sleeping/waiting.
- The blocked threads are considered as "not runnable" or "not dead" that means they can run again.

vi. Dead State:

- Every thread has a life cycle. A running thread ends its life when its execution is complete using the run(). Once the thread execution is over we may killed the thread using the stop () & this is called as the dead state.

How to create thread?

There are two ways to create a thread:

- By extending Thread class
- By implementing Runnable interface.

1. Thread class:

- Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r, String name)

Commonly used methods of Thread class:

public void run(): is used to perform action for a thread.
public void start(): starts the execution of the thread.JVM calls the run() method on the thread.
public void sleep(long miliseconds): Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
public void join(): waits for a thread to die.
public void join(long miliseconds): waits for a thread to die for the specified miliseconds.
public int getPriority(): returns the priority of the thread.
public int setPriority(int priority): changes the priority of the thread.
public String getName(): returns the name of the thread.
public void setName(String name): changes the name of the thread.
public Thread currentThread(): returns the reference of currently executing thread.
public int getId(): returns the id of the thread.
public Thread.State getState(): returns the state of the thread.
public boolean isAlive(): tests if the thread is alive.
public void yield(): causes the currently executing thread object to temporarily pause and allow other threads to execute.
public void suspend(): is used to suspend the thread(depricated).
public void resume(): is used to resume the suspended thread(depricated).
public void stop(): is used to stop the thread(depricated).
public boolean isDaemon(): tests if the thread is a daemon thread.
public void setDaemon(boolean b): marks the thread as daemon or user thread.
public void interrupt(): interrupts the thread.
public boolean isInterrupted(): tests if the thread has been interrupted.
public static boolean interrupted(): tests if the current thread has been interrupted.

2. Runnable interface:

- The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().
- **public void run():** is used to perform action for a thread.

Starting a thread:

- **start() method** of Thread class is used to start a newly created thread.
- It performs following tasks: A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

1) Java Thread Example by extending Thread class

```
class Multi extends Thread{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi t1=new Multi();
t1.start();
 }
}
```

Output: thread is running...

2) Java Thread Example by implementing Runnable interface

```
class Multi3 implements Runnable{
public void run(){
System.out.println("thread is running...");
}

public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);
t1.start();
 }
}
```

Output: thread is running...

If you are not extending the Thread class, your class object would not be treated as a thread object. So, you need to explicitly create Thread class object. We are passing the object of your class that implements Runnable so that your class run() method may execute.

EXAMPLE: // Java code for thread creation by extending the Thread class

```
class MultithreadingDemo extends Thread {
   public void run()
    {
      try {
         // Displaying the thread that is running
         System.out.println ("Thread " + Thread.currentThread().getId() + " is running");
       }
      catch (Exception e)
      {
         // Throwing an exception
         System.out.println ("Exception is caught");
      }
   }
}
// Main Class
public class Multithread
{
   public static void main(String[] args)
   {
      int n = 8; // Number of threads
      for (int i=0; i<n; i++)
      {
         MultithreadingDemo object = new MultithreadingDemo();
         object.start();
      }
   }
}

// Java code for thread creation by implementing the Runnable Interface
class MultithreadingDemo implements Runnable
{
   public void run()
```

```
    {
        try
        {
            // Displaying the thread that is running
            System.out.println ("Thread " + Thread.currentThread().getId() + " is running");
          }
        catch (Exception e)
        {
            // Throwing an exception
            System.out.println ("Exception is caught");
        }
    }
}
// Main Class
class Multithread
{
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i=0; i<n; i++)
        {
            Thread object = new Thread(new MultithreadingDemo());
            object.start();
        }
    }
}
```

