JAVA PROGRAMMING :

UNIT-I :

Introduction to OOP, procedural programming language and object oriented language, principles of OOP, applications of OOP, history of java, java features, JVM, program structure.

Variables, primitive data types, identifiers, literals, operators, expressions, precedence rules and associativity, primitive type conversion and casting, flow of control.

UNIT-II :

Classes and objects, class declaration, creating objects, methods, constructors and constructor overloading, garbage collector, importance of static keyword and examples, this keyword, arrays, command line arguments, nested classes.

UNIT-III :

Inheritance, types of inheritance, super keyword, final keyword, overriding and abstract class. Interfaces, creating the packages, using packages, importance of CLASSPATH and java.lang package. Exception handling, importance of try, catch, throw, throws and finally block, user-defined exceptions, Assertions.

UNIT-IV :

Multithreading: introduction, thread life cycle, creation of threads, thread priorities, thread synchronization, communication between threads. Reading data from files and writing data to files, random access file.

UNIT-V :

Applet class, Applet structure, Applet life cycle, sample Applet programs.

Event handling: event delegation model, sources of event, Event listeners, adapter classes, inner classes.

UNIT-VI :

AWT: introduction, components and containers, Button, Label, Checkbox, Radio Buttons, List Boxes, Choice Boxes, Container class, Layouts, Menu and Scrollbar.

UNIT-I

Introduction to OOP:

* Object Oriented Programming (OOP) is one of the most useful innovations in software development. It has strong programming paradigms and practices.
* In the early days of software development, many software projects have failed due to the following reasons:

1. Over budget
2. Over scheduled
3. Fail to meet the client's requirement
4. Faults or bugs in the software

This problem is termed as 'Software Crises'.

* OOP address this problem called software crises.
* Also software development has become much complex due to the enormous changes in the technology. OOP could handle the complexicity of software development.
* OOP is a programming paradigm which deals with the concepts of 'Object' to build software applications.
* OOP is modelled around the real world which is full of objects.
* Every object has a well-defined identity,

structure (attribute), behaviour (Operations). These objects exhibit the same behaviour in programming

* Some problems are associated with the earlier programming approaches and to overcome these problems, this new programming paradigm OOP was required.

Procedural Programming Languages and Object Oriented Programming Languages (POP and OOP):

* FORTRAN, PASCAL, COBOL, C are some of the procedural languages. When we write a large program using a procedural language, the complexicity increases and become difficult to maintain it. There are 2 main problems in procedural languages.

1. The functions have unrestricted access to global data.
2. Poor mapping to the real world.

* Also procedural languages lack the support for extensibility (Creating own data types).

* The following table presents the major differences between POP and OOP.

POP	OOP
1, Separate data from functions that operate on them	1, Encapsulate data and methods in a class.
2, Not suitable for defining abstract types.	2, Suitable for defining abstract types.
3, Debugging is difficult.	3, Debugging is easy.
4, Difficult to maintain programs.	4, Easy to maintain programs.
5, Not suitable for larger applications.	5, Suitable for larger applications.
6, POP programs work faster	6, OOP programs work slower.
7, Less resuable	7, More reusable
8, Less flexible	8, More flexible
9, Uses top-down approach.	9, Uses bottom-up approach
10, Data and procedure based.	10, Object oriented
11, Analysis and design is not so easy.	11, Analysis and design made easier.
12, One function calls another.	12, Communication among objects is done.
13, BASIC, FORTRAN, PASCAL, COBOL, care some POP languages.	13, C++, Java are some OOP languages.

Principles of OOP:

* All object oriented programming languages provide mechanisms (or) principles that help us to impliment object oriented model. There are 4 fundamental principles of OOP:

1. Abstraction
2. Encapsulation
3. Inheritance
4. Polymorphism

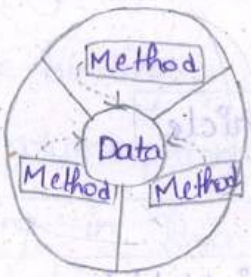
1. Abstraction:

* An essential element of object oriented programming is 'abstraction' (Data abstraction).
* We can handle the complexicity through abstraction
* For instance, consider a man who is driving a car. He doesn't know how engine works, how brakes work or how transmission happens etc., He even doesn't know about each and every part of the tens of hundreds of parts of the car. If he is overwhelm with all the implementation details of the car then he can't enjoy driving it.
* If the low-level details are hidden from the user, the user can see the system as a whole as a single object and can manage it well.
* The same can be applied to the programming also.

* We manage complexicity by concentrating only on the essential features and hiding the implementation details.
* Thus the process of highlighting the essential characteristics and suppressing the implementation(or) low-level details is known as 'abstraction.'
* This abstraction can be done at many levels.

2. Encapsulation (Data hiding):

* The process of binding data and procedures/functions/methods together as a single unit is called 'Encapsulation'.
* Encapsulation keeps both data and methods safe from outside interference and misuse.
* It restricts anyone from directly altering the data.
* The following figure shows how encapsulation protects data from misuse:



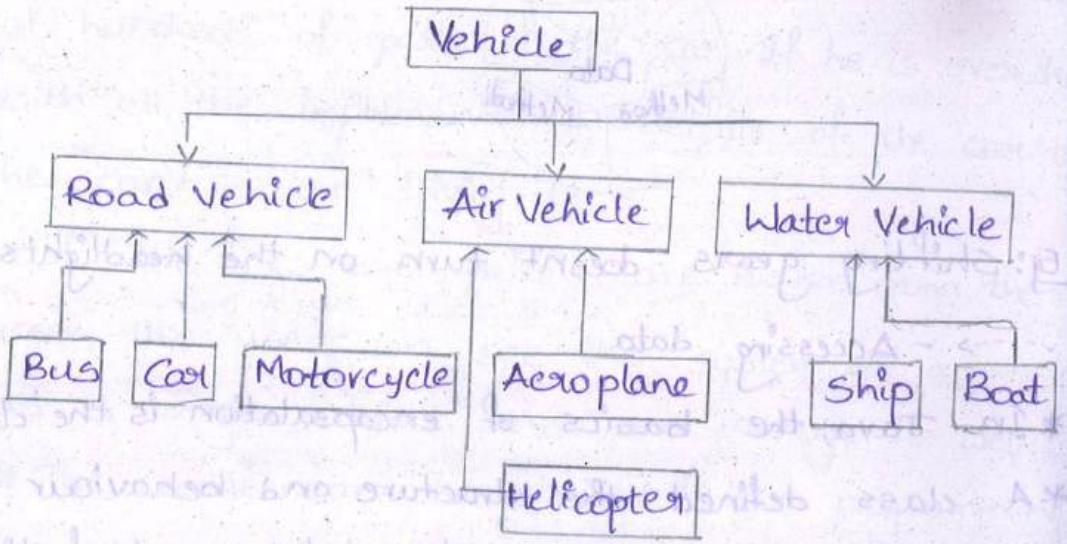
Eg: Shifting gears doesn't turn on the headlights.

----> - Accessing data

* In Java, the basics of encapsulation is the 'class'.
* A class defined the structure and behaviour (data & code) that will be shared by a set of objects.

3. Inheritance:

* Inheritance is the process by which one object requires the properties of another object.
* In this scenario, a parent child relationship exists among the classes.
* When a class inherits another class, it has all the properties of the base class and it adds some new properties of its own.
* For example, we can classify the vehicles into three classes: Road vehicles, Air vehicles & Water vehicles.
* All the three classes of vehicles have some common things like engine, speedometer etc., But each class has its own, unique features like motor bikes have disk braking system and planes have hydraulic braking system.
* The following figure shows inheritance process in real world:



* Inheritance promotes reusability.
* When a class is created, it can be distributed to other programmers so that they can use it in their programs.

4. Polymorphism :

* 'Polymorphism' is one of the important features of an object-oriented programming language.
* Polymorphism simply means 'having many forms'.
* This feature allows one interface to be used for a general class of actions. i.e., Same thing being used in different forms.
* Method overloading & Method overriding are the mechanisms where polymorphism is used in Java.
* There are two types of polymorphism.
  1. Compile-time polymorphism
  2. Run-time polymorphism
* Compile-time polymorphism is also known as early binding or static binding.
* Compile-time polymorphism is implemented by method overloading in Java.
* In case of compile time polymorphism, compiles will determine which method (among all overloaded methods) will be executed.
* Run-time polymorphism is also known as dynamic binding or late binding.

* Method overriding is a way to implement runtime polymorphism in Java.
* Run-time binding is used to determine which metho to invoke at runtime.

Applications of OOP:

* Object oriented Programming has become an standard for programming.
* OOP provides better flexibity and compatibility for developing large & complex applications.
* Applications of OOP are begining to gain importance in many areas.
* Real-business systems are often much more comple and contain many more objects with complicated attributes and methods.
* OOP is useful in these types of applications as it manages complexity.
* The promising areas where OOP techniques are applica include:
  - -> Real-time systems
  - -> Artificial Intelligence
  - -> Neural networks
  - -> Simulation & Modeling
  - -> Object Oriented & Distributed databases
  - -> Decision support systems & Expert systems
  - -> CAD/CAM systems
  - -> Client-server systems

→ Parallel Programming

History of Java:

* The innovation and development of a computer language occurs for two fundamental reasons:

1. To adapt to changing environments and uses.
2. To implement refinements and improvements in the art of programming.

* The development of Java was driven by both the elements in nearly equal measure.
* Much of the character of Java is inherited from C and C++.
* Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank and Mike Sheridan at Sun Microsystems in 1991.
* It took 18 ~~meth~~ months to develop the first working version of Java.
* This language was initially named 'Oak'. Unfortunately, this name had already been patented by some one else.
* So, it was renamed 'Java' in 1995.
* Originally Java was designed to run standalone in small embedded systems.
* In embedded systems such as microwave ovens and remote controls, many different types of CPUs are used as controllers.
* The trouble with C & C++ is that they are

designed to be compiled for a specific target.

* The problem is that compilers are expensive and time consuming to create.

* So Gosling and others began work on a portable, platform-independant language. This effort led to the creation of Java.

* World Wide Web played a very crucial role in the future of Java.

* With the emergence of the world wide web, Java was pushed forward as the web also needed/demanded portable programs.

* Java is not an extension or replacement of C++

* Internet helped Java to become popular as a computer programming language and Java inturn had a profound effect on the internet.

* Java succeeded in addressing the two important challenges of the Internet based applications: Security and portability.

* The latest version is Java 9 released on septembe 21, 2017.

* Major versions of Java and their release dates are given below:

→ JDK 1.0	Jan 23, 1996
→ JDK 1.1	Feb 19, 1997
→ J2SE 1.2	Dec 8, 1998
→ J2SE 1.3	May 8, 2000

→ J2SE 1.4 Feb 6, 2002

→ J2SE 5.0 Sep 30, 2004

→ Java SE 6 Dec 11, 2006

→ Java SE 7 Jul 28, 2011

→ Java SE 8 Mar 18, 2014

→ Java SE 9 Sep 21, 2017

→ Java SE 10 Mar 2018

→ Java SE 11 Sep 2018

→ Java SE 12 Mar 2019

→ Java SE 13 Sep 2019

Features of Java (Java Buzz words):

*The features that made Java, a powerful programming language are listed below:

- Simple
- Secure
- Portable
- Object-Oriented
- Robust
- Multi threaded
- Architectural-neutral
- Compiled & Interpreted
- High performance
- Distributed
- Dynamic

Simple:

*Java was designed in such a way that a programming professional can easily learn and use.
*With some prior programming experience, you won't feel hard to master Java.
*A c++ programmer needs a little effort to learn and use. Because Java inherits the syntax and style of c and c++.

Object-Oriented:

*Java is a pure object oriented language. That means the outermost level of data structure in Java is the object.
*Everything in Java (variables, constants, methods) are defined inside a class and accessed through objects.
*The object model of Java is very simple and easy to extend.
*Primitive types (integers, float etc.,) are kept an objects to achieve high performance.

Secure:

*Downloading a program from the internet is always suspectable.
*Some active programs can contain some malicious code and when we download then they can damage the resources of old computers.
*Java innovated a new type of returned program called 'applet'. This applet is a small Java program that can be downloaded from the internet and it can

access the other parts of the computers.
*So applets can be downloaded with the confidence that no harm will be done to your computer and that no security will be breaked/broken.

Portable:

*Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it.
*For example, the same applet can be downloaded by many users across the networks. All these were use different types of computers, processors and Oss.
*the code of the applet must work on all the system. One solution for this is to develop a separate version of applet for each type of the computer. But this solution is impractical.
*So some means of generating portable executable code is needed.
*Java's bytecode is the solution for this problem.
*Bytecode addresses two problems i.e.) security and portability.
*Bytecode is the non executable code produced by (output of) Java compiler and this can be executed by the Java run-time environment called JVM (Java Virtual Machine).
*The JVM needs to be implemented for each platform.