Program on thread priority:

```
// Java program to demonstrate getPriority() and setPriority()
import java.lang.*;
class ThreadDemo extends Thread {
    public void run()    {
        System.out.println("Inside run method");
    }
    public static void main(String[] args)
    {
        ThreadDemo t1 = new ThreadDemo();
        ThreadDemo t2 = new ThreadDemo();
        ThreadDemo t3 = new ThreadDemo();
        // Default 5
        System.out.println("t1 thread priority : " + t1.getPriority());
        // Default 5
        System.out.println("t2 thread priority : " + t2.getPriority());
        // Default 5
        System.out.println("t3 thread priority : " + t3.getPriority());
        t1.setPriority(2);
        t2.setPriority(5);
        t3.setPriority(8);
        // t3.setPriority(21); will throw IllegalArgumentException
        System.out.println("t1 thread priority : " + t1.getPriority());        // 2
        System.out.println("t2 thread priority : " + t2.getPriority());        // 5
        System.out.println("t3 thread priority : " + t3.getPriority());        // 8
```

```
        // Main thread Displays the name of currently executing Thread
        System.out.println( "Currently Executing Thread : " + Thread.currentThread().getName());
       System.out.println( "Main thread priority : " + Thread.currentThread().getPriority());
       // Main thread priority is set to 10
        Thread.currentThread().setPriority(10);
        System.out.println( "Main thread priority : " + Thread.currentThread().getPriority());
    }
}
```

Output

```
t1 thread priority : 5
t2 thread priority : 5
t3 thread priority : 5
t1 thread priority : 2
t2 thread priority : 5
t3 thread priority : 8
Currently Executing Thread : main
Main thread priority : 5
Main thread priority : 10
```

Synchronization in Java

- Synchronization in java is the capability *to control the access of multiple threads to any shared resource*.
- Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Why use Synchronization?

The synchronization is mainly used to

- To prevent thread interference.
- To prevent consistency problem.

Types of Synchronization: There are two types of synchronization

- Process Synchronization
- Thread Synchronization

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

- Mutual Exclusive
    - Synchronized method.
    - Synchronized block.
    - static synchronization.
- Cooperation (Inter-thread communication in java)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:

- by synchronized method
- by synchronized block
- by static synchronization

Concept of Lock in Java

- Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

- From Java 5 the package java.util.concurrent.locks contains several lock implementations.
- Understanding the problem without Synchronization

In this example, there is no synchronization, so output is inconsistent. Let's see the example:

```
class Table{
void printTable(int n){//method not synchronized
  for(int i=1;i<=5;i++){
    System.out.println(n*i);
    try{
     Thread.sleep(400);
    }catch(Exception e){System.out.println(e);}
  }
 }
}
class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}
}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}
class TestSynchronization1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

Output:

```
5
100
10
200
15
300
20
400
25
500
```

Java synchronized method

- If you declare any method as synchronized, it is known as synchronized method.
- Synchronized method is used to lock an object for any shared resource.
- When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

//example of java synchronized method

```
class Table{
 synchronized void printTable(int n){//synchronized method
   for(int i=1;i<=5;i++){
     System.out.println(n*i);
     try{
      Thread.sleep(400);
     }catch(Exception e){System.out.println(e);}
   }
  }
}
class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}
}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}
public class TestSynchronization2{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

Output:

```
5
10
15
20
```

```
25
100
200
300
400
500
```

Example of synchronized method by using annonymous class

In this program, we have created the two threads by annonymous class, so less coding is required.

```
//Program of synchronized method by using annonymous class
class Table{
 synchronized void printTable(int n){//synchronized method
   for(int i=1;i<=5;i++){
     System.out.println(n*i);
     try{
      Thread.sleep(400);
     }catch(Exception e){System.out.println(e);}
   }
 }
}
public class TestSynchronization3{
public static void main(String args[]){
final Table obj = new Table();//only one object
Thread t1=new Thread(){
public void run(){
obj.printTable(5);
}
};
Thread t2=new Thread(){
public void run(){
obj.printTable(100);
}
};
t1.start();
t2.start();
}
}
```

Output:

```
5
10
15
20
25
100
200
300
400
500
```

<u>Synchronized Block in Java</u>

- Synchronized block can be used to perform synchronization on any specific resource of the method.
- Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.
- If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Points to remember for Synchronized block

- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.