Robust:-

*Programs written in Java are highly reliable and robust.
*Since Java is strictly typed, compile time and run time

checks can find many errors in the program.
* There are 2 reasons to say Java is a robust language that has even evolved:
i) Memory management
ii) Exception handling
* Memory management is a tedious task. Java manages allocation and deallocation of memory in an outstanding way. The deallocation of memory in Java is completely automatic (Automatic garbage collection).
* Java also provides object oriented exception handling mechanism to handle runtime errors.

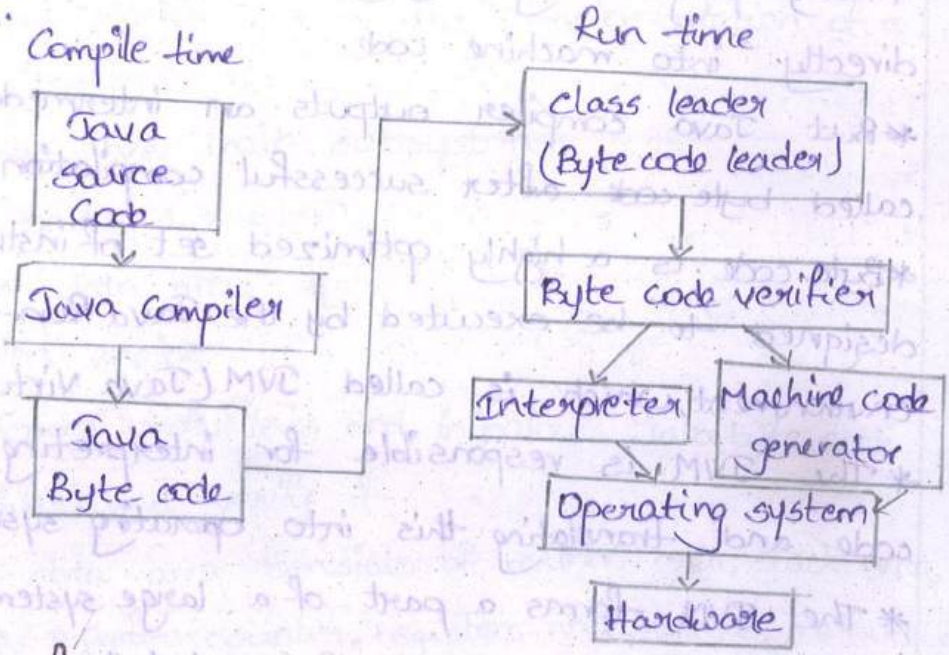
Multi-threaded :
* Java supports multithreading which occurs you to build programs that do many things simultaneously.
* A thread is an independent part of execution of a program. A program can be viewed as collection of threads and all these threads can be executed concurrently.
* Java's approach of multithreading is easy to use.

Architectural-neutral :
* A programmer cannot guarantee his program even on the same machine to execute/run successfully every time. The program successfully executed today may malfunction tomorrow even on the same machine.
* Due to technological advancements, operating systems change, processor changes/upgrades, core system resource may upgrade. So the programs may become outdated.
* So the goal is to "write once, run anywhere, anytime forever".

* Java achieved this goal to a greater extent.
* Java's byte code is the reason for this platform independence of Java.

Compiled and Interpreted:

* Java incorporates both compilation and interpretation.
* First the program/source code is compiled to the intermediate code called 'byte code.'
* Then JVM (Java Virtual Machine) interpretates the byte code instructions.
* The byte code can run on any platform on which a JVM has been displayed.
* The following figure shows compilation and interpretation in Java.



High performance:

* Interpretation of byte code slowed performance in the earlier versions, but advanced JVMs with adaptive optimization and just-in-time compilation provide high speed code execution.

Distributed :

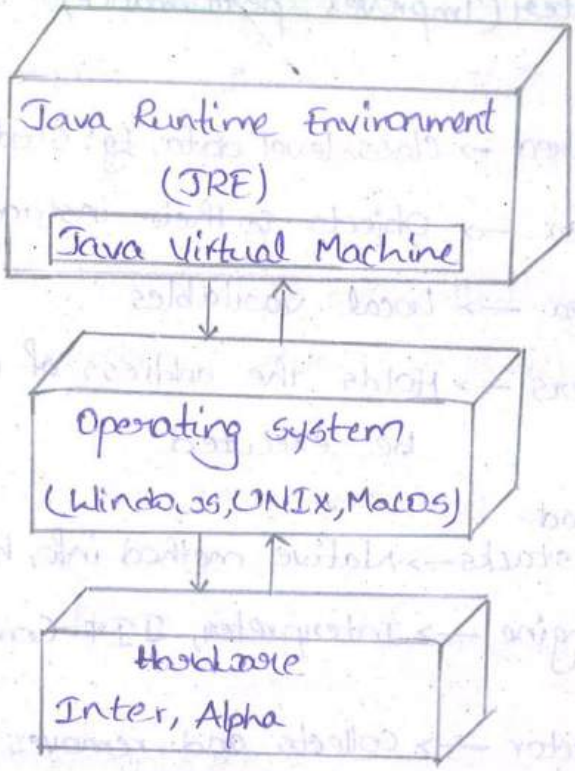
* Java is designed to work for the distributed environment of the internet it handles TCP/IP protocols.
* Java supports RMI (Remoted Method Innovation) which enables a program to invoke methods across a network.

Dyramic :

* The access to the objects is recovered at runtime dynamically because the Java programs carry substantial runtime type information with them.
* Linking is done one-demand (dynamically) in Java.
* New classes can be loaded at runtime.

JVM (Java Virtual Machine) :

* Many programming languages compile the source code directly into machine code.
* But Java compiler outputs an intermediate code called byte code after successful compilation.
* Byte code is a highly optimized set of instructions designed to be executed by the Java Run-time environment, which is called JVM (Java Virtual Machine).
* The JVM is responsible for interpreting Java byte code and translating this into operating system calls.
* The JVM forms a part of a large system called JRE.
* JRE varies according to the underlaying operating system and computer architecture.
* The following figure shows how JVM works:



Note:

*A virtual machine is a slow implementation of a physical machine.

*JVM has three main subsystems:

1. Class loader subsystem
2. Runtime data area
3. Execution engine

*Class loader loads, links and initializes Java byte code (.class files) and verify it.

*Runtime data area consists of method area, stack area, heap area, program counter, registers and native method stacks.

*The execution engine reads the bytecode assigned to the runtime data area and executed it. The execution engine contains interpreter and JIT compiler.

*The Java interpreter executes the byte code line by line and the just-in-time compiler neutralizes the disadvantages

of interpreter (Improves performance).

Note :

Method area → Class-level data, Eg: Static variable

Heap area → Objects & their instance variables

Stack area → Local variables

Pc registers → Holds the address of next instruction to be executed

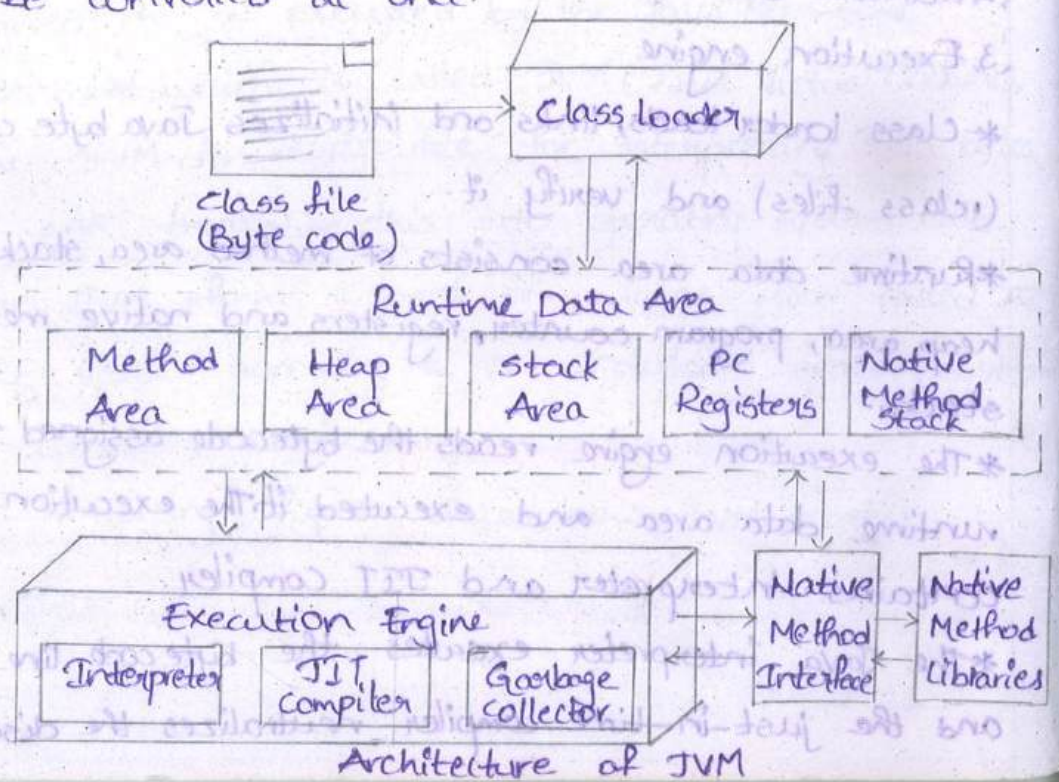
Native method stacks → Native method info, built-in methods

Execution engine → Interpreter, JIT Compiler & garbage Collector

Garbage Collector → Collects and removes unreferenced objects [System.gc()]

*JVM is responsible for interpreting Java byte code and executing it with the help of operating system and hardware.

* JIT helps for faster execution, repeated instructions can be converted at once.



Architecture of JVM

Structure of a Java program:

*A Java program contains collection of classes. A class is a template that defines structure and behaviour for its instances.

*The instances of a class are called objects. Each object contains the members (fields & methods) specified in the class.

*A field is one that holds data and a method defines operations on the fields.

*The following figure shows the structure of a Java program:

```
/* comments */
class className1 {
    datatype instance_var_Name1;
    datatype instance_var_Name2;
    :
    datatype methodname1 (parameter_list) {
        /* method definition */
    }
    datatype methodname2 (parameter_list) {
        /* method definition */
    }
    :
}
class className2 {
    :
}
```

*Let us write a simple Java program that prints "Hello World!" on the screen:

```
1, /* This is my first Java program */
2, /* Call this program as Hello.java */
3, class Hello{
4, public static void main(String args[ ]){
5, System.out.println("Hello World!");
6, }
7, }
```

Executing a Java program:

There are 4 easy steps to successfully execute a Java program:

1, Creating Source file: Open a text editor like notepad & type the above Java source code as shown in the above example.

2, Save the source file: Save the file with the name & java extension (Eg: Hello~~.java~~). The name of the java file must watch with the name of the class in which main() method is defined.

3, Compile the source code file: To compile the source file (.java file), use the javac tool at the command prompt:

```
C:\>javac programName.java
```

Eg: C:\>javac Hello.java

Once you successfully compiled the .java file, a ".class" file will be created (Eg: Hello.class). This class file contains the java bytecode of the source file.

4. Run the .class file: To run the java program you have to use the 'java' tool at the command prompt:

C:\> java programName

Eg: C:\> java Hello

When your program is successfully executed, the following output is produced:

Hello World!

Explanation of 'Hello.java' program:

* The lines 1 & 2 of the program represent the comments. Java supports both single line comments (//) and multiline comments (/* ..... */).
* Line 3 represents the begining of the class definition. 'class' is the keyword used to denote class definition and 'Hello' is the name of the class.
* Line 4 denotes the definition of the main() method inside the class. The general format for main() in java is as shown below:

public static void main(String args[])

1. The keyword 'public' is an access specifier, which allows the program to control the visibility of the class members. When a class member is defined 'public', that member may be accessed by code outside the class in which it is declared.

The main() must be declared as 'public'. Since it must be called by code outside of its class when the program is executed.

2. The keyword 'static' allows main() to be called

without having it instantiate a particular instance of the class. This is necessary because main() is called by the JVM before any objects are made.

3. The keyword 'void' simply means that the main() method doesn't return any value to the calling environment.

4. The main() method accepts only one parameter i.e., an array of string objects. Here 'args' is the name of the array of instances of string class.

* Line 5 is an executable java statement defined inside the main() method:

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

It is similar to the printf() statement in 'C'. Here 'system' is the name of the predefined class that provides access to the system and 'out' is the output stream that is connected to the console.

println() is a method used to display the output on the console.