Syntax to use synchronized block

```
synchronized (object reference expression) {
  //code block
}
```

Example of synchronized block

Let's see the simple example of synchronized block.

Program of synchronized block

```
class Table{
 void printTable(int n){
   synchronized(this){//synchronized block
     for(int i=1;i<=5;i++){
      System.out.println(n*i);
      try{
       Thread.sleep(400);
      }catch(Exception e){System.out.println(e);}
     }
   }
 }//end of the method
}
class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}
}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}
public class TestSynchronizedBlock1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
```

```
t1.start();
t2.start();
}
}
```

Output:

```
5
10
15
20
25
100
200
300
400
500
```

Same Example of synchronized block by using annonymous class:

//Program of synchronized block by using annonymous class

```
class Table{
 void printTable(int n){
   synchronized(this){//synchronized block
     for(int i=1;i<=5;i++){
      System.out.println(n*i);
      try{
       Thread.sleep(400);
      }catch(Exception e){System.out.println(e);}
     }
   }
}//end of the method
}
public class TestSynchronizedBlock2{
public static void main(String args[]){
final Table obj = new Table();//only one object
Thread t1=new Thread(){
public void run(){
obj.printTable(5);
}
};
Thread t2=new Thread(){
public void run(){
obj.printTable(100);
}
};
t1.start();
t2.start();
}
}
```

Output:

```
5
10
15
20
25
```

100
200
300
400
500

Inter-thread communication in Java

- **Inter-thread communication** or **Co-operation** is all about allowing synchronized threads to communicate with each other.
- Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.
- It is implemented by following methods of **Object class**:
  - wait()
  - notify()
  - notifyAll()

1) wait() method:

- Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
- The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	Description
public final void wait()throws InterruptedException	waits until object is notified.
public final void wait(long timeout) throws InterruptedException	waits for the specified amount of time.

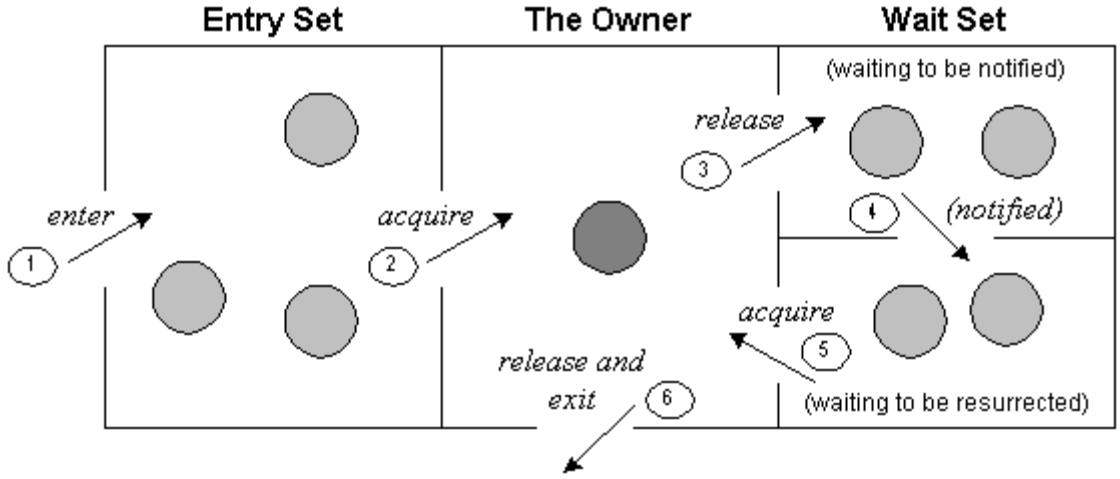
2) notify() method:

- Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.
- **Syntax:** public final void notify()

3) notifyAll() method:

- Wakes up all threads that are waiting on this object's monitor.
- **Syntax:** public final void notifyAll()

Understanding the process of inter-thread communication



The point-to-point explanation of the above diagram is as follows:

- Threads enter to acquire lock.
- Lock is acquired by on thread.
- Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
- If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
- Now thread is available to acquire lock.
- After completion of the task, thread releases the lock and exits the monitor state of the object.

Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?

- It is because they are related to lock and object has a lock.

Difference between wait and sleep?

Let's see the important differences between wait and sleep methods.

wait()	sleep()
wait() method releases the lock	sleep() method doesn't release the lock.
is the method of Object class	is the method of Thread class
is the non-static method	is the static method
should be notified by notify() or notifyAll() methods	after the specified amount of time, sleep is completed.

Example of inter thread communication in java

Let's see the simple example of inter thread communication.

```
class Customer{
int amount=10000;
synchronized void withdraw(int amount){
System.out.println("going to withdraw...");
if(this.amount<amount){
System.out.println("Less balance; waiting for deposit...");
try{wait();}
catch(Exception e){}
}
this.amount-=amount;
System.out.println("withdraw completed...");
}
synchronized void deposit(int amount){
System.out.println("going to deposit...");
this.amount+=amount;
System.out.println("deposit completed... ");
notify();
}
}
class Test{
public static void main(String args[]){
final Customer c=new Customer();
new Thread(){
public void run(){c.withdraw(15000);}
}.start();
new Thread(){
```

```
public void run(){c.deposit(10000);}
}.start();
}
}
```

Output:

```
going to withdraw...
Less balance; waiting for deposit...
going to deposit...
deposit completed...
withdraw completed
```

Daemon Thread in Java

- Daemon thread in java is a service provider thread that provides services to the user thread. Its life depends on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.
- There are many java daemon threads running automatically e.g. gc, finalizer etc.
- You can see all the detail by typing the jconsole in the command prompt. The jconsole tool provides information about the loaded classes, memory usage, running threads etc.

Points to remember for Daemon Thread in Java

- It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
- Its life depends on user threads.
- It is a low priority thread.

Why JVM terminates the daemon thread if there is no user thread?

- The sole purpose of the daemon thread is that it provides services to user thread for background supporting task. If there is no user thread, why should JVM keep running this thread. That is why JVM terminates the daemon thread if there is no user thread.

Methods for Java Daemon thread by Thread class

The java.lang.Thread class provides two methods for java daemon thread.

Method	Description
public void setDaemon(boolean status)	is used to mark the current thread as daemon thread or user thread.
public boolean isDaemon()	is used to check that current is daemon.

Simple example of Daemon thread in java

File: MyThread.java

```
public class TestDaemonThread1 extends Thread{
 public void run(){
  if(Thread.currentThread().isDaemon()){//checking for daemon thread
   System.out.println("daemon thread work");
  }
  else{
  System.out.println("user thread work");
 }
 }
 public static void main(String[] args){
  TestDaemonThread1 t1=new TestDaemonThread1();//creating thread
  TestDaemonThread1 t2=new TestDaemonThread1();
  TestDaemonThread1 t3=new TestDaemonThread1();
```

```
  t1.setDaemon(true);//now t1 is daemon thread
  t1.start();//starting threads
  t2.start();
  t3.start();
 }
}
```

Output

```
daemon thread work
user thread work
user thread work
```

Note: If you want to make a user thread as Daemon, it must not be started otherwise it will throw IllegalThreadStateException.