Primitive data types:

* Java defines 8 primitive data types byte, short, int, long, float, double, char & boolean.
* Primitive types are also referred to as simple types on built-in types.
* These primitive data types can be put into 4 groups:

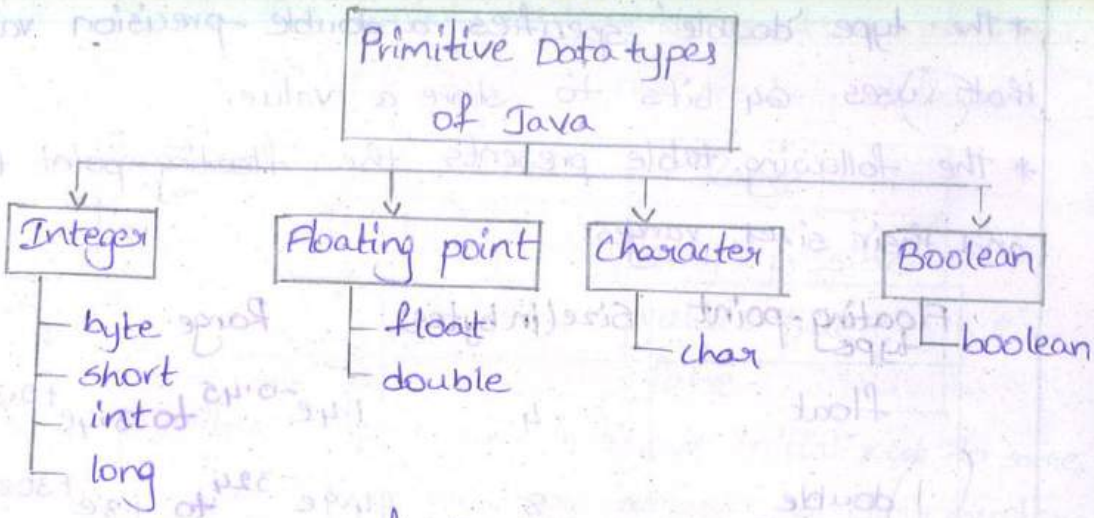


fig: Primitive types of Java

1. Integers :

* This group includes byte, short, int and long which are whole-valued signed numbers.
* Java doesn't support unsigned integers.
* The following table presents the integer types and their ranges:

Integer type	Size (in bytes)	Range
byte	1	-128 to 127
short	2	-32,768 to 32,767
int	4	-2^{31} to $2^{31}-1$
long	8	-2^{63} to $2^{63}-1$

2. Floating-point types:

* This data types is for floating-point numbers or real numbers.
* There are 2 kinds of floating point number types: float and double.
* The type 'float' specifies a single precision value that uses 32 bits of storage.

* The type 'double' specifies a double-precision value that uses 64 bits to store a value.
* The following table presents the floating-point types and their sizes, ranges:

Floating-point type	Size (in bytes)	Range
float	4	$1.4e^{-0.45}$ to $3.4e^{+0.38}$
double	8	$4.9e^{-324}$ to $1.8e^{+308}$

3. Character :

* In Java, the data type used to store characters is 'char'. The 'char' in Java is not same as 'char' in C or C++.
* In C or C++, 'char' type is 8 bits wide as they uses ASCII character set.
* But Java uses the unicode character set. So char in Java is 16 bits wide.
* The range of char is 0 to 65,535 Or it ranges from '\u0000' to '\uffff'.

Character type	Size (in bytes)	Range
char	2	0 to 65,535

4. Boolean type :

* Java has one primitive type for logical values called 'boolean'.
* It can have only one of two possible values 'true' or 'false.'
* This type is returned by all relational operators.
* Boolean type is required by the conditional expressions

that govern the control statements such as if & for.
* The size of this type is not precisely defined.

Boolean type	Size	Range
boolean	not precisely defined	Only 2 possible values true & false

* Java primitive types are ~~initiated~~ initialized to some default values when they are declared as class members.
* The primitive types and their default values are shown below:

Data type	Default value
byte	0
short	0
int	0
long	0
char	'\u0000'
float	0.0f
double	0.0d
boolean	false

Identifiers :

* Identifiers are the set of valid strings which are allowed as names in a computer language.
* Simply an identifier is a name used to identify a variable, constant, class, method, package or an interface in Java.
* There are some rules for Java identifiers:
  1. The first character of an identifier must be a letter, an underscore or a dollar($) sign.

Type of literal	Example

Numeric literals
• Binary literals
• Octal literals
• Hexadecimal literals
• Integer literals
• Floating point literals
	
	char bin = 0b1010;
	int x = 0150;
	int y = 0x458;
	long l = 25637L;
	float f = 2.36F;
	double d = 23.6;

character literals	char c = 'A'; char c1 = '\u0041'; char c2 = '\t';
Boolean literals	boolean flag = false; boolean isMine = true;
String literals	String s = "This is a string";
Null literals	Obj = null;

Note:
There are 3 reserved literals in Java - true, false null

Variable:

1. Numeric Literals:

* Numeric literals are the numeric values assigned variables or constants. They are again divided into t following types:

* Binary literals, octal literals, hexadecimal literals, inte literals, floating point literals.

1. Binary literals are a combination of 0's and 1's.

Binary literals can be assigned to in Java. They must be prefixed with 0b or 0B.

Eg: int b1=0B11001101

char b=0b1010

2, Octal literals must be prefixed with a '0' and only the digits from 0 to 7.

Eg: int x=025;

long y=023421;

3, Hexadecimal literals are prefixed with '0x' or '0X'. The digits 0 to 9 and A to F (a to f) are only allowed.

Eg: int a=0X23F;

long b=0x54A3;

long l=53L;

4, Integer literals are of 'int' type by default. To define them as 'long' we can place suffix 'L' or 'l' after the number.

Eg: int speed=78;

long miles=235L;

long x=568932l;

5, All floating point literals are of type 'double' by default. To define them as float literals, we have to suffix them 'F' or 'f'.

Eg: float sal=39.6F;

double y=26.246;

2, Boolean literals:

*A boolean literal is specified as either 'true' or 'false'. By default it takes the value 'false'.

Eg: boolean isMine = ~~false~~true;

boolean flag = false;

3, Character literals:

*A single character enclosed in single quotes (' ') is a char literal. We can also use prefix 'u' followed by four hexadecimal digits representing the 16-bit unicode character.

Eg: char c1 = 'B';

char c2 = '\u0040';

char c3 = '\u00ff';

Note:

* A single quote(' '), a black slash or a unprintable character can be specified as a literal with the help of escape sequence.

*An escape sequence represents a character by using special syntax that begin with a single backslash character (\).

Some special escape sequences are given below:

Escape sequence	Meaning
\\	Backslash
\'	Single quotation mark
\"	Double quotation mark
\t	Horizontal tab
\n	New line character
\r	carrage return
\f	Form feed character

The following p·table presents some unicode escape sequences:

Unicode Escape Sequence	Meaning
'\u0041'	Capital letter A
'\u0030'	Digit zero (0)
'\u0022'	Double quotation mark
'\u0020'	Widespace character
'\u0009'	Horizontal tab

4, String literals:

*String literals contain zero or more characters within double quotes.

Eg: String s = "This is a string";

String s1 = " ";

5, Null literals:

*Null literals are assigned to the object reference variable.

Eg: obj = null;

Variables:

*A variable is a named memory location used to store value that can change during the execution of a program.

*Java variables are declared using the following syntax:

```
datatype identifier = literal;
        (or)
datatype variable_name = value;
```

where

datatype may be any of the primitive types of Java.

*Identifier is the name used to identify the variable.

*Literal is the value that is assign to the variable

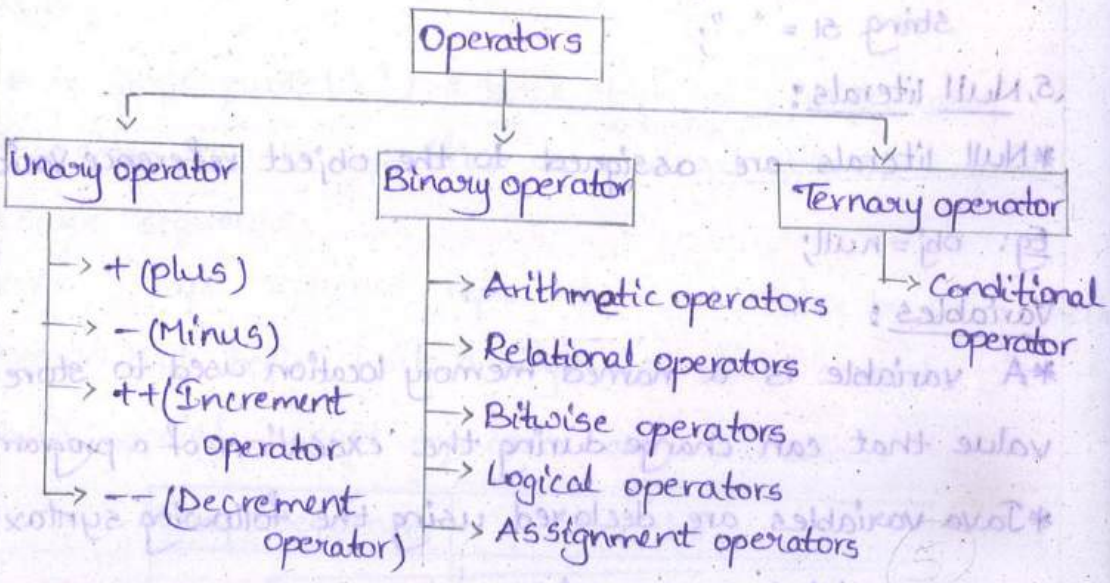
Eg: int var = 25;

```
float a = 34.9;
char c = 'M';
```

Operators:

* An operator is a symbol that is used to perform an operation on one or more operands.
* An operator that performs an operation on one operand is called a unary operator.
* An operator that performs an operation on two operands is called a binary operator.
* An operator that performs an operation on three operands is called a ternary operator.

Java supports these three types of operators.



1. Unary Operator:

* Unary operator has single operand.

i) + (Unary plus):

This operator represents the positive value of arithmetic operand.

Syntax: +[operand]

Eg:

```
int a;
a = +10;
```

ii) - (Unary minus):

This operator represents the negative value of arithmetic operand.

Syntax:

```
-[operand]
```

Eg:

```
int a;
a = -10;
```

iii) Increment and Decrement Operators:

* These unary operators are used to increment value of the operand with one or decrement value of the operand with one.
* There are mainly 2 types of increment & decrement operators.

Increment types	Decrement types
1, Pre Increment	1, Pre Decrement
2, Post Increment	2, Post Decrement

Increment → Pre Increment
Increment → Post Increment

Pre Increment: ++x → x+1 = x (Increment)
x = x (Assignment)
Increment then assignment

Post Increment: x++ → x = x (Assignment)
x = x+1 (Increment)
Assignment then increment

Eg: Post and Pre incrementing variable

```
class PostIncrement{
    public static void main(String args[]){
        int num1=1;
        int num2=100;
        num1++;
        ++num2;
        System.out.println("Num1= "+num1);
        System.out.println("Num2= "+num2);
    }
}
```

Output:

```
Num1=2
Num2=101
```

Decrement → Pre Decrement
Decrement → Post Decrement

Pre Decrement: --x → x-1=x
Pre Decrement: --x → x=x

Post Decrement: x-- → x=x
Post Decrement: x-- → x=x-1

Eg:

```
class JavaProgram{
    public static void main(String args[]){
        int a=1;
        int b=2;
        int c;
        int d;
        c=--b;
```

```
        d=a--;
        c--;
        System.out.println("a= "+a);
        System.out.println("b= "+b);
        System.out.println("c= "+c);
        System.out.println("d= "+d);
    }
}
```

Output:

a=
b=
c=
d=

2. Binary Operator:

Binary operator contains two operands and one operator.

i) Arithmetic operators:

The arithmetic operators perform various mathematical operations.

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder

Eg:

```
class Test{
    public static void main(String args[]){
        int a=10;
        int b=20;
        int c=25;
        int d=25;
```

```
        System.out.println("a+b = " +(a+b));
        System.out.println("a-b = " +(a-b));
        System.out.println("a*b = " +(a*b));
        System.out.println("b/a = " +(b/a));
        System.out.println("b%a = " +(b%a));
    }
}
```

Output:

a+b = 30

a-b = -10

a*b = 200

b/a = 2

b%a = 0

ii) Relational Operators :

Relational operators are used to compare the numerical values. The following table illustrates the relational operators in Java:

Operator	Description
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Is equal to
!=	Not equal to

Eg:

```
class Test{
    public static void main(String args[ ]){
        int a=10;
        int b=20;
```

```
        System.out.println("a==b= " + (a==b));
        System.out.println("a!=b= " + (a!=b));
        System.out.println("a>b= " + (a>b));
        System.out.println("a<b= " + (a<b));
        System.out.println("b>=a= " + (b>=a));
        System.out.println("b<=a= " + (b<=a));
    }
}
```

Output:

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

(ii) Bitwise Operators:

Bitwise operators works on bits & perform bit-by-bit operation.

The following table lists the bit-wise operators in Java:

Operator	Description
&	Bit-wise AND (a&b)
|	Bit-wise OR (a|b)
^	Bit-wise XOR (a^b)
~	One's complement (~a)
<<	Left shift (a<<2)
>>	Right shift (a>>2)
>>>	Unsigned Right shift

Eg:

```
class Test4
    public static void main(String args[ ]){
        int a=60;
        int b=13;
        int c=0;
        c=a&b;
        System.out.println("a&b= "+c);
        c=a|b;
        System.out.println("a|b= "+c);
        c=a^b;
        System.out.println("a^b= "+c);
        c=~a;
        System.out.println("~a= "+c);
        c=a<<2;
        System.out.println("a<<2= "+c);
        c=a>>2;
        System.out.println("a>>2= "+c);
        c=a>>>2;
        System.out.println("a>>>2= "+c);
    }
}
```

Output:

a&b = 12

a|b = 61

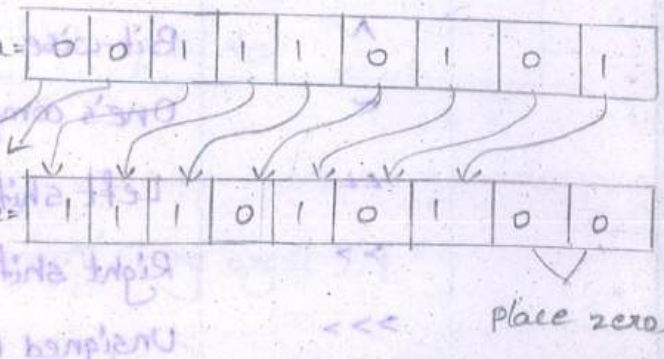
a^b = 49

~a = -61

a<<2 = 240

a>>2 = 15

a>>>2 = 15



(iv) Logical Operators:

* The logical operators provide a way to make decisions based on the values of multiple variables.
* Logical operators makes it possible to direct the flow of a program and are used frequently with conditional statements and loops.

Operator	Description
&&	Logical AND
||	Logical OR
!	Logical NOT

Eg:

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

Output:

a &&b=false

a||b=true

!(a &&b)=true

V) Assignment Operators:

* The assignment operators assign a data value to a variable.
* The simplest form of assignment operator just assigns some value, while other perform some other operation before making the assignment.

The following table lists the operators in Java:

Operator	Description
=	Simple assignment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
<<=	Left shift and assignment
>>=	Right shift assignment
&=	Bitwise AND assignment
^=	Bitwise exclusive OR assignment
|=	Bitwise inclusive OR assignment

Eg:

```
class Test{
    public static void main(String args[ ]){
        int a=10;
        int b=20;
        int c=0;
        c=a+b;
        System.out.println(" c=a+b= " +c);
        c+=a;
        System.out.println(" c+=a= " +c);
        c-=a;
        System.out.println(" c-=a= " +c);
        c*=a;
        System.out.println(" c*=a= " +c);
        a=10;
        c=15;
```

```
        c%=a;
        System.out.println("c%=a= "+c);
        c<<=2;
        System.out.println("c<<2= "+c);
        c>>=2;
        System.out.println("c>>=2= "+c);
        c&=a;
        System.out.println("c&=2= "+c);
        c^=a;
        System.out.println("c^=a= "+c);
        c|=a;
        System.out.println("c|=a= "+c);
    }
}
```

Output:

```
c=a+b=30
c+=a=40
c-=a=30
c*=a=300
c%=a=5
c<<2=
c>>2=
c&=2=
c^=a=
c|=a=
```

3. Ternary Operators:

i) Conditional operators:

* Conditional operator is also known as the ternary operators.
* This operator consists of 3 operands and is used to

evaluate boolean expressions.

* The goal of the operator is to decide which value should be assigned to the variable.

The operator is written as:

variable x = (expression)? value if true : value if false

Eg:

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

Output:

Value of b is:30

Value of b is:20

Expression:

An expression is a combination of operators and operands. Expressions may contains identifiers, types, literals, separators and operators.

Eg:

```
int x=20, y=30;
c=a*b+d/e;
```

Precedence and Associativity Rules:

* The precedence rules are used to determine the order of evaluation priority if there are two operators with different precedence.
* Associativity rules are used to determine the order of evaluation if the precedence of operators is same.
* Associativity is of two types.
1. Left to right (LR)
2. Right to left (RL)
* Precedence and associativity can be overridden with the help of paranthesis(( )).

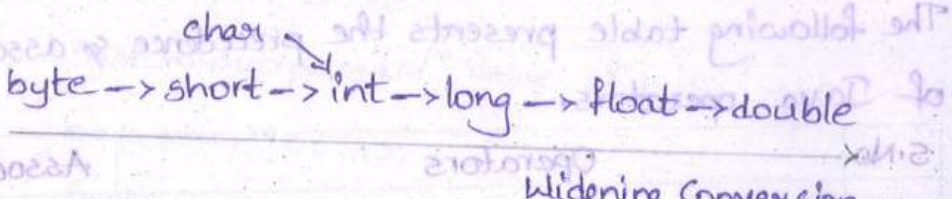
The following table presents the precedence & associativity of Java operators.

S.No.	Operators	Associativity
1.	., [ ], (args), i++, i--	LR
2.	++i, --i, +i, -i, ~, !	RL
3.	new, (type casting)	RL
4.	*, /, %	LR
5.	+, -	LR
6.	<<, >>, >>>	LR
7.	<, >, <=, >=, instanceof	Non associativity
8.	==, !=	LR
9.	&	LR
10.	^	LR
11.	|	LR
12.	&&	LR
13.	||	LR
14.	?:	RL
15.	=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=	RL

Type Conversion and Type Casting:

* Java supports automatic type conversions. When the type of the expression on the right hand side of an assignment operation can be safely promoted to the type of the variable on the left hand side of the assignment.
* the conversions which are implicitly in nature are called type conversion (or) Widening conversion.
* For converting one type to another, the types of the left and right hand side must be compatable.

The following figure shows type conversion (or) promotion of types in Java.



* For example int, byte, short & char can all fit inside int.
* double, byte, short, char, int, long, float can all fit inside double.

Eg: float x=2.6;
double y=x;

* Type conversion (or) Promotion takes place while evaluating expression for arithmetic operators.

Eg:
```
int a=10;
float c;
c= a/3.0f;
```

Type Casting:

Consider the following statement

```
byte b=10;
short s=30;
b = b+s; //invalid
```

Here the result of 'b+s' is a value of short type. So it cannot be implicitly converted into a byte type.

In such a case we have to use mechanism called type casting.

* Type casting is known as narrowing conversion or demotion.

double ← float ← long ← int ← char ← short ← byte

Narrowing Conversion

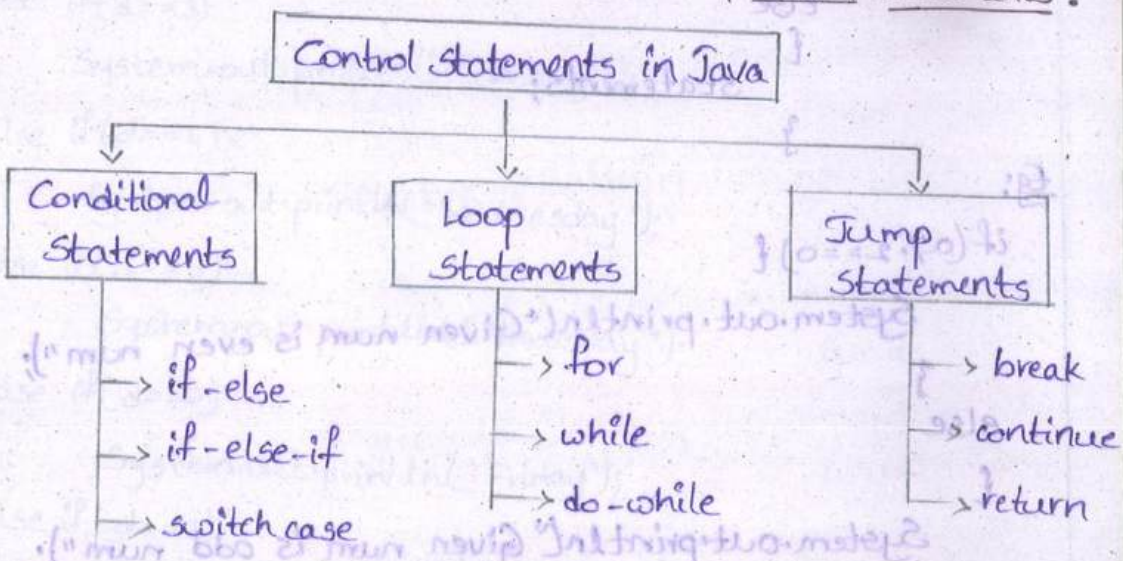
Eg:

```
b=(byte)b+s; // valid
int i=(int)8.0/3.0;
```

* A casting operation tells the compiler to cast the value before the assignment.

For example, a double value is being casted to an int value. The fractional part is truncated resulting in an integer.

Flow of control / Control flow statements / Control statements:



Conditional statements:

1. if statement:

The first contained statement, if statement only executes when the specified condition is true.

Syntax:

```
if(condition){
        Statements;
}
```

Eg:

```
if(a%2==0){
    System.out.println("Given num is even num");
}
```

2. if-else statement:

In if-else statement, if the specified condition in the if statement is false, then the statement after the else keyword will execute.

Syntax:

```
if(condition){
        Statements;
}
else
{
    Statements;
}
```

Eg:

```
if(a%2==0){
    System.out.println("Given num is even num");
}
else
{
    System.out.println("Given num is odd num");
}
```

3. if-else-if:

This statement following the else keyword can be another if or if-else statement.

Syntax:

```
if(condition)
    statements;
else if(condition)
    statements;
else if(condition)
    statements;
else
    statement
```

Whenever the condition is true, the associated statement will be executed and the remaining conditions will be passed. If none of the conditions are true then the else block will execute.

Eg:

```
if(d==1)
    System.out.println("Sunday");
else if(d==2)
    System.out.println("Monday");
else if(d==3)
    System.out.println("Tuesday");
else if(d==4)
    System.out.println("Wednesday");
else if(d==5)
    System.out.println("Thursday");
else if(d==6)
    System.out.println("Friday");
else if(d==7)
    System.out.println("Saturday");
```

```
else
    System.out.println("Invalid input!");
```

4. Switch:

The switch statement is a multi-way branch statement. The switch statement of Java is is another selection statement of Java is another selection statement that defines multiple paths of execution of a program.

Syntax:

```
switch(control_expression){
  case expression 1: <statement>;
  case expression 2: <statement>;
  ;
  case expression n: <statement>;
  default : <statement>;
}
```

Eg:

```
switch(day)
case 1: System.out.println("Monday");
        break;
case 2: System.out.println("Tuesday");
        break;
case 3: System.out.println("Wednesday");
        break;
case 4: System.out.println("Thursday");
        break;
case 5: System.out.println("Friday");
        break;
case 6: System.out.println("Saturday");
        break;
case 7: System.out.println("Sunday");
        break;
```

```
default: System.out.println("Invalid entry");
        break;
}
```

Loop statements:

1. for loop:

The statements will executed until the condition become false.

Syntax:

```
for(initialization; condition; increment or decrement){
        statement;
}
```

Eg:

```
for(int i=1; i<=5; i++)
{
System.out.println("Cube of "+i+" is: "+i*i*i);
}
```

2. while loop:

Until the condition become false the loop will be executed.

Syntax:

```
while(expression){
        <statement>
}
```

Eg:

```
while(i<=5){
    System.out.println("Cube of "+i+" is: "+i*i*i);
    i++;
}
```

3. do-while loop:

First the do block statements are executed then the condition given in which statement is checked. So in this case, even the condition is false in the first attempt,

do block of code is executed atleast once.

Syntax:

```
do {
    <statement>
}while(expression);
```

Eg:

```
do
{
System.out.println("Cube of "+i+" is: "+i*i*i);
}while (i<=5)
```

Jump statements:

1. break:

The break statement is used for breaking the execution of a loop.