File: MyThread.java

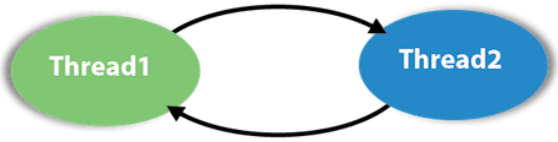
```
class TestDaemonThread2 extends Thread{
 public void run(){
  System.out.println("Name: "+Thread.currentThread().getName());
  System.out.println("Daemon: "+Thread.currentThread().isDaemon());
 }
public static void main(String[] args){
  TestDaemonThread2 t1=new TestDaemonThread2();
  TestDaemonThread2 t2=new TestDaemonThread2();
  t1.start();
  t1.setDaemon(true);//will throw exception here
  t2.start();
 }
}
```

Output:

```
exception in thread main: java.lang.IllegalThreadStateException
```

<u>Deadlock in java</u>

- Deadlock in java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



Example of Deadlock in java

```
public class TestDeadlockExample1 {
  public static void main(String[] args) {
    final String resource1 = "ABC";
    final String resource2 = "XYZ";
    // t1 tries to lock resource1 then resource2
    Thread t1 = new Thread() {
      public void run() {
          synchronized (resource1) {
           System.out.println("Thread 1: locked resource 1");
           try { Thread.sleep(100);}
            catch (Exception e) {}
             synchronized (resource2) {
```

```
                System.out.println("Thread 1: locked resource 2");
              }
            }
          }
        };
        // t2 tries to lock resource2 then resource1
        Thread t2 = new Thread() {
          public void run() {
            synchronized (resource2) {
              System.out.println("Thread 2: locked resource 2");
               try { Thread.sleep(100);}
                catch (Exception e) {}
                synchronized (resource1) {
                System.out.println("Thread 2: locked resource 1");
              }
            }
          }
        };
        t1.start();
        t2.start();
      }
    }
```

Output:

```
Thread 1: locked resource 1
Thread 2: locked resource 2
```

<u>INTTERRUPTION IN THREADS</u>:

- If any thread is suspended or sleeping or waiting using the suspend() or sleep() or wait() then by calling the interrupt() one thread can break the sleeping or waiting state of that thread and throws an exception known as Interrupted Exception.
- If a thread is not in a sleeping or waiting state calling the Interrupt() performs the normal behaviour & set the interrupt flag bit as 1 for true.

//A program to show the interruption of thread

```
class test extends Thread {
public void run() {
try { Thread.sleep(2000);
System.out.println("Task  slept");
}
catch (InterruptedException  e) {
throw new RuntimeException("Thread interrupt…. ");
}
}
public static void main(String args[])
{
test obj=new test; Obj.start();
try { Obj.interrupt(); }
catch(Exception e) {
System.out.println("Exception handled");     }
}
}
```

Java Inner Classes

- Java inner class or nested class is a class which is declared inside the class or interface.
- We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.
- Additionally, it can access all the members of outer class including private data members and methods.

Syntax of Inner class

```
class Java_Outer_class{
 //code
 class Java_Inner_class{
  //code
 }
}
```

Advantage of java inner classes

There are basically three advantages of inner classes in java. They are as follows:

1) Nested classes represent a special type of relationship that is it can access all the members (data members and methods) of outer class including private.

2) Nested classes are used to develop more readable and maintainable code because it logically group classes and interfaces in one place only.

3) Code Optimization: It requires less code to write.

Difference between nested class and inner class in Java

- Inner class is a part of nested class. Non-static nested classes are known as inner classes.

Types of Nested classes

There are two types of nested classes non-static and static nested classes.The non-static nested classes are also known as inner classes.

- Non-static nested class (inner class)
  - Member inner class
  - Anonymous inner class
  - Local inner class
- Static nested class

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing interface or extending class. Its name is decided by the java compiler.
Local Inner Class	A class created within method.
Static Nested Class	A static class created within class.
Nested Interface	An interface created within class or interface.

i. Java Member inner class

A non-static class that is created inside a class but outside a method is called member inner class.