Syntax:

```
break;
```

Eg:

```
for (int i=1; i<=5; i++)
{
System.out.println("Cube of "+i+" is: "+i*i*i);
if (i==4)
break;
}
```

2. Continue:

This is a branching statement that are used in the looping statements to skip the current iteration of the loop and resume the next iteration;

Syntax:

```
continue;
```

Eg:

```
for(int i=1; i<=5; i++)
{
if(i==4)
continue;
System.out.println("Cube of "+i+" is: "+i*i*i);
}
```

3. return:

It is a special branching statement that transfers the control to the caller of the method. This statement is used to return a value to the caller method and terminates execution of method.

This has 2 forms: one that return a value and the other that cannot return. The returned value type must match the return type of method.

Syntax:

```
return;
```

Eg:

```
public static void Hello(){
    System.out.println("Hello "+welcome());
}
static String welcome(){
    return "user";
}
```

UNIT-II

Classes:

* Class and object are the basic building blocks of object oriented programming.
* A class is a blue print that defines a no. of variables and methods common to all objects of a specific kind.

(or)

* A class is a logical template that defines the structure and behaviour of a specific type of objects.

(or)

* A class can be viewed as a user defined data type and an object is a variable of that user defined type.

Objects:

* The object oriented programming paradigm revolves along objects. An object is an entity that encapsulates variables and methods that operate on those variables.
* If we consider a class as user defined datatype then we can a view an object as a variable of that datatype.
* For example, if we consider a class called Employee, which defines three variables EmpId, EmpName and Salary & two methods getEmpDetails( ) and setEmpDetails( ).
* The following figure shows visual representation of Employee class:

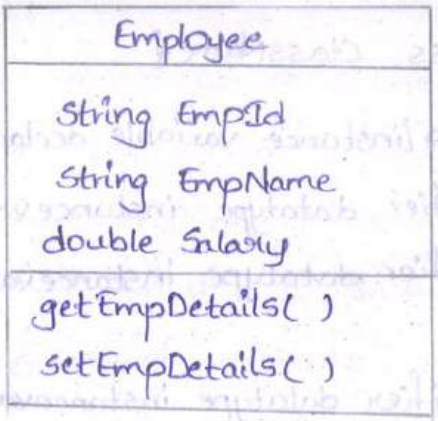


fig: Representation of class

* Since an object is an instance of a class, it can be visually represented as shown below:

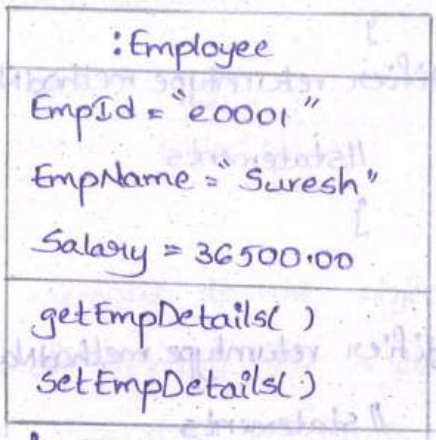


fig: Object representation

Declaration of a class:

* Declaring a class is very simple in Java. The "class" keyword helps us to declare a Java class.
* We define a set of data members (class/instance variable) and a set of methods inside a class.
* The following is the syntax for defining a class in Java:

```
modifier class className{
    //class instance variable declaration
    modifier datatype instancevar1;
    modifier datatype instancevar2;
    ⋮
    modifier datatype instancevarN;
    //method declaration
    modifier returntype methodName1(parameters list){
        //statements
    }
    modifier returntype methodName2(parameter list){
        //statements
    }
    ⋮
    modifier returntype methodNameM(parameter list){
        // statements
    }
}
```

Eg:

```
class Shape
{
  double length;
  double breadth;
}
```

The above code defines a class named Shape in which two instance variables are there.

Creating Objects:

*Objects are the instances of classes that have already been defined.

*A class can have any no. of instances.

*An object can be created in Java in 2 steps:

Step1: Declaring an object reference

Step2: Allocating memory to the object

Step1: Declaring an object reference

First we should declare a variable of a class type using the following syntax:

Syntax:

```
className objectName;
```

Eg: Shape s;

Step2: Allocating memory to the object

The allocation of memory for a class instance is done using the 'new' operator in Java.

The syntax is shown below:

Syntax:

```
objectName=new className( );
```

Eg: s=new Shape( );

*When we declare an object reference, it points to 'null' by default. 'new' operator actually creates an object and allocates memory for it and then the object reference will point to the newly created object.

The following figure shows the effect of both the statements:

Statement	Effect
Shape s;	s → null
s=new Shape();	s → :Shape length breadth

fig: Creating an object

* The above statements can be combined and written as a single statement as shown below:

Shape

Syntax:

```
ClassName ObjectName=new ClassName();
```

Eg: Shape s=new Shape();

Example:

```
//ObjectDemo.java
class Shape{
    double length;
    double breadth;
    }
class ObjectDemo{
    public static void main(String args[]){
            Shape s;    //Declaring object
            s=new Shape(); // Allocating memory to the object
            s.length=10;
            s.breadth=5;
            double a=s.length*s.breadth;
            System.out.println("Area= "+a);
        }
}
```

Output:

Area = 50

Note:

Once an object has been created, its instance variable can be assessed using the object name and dot (.) operator.

Eg: s.length

Methods:

* A method is just like a function in the procedural languages. A method contains a set of statements to be executed to achieve a particular objective.

The following is the syntax for creating methods:

Syntax:

```
returntype method_name(parameter_list){
        //statements_list
}
```

* Here method_name is the name of the method
  parameter_list specifies the argument to be passed to the method when it is invoked
  returntype is the data type of the value returned by the method
  statement_list indicates the set of statements to be executed while the method execution
* Usually methods have the 'return' statement as the last statement in the statement_list.
* When a method doesn't return any value, its returntype must be specified as 'void.'
* To invoke a method, we require an object reference. Methods are invoked using the following syntax:

Syntax:

Non static methods = `object_name.method-name(parameter_list);`

Note:

Class level methods (static methods) doesn't require th object reference. class name itself is enough to invo them.

Syntax:

Static methods = `class_name.classmethod_name(parameter_list);`

Example:

```
//MethodDemo.java
class Shape{
    double length;
    double breadth;
    double area(){
        return length*breadth;
    }
}
class MethodDemo{
    public static void main(String args[]){
        Shape s=new Shape();
        s.length=10;
        s.breadth=30;
        double area=s.area();
        System.out.println("Area = "+area);
    }
}
```

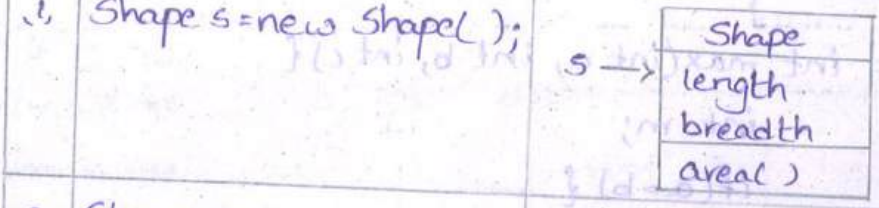
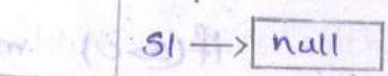
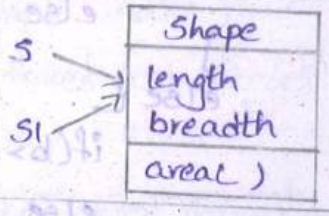
Output:

```
Area=300
```

Assigning Object Reference:

* When we assign an object to another object reference, the result will not be same as we expect.
* The assignment takes place in different manner, when object references are assigned.

Consider the following example:

S.No	Statement	Effect
1.	Shape s=new Shape( );	
2.	Shape s1;	
3.	s1=s;	

Method Overloading:

* Method Overloading is a way of implementing polymorphism mechanism in Java.
* When we define a class, we can have multiple methods with the same name as long as the number (or) type of parameters vary or as long as the return type of the methods vary.
* This property of having multiple methods with the same name inside a class is known as "method overloading."
* When a method is overloaded, it will have multiple versions
* The resolution of which version happens at compile time only. That means at compile time, the java compiler decides which version among all the overloaded versions to execute.
* Hence method overloading is an example for compile

time polymorphism (or) static polymorphism (or) early bin

* the following example illustrates method overloading

Java:

```
class OverloadingDemo{
    int max(int a, int b){
        if a>b  return a;
        else    return b;
    }
    int max(int a, int b, int c){
      int m;
      if(a>b){
          if(a>c)  m=a;
          else     m=c;
      }
      else{
          if(b>c)  m=b;
          else     m=c;
      }
      return m;
    }
    int max(int []a, int n){
      int m=a[0];
      for(int i=1; i<n; i++){
          if(a[i]>m)  m=a[i];
      }
      return m;
    }
    public static void main(String args[])
    {
      OverloadingDemo o=new OverloadingDemo
      int m=o.max(56,80);
      System.out.println("Maximum="+m);
```

```
        m=o.max(78, 26, 55);
        System.out.println("Maximum= "+m);
        int arr[]=new int[];
        arr={26, 75, 92, 16, 54, 87};
        m=o.max(arr, 6);
        System.out.println("Maximum = "+m);
    }
}
```

Output:

```
Maximum=80
Maximum=78
Maximum=92
```

* In the above program, we have a class with 3 methods all of which have the same name 'max()'.
* The first version of max() takes two integer arguments and return an integer.
* The second version of max() takes three integer arguments and return an integer.
* The third version of max() takes an integer array arguments
* Since we have three methods with the same name in a class, method overloading been implemented here.
* The output of the above program is given below:

```
Maximum=80
Maximum=78
Maximum=92
```

Constructors:

* A constructor is a special method defined inside a class. It can contain the object initialization code.
* You may require your class instances to be initialized as soon as they are created by for that

constructor will be useful to you.

* A constructor is a method which has the same name as the name of the class in which it resides.
* Constructors do not have return type even not void.
* A constructor implicitly has a return type. The class type itself. So you should not declare a constructor's return type as void.
* Constructor will be automatically executed as soon as an object is created by using 'new' operator.
* If you don't have a constructor defined inside your class, the JVM will create a default constructor and executes it at the time of object creation.
* A default constructor means a method with classname, with no parameters and no body.
* The following is an example of a constructor

```
class Rectangle{                                    //Default constructor
    double length;
    double breadth;                                 //instance variables
    //Constructor
    Rectangle(){
        length=20;
        breadth=10;
    }
    double area(){
        return length*breadth;
    }
}
class ConstructorDemo{
    public static void main(String args[]){
```

```
        Rectangle r1=new Rectangle();
        Rectangle r2=new Rectangle();
        System.out.println("Area of object r1: "+r1.area());
        System.out.println("Area of object r2: "+r2.area());
    }
}
```

Output:

```
Area of object r1:200
Area of object r2:200
```

Types of Constructor:

1. Default constructor/parameterless constructor
2. Parameterized constructor

Parameterized Constructor:

* You may wish that every object of a class to be initialized to a different state.
* To do so, you require a constructor that accepts parameters (or) arguments.
* A constructor that takes parameters is known as a "parameterized constructor."
* When you have a parameterized constructor defined inside a class, you have to pass those parameters to the same constructor, while you are creating an object.
* The following example shows the creation and usage of parameterized constructor:

```
class Rectangle{
    double length;
    double breadth;
    //parameterized constructor
    Rectangle(double l, double b){
        length=l;
        breadth=b;
    }
```

```
    double area(){
        return length*breadth;
    }
}
class ConstructorDemo{
    public static void main(String args[]){
        Rectangle r1=new Rectangle(15,8);
        Rectangle r2=new Rectangle(25,10);
        System.out.println("Area of object r1: "+r1.area());
        System.out.println("Area of object r2: "+r2.area());
    }
}
```

Output:

```
Area of object r1=120
Area of object r2=250
```

Constructor Overloading:

* Just like methods, constructors can be overloaded.
* You can have multiple constructors inside a class as long as the parameters vary i.e., having different types of parameters (or) different no. of parameters.
* Such a mechanism of having multiple constructor inside a class is known as "Constructor Overloading."
* Constructor overloading is another way to implement polymorphism in Java.
* The overloaded constructors are differentiated on the basis of arguments passed to them.

* The following example illustrates constructor overloading:

```
class Rectangle{
    double length;
    double breadth;
    //constructor1
    Rectangle(){
        length=30;
        breadth=20;
    }
    //constructor2
    Rectangle(double l, double b){
        length=l;
        breadth=b;
    }
    double area(){
        return length*breadth;
    }
}
class ConstructorOverloadingDemo{
    public static void main(String args[]){
        Rectangle r1=new Rectangle(); //creating object using default constructor
        Rectangle r2=new Rectangle(15,12) //creating object using parameterized
        System.out.println("Area of object r1: "+r1.area());
        System.out.println("Area of object r2: "+r2.area());
    }
}
```

Output:

Area of object r1=600

Area of object r2=180

Constructor Versus Methods:

Constructors	Methods
1, Constructors don't have return type even not void.	1, Methods have return type or can be void.
2, The name of the constructor should exactly same as the class name.	2, A method can have any name otherthan the class name.
3, A constructor is executed only once's as soon as an object is created and not there after.	3, Method is to be invoked after an object is created & can be executedmany no. of times.
4, Constructors can have parameters..	4, Methods can also have parameters.
5, Constructors can be overloaded.	5, Methods can also overloaded.
6, Constructors can be declared as public, private, protected or default.	6, Methods can also be declared as public, private, protected or default.
7, Constructors cannot be declared abstract, static, final or synchronized.	7, Methods can be declared abstract, static, final or synchronized
8, Constructors cannot be inherited.	8, Methods can be inherited.
9, The roal of constructor is to initialize objects.	9, The roal of method is to perform operations.

Garbage Collection:

*In many programming languages it is the duity of the programmer to allocate and deallocate memory in the heap. (dynamic memory).

*In Java programmer can be able to allocate memory dynamically by creating objects without worrying abart the deallocation of memory, when the objects become no longer useful.

* The Java Runtime environment (JVM) deletes objects when it determines those objects are no longer required.

* the Java runtime environment has it's aon algorithms for deciding when the memory allocated to an object must be released. This automated process is known as "Garbage Collection."

* The Java runtime environment has a garbage collector that periodically freeze the memory used by the objects that are no longer needed.

* there are 2 basic approaches used by garbage collectors:

1. Reference counting
2. Tracing

1. Reference Counting:

*It maintainances a reference caunt for every object. A newly created object will have a reference caunt one(1) Throughout it's lifetime, the reference caunt of an object may get increased or decreased.

*When the other objects refer the object under consideration if reference caunt increases. As the

referencing objects move to the other object, reference count decreases.

*When the reference count of an object becomes zero(0), it can be garbage collected.

2. Tracing: (Mark & Sweep)

* This mechanism scans all the objects in the memory and marks those objects that are referencing.
* The objects that are not marked are treated as garbage and they can be ~~sweeped~~ swept out of the memory.
* This mechanism is also known as "mark and sweep garbage collection."
* Mark and sweep garbage collectors further use the techniques of compaction and copying.
* Compaction moves all live objects towards one end making the other end a large free space.
* Copying techniques copy all the live objects besides each other into a new space and the whole space is considered free now.
* Garbage collector executed either synchronously or asynchronously. It executes synchronously when the system run's out of the memory and asynchronously when the system is idle.
* The garbage collector can be invoked at runtime by calling System.gc() or Runtime.gc() but this invokation doesn't guarantee the garbage collection is done.

Finalization:

* Before an object gets garbage collected the garbage collector gives the object an opportunity to clean up itself through a called to the 'finalize()' method of that object. This process is known as finalization.
* The finalize() method is a free defined method in 'java.lang.Object' class. A class must override a finalize() method to perform any cleanup activity if required by the object.

Advantages of garbage collection:

* Garbage collection lets the programmer to create as many objects as he wants and allocate memory to them without worrying about deallocation of memory.
* It also helps ensuring the integrity of the programmes.

Disadvantages of garbage collection:

* The disadvantage of garbage collection is a overhead to keep track of objects that are not being referred.
* The finalization and cleaning up the unreferenced objects take more CPU time.

The static keyword:

* 'Static' is an important keyword in Java. It can be used for the following purposes:

1. To declare static variables.
2. To define static methods.
3. To define static blocks.

* Static members are the class level members.

1. Static variables:

* Static variables are also known as class variables.
* Static variables are declared inside a class but outside a method just like instance variables.
* The only difference in the declaration of static variables and instance variables is the 'static' keyword.
* The syntax for creating a static variable is given below:

Syntax:

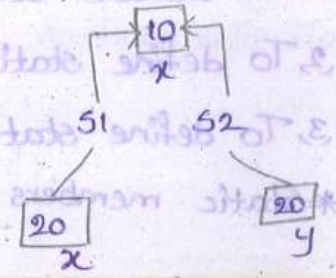
```
static datatype variable_name = value;
```

* Static variables have class scope. When you declare a variable as static, only one copy of it is created in memory. That copy will be shared among all the instances of the class to which it belongs.
* Simply static variables are shared variables for all the objects of a particular class.
* The following example shows the concepts of static variables:

```
class Sample {
    static int x=10; //class variable/static variable
    int y=20; //instance variable
}
```

Suppose s1, s2 are two objects of sample

```
Sample s1=new Sample();
Sample s2=new Sample();
```



* Since x is a static variable, only one copy of it is available in the memory and that will be shared between s1 and s2.

* Since y is a non-static variable / instance variable, separate copys of y are available for s1 and s2.

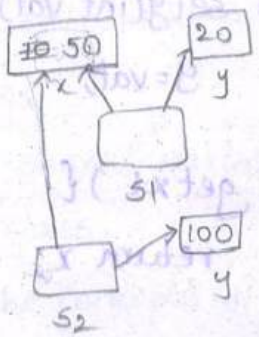
Eg:

```
// Test.java
class Sample{
    static int x;
        int y;
    void setx(int val){
        x=val;
    }
    void sety(int val){
        y=val;
    }
    int getx(){
        return x;
    }
    int gety(){
        return y;
    }
}
class Test{
    public static void main(String args[]){
        Sample s1=new Sample();
        Sample s2=new Sample();
        s1.setx(10);
        s1.sety(20);
        System.out.println("Value of x set by s1: "
                +s1.getx());
```

```
        System.out.println("Value of y set by s1: "
                                    +s1.gety());
        s2.setx(50);
        s2.sety(100);
        System.out.println("Value of x set by s2: "
                                    +s2.getx());
        System.out.println("Value of y set by s2: "
                                    +s2.gety());
        System.out.println("Value of x set by s1: "
                                    +s1.getx());
        System.out.println("Value of x accessed through
                        class name: " +sample.x);
    }
}
```

Output:

```
Value of x set by s1:10
Value of y set by s1:20
Value of x set by s2:50
Value of y set by s2:100
Value of x set by s1:50
Value of x accessed through class name:50
```



Note:

*Static variables can be accessed through the class name:

```
class_name.variable_name;
```

Eg: Sample.x;

2, Static methods : //class methods

*Just like static variables we can also declare the methods of class as static.

```
static returntype methodname(parameterlist) {
        ≡
}
```

* Static methods are also called as class methods.
* Similar to static variables, static methods can be called using the class name as shown below:

```
class_name.method_name( );
```

* Calling a static method doesnot required an object reference.
* Static methods have several restrictions as listed below:

1. They can call other static methods only.
2. They must access the static data/variables only.
3. Static methods should not refer 'this' or 'super' keyword in any way.

* In the above example, the method setx( ) can be declared as static but the sety( ) method shouldnot be declaredasstatic. The reason here is setx( ) method is accessing a static variable x but sety( ) method is accessing a non-static variable y.

3. Static block:

* A static block is a block inside a class that contains the initialization code that initializes the static members of the class.
* A static block is written using following syntax:

```
static{
    //Initialization code
}
```

* The initialization of the static variable can be done directly when the initialization is simple.

*If you have complex initialization logic for static members then you can use a static block.

*A static block is automatically executed only once's. When the class is first loaded into the JVM, the static block get executed even before the main( ) method.

Eg:

```
class StaticBlockDemo {
    static int x;
    static int y;
    //Static Block
    static {
        System.out.println("In Static Block.....");
        y=3;
        x=y*15;
    }
    StaticBlockDemo( ) {  // Constructor
        System.out.println("In Constructor.....");
    }
    public static void main(String args[ ]) {
        System.out.println("In main( ) method.....");
        StaticBlockDemo s=new StaticBlockDemo( );
        System.out.println("Value of x using s = " +s.x);
        System.out.println("Value of y using s = " +s.y);
        System.out.println("Value of x using class name="
                +StaticBlockDemo.x);
        System.out.println("Value of y using class name="
                +StaticBlockDemo.y);
    }
}
```

Output:

In Static Block----
In main() method----
In Constructor---
Value of x using s=45
Value of y using s=3
Value of x using class name=45
Value of y using class name=3

'this' keyword:

* 'this' keyword is used in an instance method to reference to the object that contains the method. Simply 'this' refers to the current object
* Whenever and were every a reference to the object of current class type is required, 'this' can be used.
*It can also be used to differentiate between an instance variable and a local variable when both of the same name.

Consider the following code:

```
class Rectangle{
        double length;
        double breadth;
        Rectangle(double length,double breadth)
        {
           length=length;
           breadth=breadth;
        }
        double areal){
                return length*breadth;
        }
        public static void main(String args[])
        {
```

```
        Rectangle r=new Rectangle(20,10);
        System.out.println("Area of r:" +r.area());
    }
}
```

In the above example, in the constructors definition (line 4-8) we used same names for local variables & instance variables (length & breadth). While executing the program JVM could not be able differentiate between them. It treats both as instance variables. So both length & breadth hold 0.0 (as they are class members). So the area will be 0.0 only.

* To distinguish between the local & instance variables we can make use of 'this' keyword as follows:

```
class Rectangle{
    double length;
    double breadth;
    Rectangle(double length, double breadth)
    {
    this.length = length;
    this.breadth=breadth;
    }
    double area(){
        return length*breadth;
    }
    public static void main(String args[]){
        Rectangle r=new Rectangle(20,10);
        System.out.println("Area of r:" +r.area());
    }
}
```

Output:

Area of r:200

* There is another use of 'this' keyword. We can call one constructor within from the other constructor of the same class using 'this' keyword.
* This process of calling one constructor from another constructor is known as "constructor chaining."

Consider the following example program:

```
class Rectangle{
    double length;
    double breadth;
    //First constructor
    Rectangle( ) {
        //Constructor chain
        this(14,10);
    }
    //Second Constructor
    Rectangle(double length, double breadth) {
        this.length=length;
        this.breadth=breadth;
    }
    double area( ){
        return length*breadth;
    }
    public static void main(String args[ ]){
        Rectangle r1=new Rectangle(20,10);
        System.out.println("Area of r1: " +r1.area( ));
        Rectangle r2=new Rectangle( );
        System.out.println("Area of r2: " +r2.area( ));
    }
}
```

In the above class we define 2 constructors i.e., a parameter less constructor and a parameterized constructor. Inside the parameter less constructor we have used 'this' keyword for constructor chaining.

* When you create object of Rectangle class using parameter less constructor, constructor chaining will happens by pass 14 & 10 as parameters to the parameterized constructor.

Output:

Area of r1: 200

Area of r2: 140

Note:

The keyword 'this' must not be refered inside a static method.

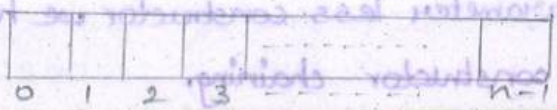
Arrays:

* An array is a memory space allocated that can store multiple values of same datatype in continuous location.
* Simply an array is a collection of homogenous elements.
* The individual values of an array are known as elements
* Arrays can be of 2 types:

1. One dimensional (or) single dimensional array
2. Multi-dimensional array.

1. One dimensional array:

* In a one dimensional array, a single subscript([ ]) or index is used. Each index refers to an individual element of the array.
* Array indices start from zero(0) & go upto n-1, if the size of the array is n.
* The index zero(0) represent first element, the index 1 represent second element & the index n-1 represent the last element in the array.



* Creating an array is similar to creating an object.
It involves 3 steps.

Step1: Declaring an array

Step2: Allocating memory

Step3: Assigning values

Declaring an array:

Declaration of an array can be done by using the following syntax:

```
datatype arrayname[ ];
        (or)
datatype[ ] arrayname;
```

Eg: int arr[ ];
(or)
int[ ] arr;

Here datatype refers to the type of values that the array contains.

Allocating memory:

Allocation of memory to array elements is done using following syntax:

```
arrayname = new datatype[size];
```

Eg: arr = new int[5];

Here
size refers to the no. of elements in that array.

*Alternatively we can combine the above 2 steps as follows:

```
datatype[ ] arrayname = new datatype[size];
        (or)
datatype arrayname[ ] = new datatype[size];
```

Eg: int arr[ ] = new int[5];
(or)
int[ ] arr = new int[5];

Assigning values:

Assignment of values to an array is also known as initialization of array. It can be done either by individual assignment (or) direct assignment values can also be assigned to reading from keyboard.