Syntax:

```
class Outer{
 //code
 class Inner{
  //code
 }
}
```

Java Member inner class example

In this example, we are creating msg() method in member inner class that is accessing the private data member of outer class.

```
class TestMemberOuter1{
 private int data=30;
 class Inner{
  void msg(){System.out.println("data is "+data);}
 }
 public static void main(String args[]){
  TestMemberOuter1 obj=new TestMemberOuter1();
  TestMemberOuter1.Inner in=obj.new Inner();
  in.msg();
 }
}
```

Output: data is 30

ii. Anonymous inner classes:

Anonymous inner classes are declared without any name at all. They are created in two ways.

a. As subclass of specified type

```
class Demo {
   void show() {
      System.out.println("i am in show method of super class");
   }
}
class Demo1 {

   //  An anonymous class with Demo as base class
   static Demo d = new Demo() {
       void show() {
           super.show();
           System.out.println("i am in Flavor1Demo class");
       }
   };
   public static void main(String[] args){
       d.show();
   }
}
```

Output

```
i am in show method of super class
i am in Demo1 class
```

In the above code, we have two class Demo and Demo1. Here demo act as super class and anonymous class acts as a subclass, both classes have a method show(). In anonymous class show() method is overridden.

b. As implementer of the specified interface

```
class Demo2 {
     // An anonymous class that implements Hello interface
    static Hello h = new Hello() {
        public void show() {
            System.out.println("i am in anonymous class");
        }
    };
  public static void main(String[] args) {
        h.show();
```

```
    }
}
interface Hello {
    void show();
}
```

Output: i am in anonymous class

In above code we create an object of anonymous inner class but this anonymous inner class is an implementer of the interface Hello. Any anonymous inner class can implement only one interface at one time. It can either extend a class or implement interface at a time.

iii. Java Local inner class

- A class i.e. created inside a method is called local inner class in java. If you want to invoke the methods of local inner class, you must instantiate this class inside the method.

Java local inner class example

```
public class localInner1{
 private int data=30;//instance variable
 void display(){
  class Local{
   void msg(){System.out.println(data);}
  }
  Local l=new Local();
  l.msg();
 }
 public static void main(String args[]){
  localInner1 obj=new localInner1();
  obj.display();
 }
}
```

Output: 30

Rules for Java Local Inner class

1) Local inner class cannot be invoked from outside the method.

2) Local inner class cannot access non-final local variable till JDK 1.7. Since JDK 1.8, it is possible to access the non-final local variable in local inner class.

Example of local inner class with local variable

```
class localInner2{
 private int data=30;//instance variable
 void display(){
  int value=50;//local variable must be final till jdk 1.7 only
  class Local{
   void msg(){System.out.println(value);}
  }
  Local l=new Local();
  l.msg();
 }
 public static void main(String args[]){
  localInner2 obj=new localInner2();
  obj.display();
 }
}
```

Output: 50

iv. Java static nested class

- A static class i.e. created inside a class is called static nested class in java. It cannot access non-static data members and methods. It can be accessed by outer class name.
- It can access static data members of outer class including private.
- Static nested class cannot access non-static (instance) data member or method.

Java static nested class example with instance method

```
class TestOuter1{
  static int data=30;
  static class Inner{
   void msg(){
        System.out.println("data is "+data);
        }
  }
  public static void main(String args[]){
  TestOuter1.Inner obj=new TestOuter1.Inner();
  obj.msg();
  }
}
```

Output: data is 30

- In this example, you need to create the instance of static nested class because it has instance method msg(). But you don't need to create the object of Outer class because nested class is static and static properties, methods or classes can be accessed without object.

Java static nested class example with static method

- If you have the static member inside static nested class, you don't need to create instance of static nested class.

```
class TestOuter2{
  static int data=30;
  static class Inner{
   static void msg(){
        System.out.println("data is "+data);
        }
  }
  public static void main(String args[]){
  TestOuter2.Inner.msg(); //no need to create the instance of static nested class
  }
}
```

Output: data is 30