Individual assignment:

Syntax: `arrayname[index] = value;`

```
int[] arr = new int[5];
```

Eg: `arr[0]=22;`

`arr[1]=35;`

`arr[2]=50;`

Direct assignment:

Creating & assigning values is done at the sametime using the following syntax:

`type arrayname[ ]={list of value};`

Eg: `int arr[ ]={22, 35, 50, 65, 72};`

Reading values from keyboard:

It requires a loop.

```
Scanner sc=new Scanner(System.in);
int arr[ ]=new int[5];
for(int i=0; i<5; i++){
        //Reading values from keyboard & assigning
        arr[i]=sc.nextInt( );
}
```

2, Two dimensional array:

*A two dimensional array is defined as an array of array. A 2-D array can be treated as a table that is formed from rows & columns.

*A 2-D array can be creating using the following syntax:

```
datatype arrayname[ ][ ] = new int[size1][size2]
```

Here size1 represents the no. of rows &
size2 represents the no. of columns

Eg: int arr[ ][ ] = new int[3][2];

	0	1
0	arr[0][0]	arr[0][1]
1	arr[1][0]	arr[1][1]
2	arr[2][0]	arr[2][1]

Assignment of elements can be done in following ways:

Individual assignment:

Syntax: arrayname[index1][index2] = value;

Eg: arr[0][0] = 15;
arr[0][1] = 13;
arr[1][0] = 12;

Direct assignment:

Creating and assigning values is done at the same time using the following syntax:

```
type arrayname[ ][ ] = {list of arrays};
```

Eg: int arr[ ][ ] = {{15, 18}, {12, 20}, {17, 19}};

Reading value from keyboard:

It requires 2 nested loops, for rows & columns

```
Scanner sc = new Scanner(System.in);
int arr[ ][ ] = new int[3][2];
for(int i=0; i<3; i++){
    for(int j=0; j<2; j++){
        arr[i][j] = sc.nextInt();
}}
```

```
/* Program to implement matrix multiplication */
import java.util.Scanner;
class Matrix {
    public static void main(String args[]){
        int[][] A = new int[10][10];
        int[][] B = new int[10][10];
        int[][] C = new int[10][10];
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter the no. of rows of
                                first matrix: ");
        int r1=sc.nextInt();
        System.out.println("Enter the no. of columns of
                                first matrix: ");
        int c1=sc.nextInt();
        System.out.println("Enter the no. of rows of
                                second matrix: ");
        int r2=sc.nextInt();
        System.out.println("Enter the no. of columns
                                of second matrix: ");
        int c2=sc.nextInt();
        if(c1==r2){
            System.out.println("Enter the elements of
                                first matrix :");
            for(int i=0; i<r1; i++){
                for(int j=0; j<c1; j++){
                    A[i][j]=sc.nextInt();
            }}
            System.out.println("Enter the elements of
                                second matrix: ");
            for(int i=0; i<r2; i++){
```

```
        for(int j=0; j<c2; j++){
            B[i][j] = sc.nextInt()
        }
    }
    for(int i=0; i<r1; i++){
        for(int j=0; j<c2; j++){
            c[i][j]=0;
            for(int k=0; k<c1; k++){
                c[i][j]+=A[i][k]*B[k][j];
            }
        }
    }
    System.out.println("First matrix is: ");
    for(int i=0; i<r1; i++){
        for(int j=0; j<c1; j++){
            System.out.print(A[i][j] +"\t");
        }
        System.out.print("\n");
    }
    System.out.println("Second matrix is: ");
    for(int i=0; i<r2; i++){
        for(int j=0; j<c2; j++){
            System.out.print(B[i][j] +"\t");
        }
        System.out.print("\n");
    }
    System.out.println("Resultant matrix after multiplication");
    for(int i=0; i<r1; i++){
        for(int j=0; j<c2;j++){
            System.out.print(c[i][j] +"\t");
        }
        System.out.print("\n");
    }
    else{
        System.out.println("Matrix multiplication is not possible");
}}}
```

Passing array as arguments to methods:

It is possible to pass arrays as parameters to methods in Java.

The following syntax is as follows:

```
returntype methodname(type[] array-name){
        ≡
}
```

The following example defines a method which accepts a single dimensional integer array as arguments and returns the some of the elements in that array.

```
/*Program to illustrate passing arrays to methods as arguments
import java.util.Scanner;
class Sum{
    //Defines a method which takes array as arguments
    int findSum(int[] arr){
        int s=0;
//for loop  for(int i=0; i<arr.length; i++){
            s=s+arr[i];
        }
        return s;
    }
public static void main(String args[]){
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter the size of the array:");
        int n=sc.nextInt();
        int[] elements=new int[n];
        System.out.println("Enter the elements:");
        for(int i=0; i<n; i++){
            elements[i]=sc.nextInt();
        }
```

//for-each loop
for(int n:a){
s+=n;
}

```
        Sum obj=new Sum();
        int s=obj.findSum(elements);
        System.out.println("Sum of the elements of given
                                array: "+s);
    }
}
```

Output:

```
Enter the size of the array:
5
Enter the elements:
1
2
3
4
5
Sum of the elements of given array: 15
```

Returning multiple values from methods:

It is possible to return multiple values from a method. It can be done in 2 ways:

1. When a method return an array
2. When a method return an object

1. Method returning arrays:

When a method return an array as return value.

It's syntax is as follows:

```
type[] methodName(parameter list){
    ==
}
```

The following program illustrates a method returning a 2-D array as return value.

```
/* Program to illustrate method returning arrays */
import java.util.Scanner;
class MatrixAddition{
    // Method returning a 2 dimensional array
    static int[][] add(int[][] a, int[][] b, r, c){
        int[][] m=new int[r][c];
        for(int i=0; i<r; i++){
            for(int j=0; j<c; j++){
                m[i][j]=a[i][j]+b[i][j];
            }
        }
        return m; //Returning array
    }
    public static void main(String args[]){
        MatrixAddition m=new MatrixAddition();
        int[][] A=new int[10][10];
        int[][] B=new int[10][10];
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter the no. of rows of matrices: ");
        int r=sc.nextInt();
        System.out.println("Enter the no. columns of matrices: ");
        int c=sc.nextInt();
        System.out.println("Enter the elements of first matrix: ");
        for(int i=0; i<r; i++){
            for(int j=0; j<c; j++){
                A[i][j]=sc.nextInt();
            }
        }
        System.out.println("Enter the elements of second matrix: ");
        for(int i=0; i<r; i++){
            for(int j=0; j<c; j++){
```

```
                B[i][j]=sc.nextInt();
            }
        }
        int[][] C=m.add(A,B,r,c);
        System.out.println("First matrix is: ");
        for(int i=0; i<r; i++){
            for(int j=0; j<c; j++){
                System.out.print(A[i][j] +"\t");
            }
            System.out.print("\n");
        }
        System.out.println("Second matrix is: ");
        for(int i=0; i<r; i++){
            for(int j=0; j<c; j++){
                System.out.print(B[i][j] +"\t");
            }
            System.out.print("\n");
        }
        System.out.println("Resultant matrix is: ");
        for(int i=0; i<r; i++){
            for(int j=0; j<c; j++){
                System.out.print(C[i][j] +"\t");
            }
            System.out.print("\n");
        }
    }
}
```

2. Methods returning objects:

* Another possible way to return multiple values from a method is to return an object. An object encapsulates multiple data items.
* The following example shows how a method return

an object as return value and also a method accepting an object as a parameter.

```
class Employee{
    int empId;
    String empName;
    double sal;
    //Method that returns object
    Static Employee createEmp(){
        System.out.println("Employee object created!");
        return new Employee(); //Returning object
    }
    void setEmpDetails(int eid, String n, double s){
        empId=eid;
        empName=n;
        sal=s;
    }
    //Method accepting object as parameter
    static void printEmpDetails(Employee e){
        System.out.println("Employee ID: "+e.empId);
        System.out.println("Name: " +e.empName);
        System.out.println("Salary: " +e.sal);
    }
    public static void main(String[] arg){
        Employee obj;
        obj=createEmp(); obj=Employee.CreateEmp();
        obj.setEmpDetails(125, "Naresh", 26000);
        printEmpDetails(obj);
    }
}
```

Using for-each loop with arrays:

An enhanced version of 'for' loop called 'for-each' was introduced Java to iterate over the elements of an array. The loop continuous to execute as long as there are elements in the given array. The loop terminates when all the elements of the array are iterated.

The syntax is as follows:

```
for(type var : arrayName){
        //body of the loop
}
```

Eg:

```
class Sample{
    public static void main(String args[ ]){
        int[ ] arr={26,55,72,19,21};
        int sum=0;
        for(int i : arr){ //for-each loop
            sum=sum+i;
        }
        System.out.println("Sum of array elements:"+sum);
    }
}
```

o/p: Sum of array elements: 193

Command Line arguments:

* While executing a Java application from the command line, you can pass a no. of arguments to the application from the command line itself.
* The arguments passed to the application from the command line at the time of execution are known as "command line arguments." command line arguments

```
$ java programName arg1, arg2, ..., argN
```

* The arguments we supply from the command line to the application are stored in a 'String' array, which is the argument for the main() method of the application.
* The format of main() method of a Java program is given below:

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

* Here 'args' is an array of string objects that holds the command line arguments.
* args[0] is the first command line argument passed to the main() method.
* The program name itself is the first command line arguments and it is placed at index 0 of args.
* The following is a program that accepts command line arguments and prints those arguments:

```
class CommandLineArgsDemo{
    public static void main(String[] args){
        System.out.println("No. of command line
                arguments is: "+args.length);
        for(int i=0; i<args.length; i++){
            System.out.println("Argument "+(i+1)+": "
                +args[i]);
        }
    }
}
```

Output:

```
$javac CommandLineArgsDemo.java
$java CommandLineArgsDemo Hai Hello
No.of commandline arguments is:3
Argument 1:CommandLineArgsDemo
Argument 2:Hai
Argument 3:Hello
```

Eg-2:

```
/*Program to read n numbers as command line arguments
and find there & their sum */
class Sum{
    public static void main(String args[]){
        System.out.println("No.of numbers given: "
                                +(args.length-1));
        double s=0;
        for(int i=1; i<args.length; i++){
            s=s+Double.parseDouble(args[i]);
        }
        System.out.println("Sum of the given numbers:"
                                +s);
    }
}
```

Output:

```
$javac Sum.java
$java Sum 10 20 30 40.9
No.of numbers given:4
Sum of the given numbers: 100.9
```

Nested Classes:

Nested classes are the classes defined inside the other classes. There are 4 types of nested classes as listed below:

1. Non-static nested class (or) Inner class
2. Static nested class
3. Local inner class
4. Anonymous inner class

1. Inner class:

* An inner class is a class defined inside another class just like any instance variable (or) methods.
* These non-static nested classes can have access to all the members of the outerclass in which it is defined.
* Inner classes cannot have static members or methods. We can declare static constants inside an inner class.
* The following is an example to illustrate non-static nested classes:

```
//program to demonstrate non-static nested class
class Outer{
    int x;
    static int y;
    outer(int a, int b){
        x=a;
        y=b;
    }
    //Defining a inner class
    class Inner{
```

```
    int z;
    static finally int w=100;
    Inner(int a){
        z=a;
    }
    void show(){
        System.out.println("From inner class.....");
        System.out.println("x = " +x);
        System.out.println("y = " +y);
        System.out.println("z = " +z);
        System.out.println("w = " +w);
    }
}
void createInnerObject(int n){
    // Creating inner class object
    new Inner(n).show();
}
}
class InnerClassDemo{
    public static void main(String args[]){
        Outer o=new Outer(10,20);
        o.createInnerObject(30);
    }
}
```

Output:

From inner class.....

x=10

y=20

z=30

w=100

2, Static nested class :

* A static nested class is a nested class that is declared 'static.'
* Unlike inner classes, static nested class can have static variables and static methods along with static constants inside it.
* Like a static block, static nested class cannot have access to the non-static member of the outer class in which it is defined.
* The following is the example program which illustrates a static nested class:

```
//Program to demonstrate static nested class
class Outer{
    int x;
    static int y;
    Outer(int a, int b){
        x=a;
        y=b;
    }
    //Defining a nested class
    static class NestedClass{
        Static final int w=100;
        NestedClass(int a){
            z=a;
        }
        Void show(){
            System.out.println("x is not available
                here, since it is non-static");
```

```
            System.out.println("y = " +y);
            System.out.println("z = " +z);
            System.out.println("w = " +w);
        }
    }
    void createNestedClassObject(int n){
        //Create Nested class object
        new NestedClass(n).show();
    }
}
class StaticNestedClassDemo{
    public static void main(String args[]){
        Outer o=new Outer(10,20);
        o.createNestedClassObject(30);
        System.out.println("Value of x using outer class
                            object = " +o.x);
    }
}
```

Output:

```
x is not available here, since it is non-static
y=20
z=30
w=100
Value of x using outer class object =10
```

3. Local inner class:

* A local inner class is a class defined inside a method of another class.
* Since a local inner class is defined a method, it has a local scope of that method.
* A local inner class can access only the final local member (local constant) of the method & final parameters

of that method.

* The following is an example program:

```
/* Program to demonstrate local inner class */
class Outer{
    int x;
    static int y;
    Outer(int a, int b){
        x=a;
        y=b;
    }
    //Instance method definition
    void outerInstanceMethod(int a){
        final int z=a;
        int w=a+2;
        //Local inner class definition
        class LocalInner{
            void show(){
                System.out.println("From local inner
                                    class:");
                System.out.println("x = "+x);
                System.out.println("y= "+y);
                System.out.println("z = "+z);
                System.out.println("We cannot be
                    accessed here since it is not
                                    final");
            }
        }
        new LocalInner().show();
        System.out.println("From outer instance
                            method w= "+w);
    }
}
```

```
class LocalInnerDemo{
    public static void main(String args[]){
        Outer o=new Outer(10,20);
        o.OuterInstanceMethod(30);
    }
}
```

Output :

4, Anonymous inner class :

*An anonymous inner class is a local inner class that has no name.

*For an anonymous inner class we can have only one instance(object).

*Anonymous inner class are very effective in implementing event handling in Java.

Need of nested class(or) Use of nested class:

*Nested class bring more object orientation into your programming. The reason is that they let you encapsulate logic into classes.

*Inner classes provide a structure hierarchy.

*Nested classes allow call backs to be defined conviently.

*Nested classes are very effective in implementing event handling in Java.

* If you feel that a class will be useful to only one class then it is better to next a class into another.

* Nested class enchance readability & maintainability of the code.

UNIT-III

Inheritance:

* Inheritance is one of corner stones of object oriented programming.
* Inheritance is the process of creating new class from the existing class by acquiring the structure and behaviour of those existing classes.
* Through the process of inheritance we can obtain the new class. The new classes created through inheritance are known as sub class (or) child class (or) derived class.
* They existing classes from which the new one's are derived are known as base class (or) super class (or) parent class.
* A child class can access all the members (except the private members) of it's parent and a child class can have it's own/unique structure or behaviour which cannot be accessed by it's base class.
* The following figure shows the inheritance relationship:

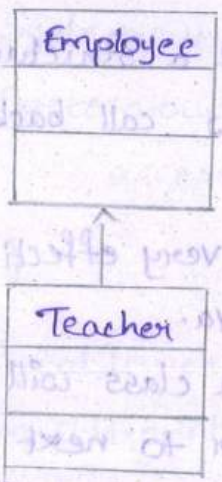


fig: Inheritance

* Inheritance indicates a generalization-specialization relationship between two classes. (2)

* Inheritance promotes the reusability of code.

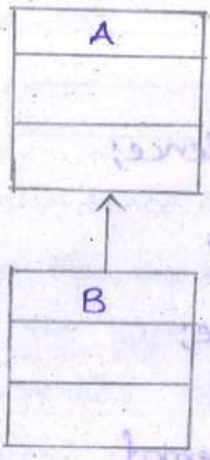
Types of inheritance:

There are 5 types of inheritance as listed below:

1. Single inheritance
2. Multilevel inheritance
3. Multiple inheritance
4. Hierarchical inheritance
5. Hybrid inheritance

1. Single Inheritance:

* When only one class is derived from an existing class, the inheritance is termed as single inheritance.
* In single inheritance, classes have only one base class.
* The following figure shows single inheritance.



The following is an example program that illustrate single inheritance:

```
//program to illustrate single inheritance
class Employee{
    int empId;
    String empName;
    void printDetails(){
        System.out.println("Employee ID: "+empId);
        System.out.println(" Name: "+empName);
    }
}
class Teacher extends Employee{
    int experience;
    String course;
    Teacher(int id, String name, int exp, String course){
        empId=id;
        empName=name;
        experience=exp;
        course=c;
    }
    int getExp(){
        return experience;
    }
    String getCourse(){
        return course;
    }
}
class SingleInheritanceDemo{
    public static void main(String args[]){
        Teacher t=new Teacher(123,"Mohan", 10,
                              "computerscience");
        t.printDetails();
        System.out.println("Teaching Experience: "
                           +t.getExp()+"years");
```

```
        System.out.println("Course:"+t.getCourse());
    }
}
```

Output:

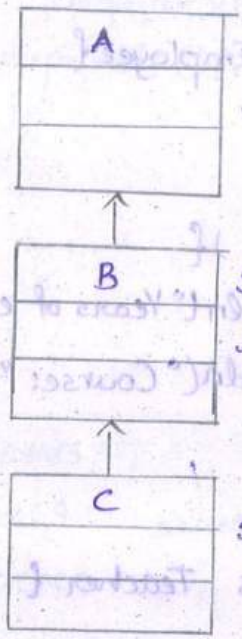
Employee ID: 123

Name: Mohan

Teaching Experience: 10

Course: computerscience

2. Multilevel Inheritance:



* In multilevel inheritance A subclass can be a parent of another class. From the above fig. B is a subclass of A and inturn it is the superclass of C. The class C now will have the properties of both A & B along with its unique features.

* This inheritance can go down to any no. of levels. As the levels of inheritance increase, the code becomes more complex.

The following program demonstrates multilevel inheritance:

```
/* Program to illustrate multilevel inheritance */
class Employee{
    int empId;
    String empName;
    void printDetails( ){
        System.out.println("Employee ID : " +empId);
        System.out.println("Employee Name : " +empName);
    }
}
// First level of inheritance
class Teacher extends Employee{
    int experience;
    String course;
    void printDetails( ){
        System.out.println("Years of experience : " +experience);
        System.out.println("Course: " +course);
    }
}
// Second level of inheritance
class Professor extends Teacher {
    int phdyear;
    String phduniv;
    Professor(int id, String name, int exp, String cour,
                                    int year, String univ){
        empId=id;
        empName=name;
        experience=exp;
        course=cour;
        phdyear=year;
        phduniv=univ;
    }
    void display( ) {
```

```
        printDetails();
        printDetails1();
        System.out.println("Ph.D awarded in : "+phdyear);
        System.out.println("University : "+phduniv);
    }
}
class MultilevelInheritanceDemo{
    public static void main(){
        Professor p=new Processor(471,"NSVVS",9,
                "ComputerScience",2015,"Andhra University");
        System.out.println("Professor Details");
        p.display();
    }
}
```

Output:

```
Professor Details
Employee ID:471
Employee Name: NSVVS
Years of experience:9
Course:ComputerScience
ph.D awarded in:2015
University:Andhra University
```

3, Multiple Inheritance:

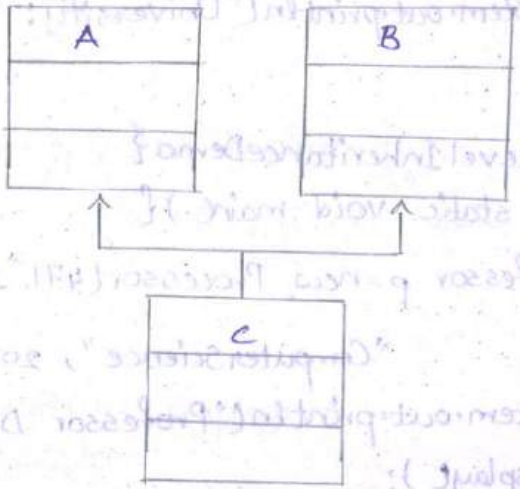
*When a sub class is derived from more than super classes, the inheritance is known as "multiple inheritance".

*In multiple inheritance a child class will have multiple parent classes. So, the child acquires the structure and behaviour of all those parent classes. There are some problems associated with multiple inheritance.

For instance, class 'c' is inherited from 2 classes. ie, A & B. suppose that A & B have same property named

'x'. When the object of class C tries to access the property 'x' there will a collision with the name.

The following figure shows multiple inheritance.

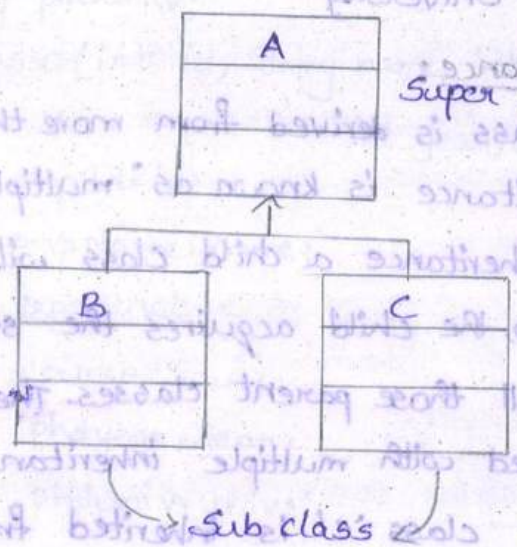


* Java doesn't support this multiple inheritance. But the same can be achieved through interfaces.

4. Hierarchical Inheritance:

* When a single base class has multiple subclasses, the type of inheritance is said to be hierarchical inheritance.
* In hierarchical inheritance, multiple child classes can be derived from single parent.

The following figure shows hierarchical inheritance.



```
//Program to illustrate hierarchical inheritance
class Employee {
    int empId;
    String empName;
    void printDetails( ){
        System.out.println("Employee ID:" +empId);
        System.out.println("Employee Name: " +empName);
    }
}
class Teacher extends Employee {
    int experience;
    String course;
    Teacher(int id, String name, int exp, String c){
        empId=id;
        empName =name;
        experience =exp;
        course =c;
    }
    void teacherDetails( ){
        printDetails( );
        System.out.println("Experience:" +experience);
        System.out.println("Course: " +course);
    }
}
class Technician extends Employee {
    String lab;
    Technician(int id, String n, String l){
        empId=id;
        empName=n;
        lab=l;
    }
    void printDetails2( ){
```

```
        printDetails();
        System.out.println("Lab: "+lab);
    }
}
class HierarchicalInheritanceDemo{
    public static void main(String args[]){
        Teacher t=new Teacher(525, "Bhargavi", 8,
                              "Computer Science");
        System.out.println("----* Teacher Details *----");
        t.printDetails();
        Technician t = new Technician(205, "Raju",
                                      "Mining Lab");
        System.out.println("----* Technician Details *----");
        t.printDetails2();
    }
}
```

Output:

```
----* Teacher Details *----
Employee ID:525
Employee Name: Bhargavi
Experience :8
Course: Computer Science
----* Technician Details *----
Employee ID: 205
Employee Name : Raju
Lab: Mining Lab
```

5, Hybrid inheritance:

*It is the combination of any of the other 4 types of inheritance.

The following figure shows hybrid inheritance amongst

classes.

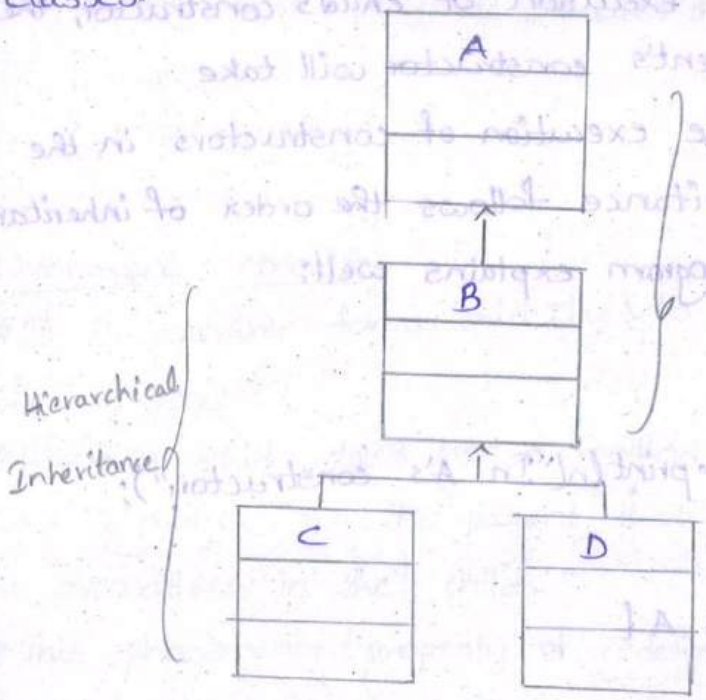


fig: Hybrid Inheritance

Creating new classes through inheritance (extends keyword):

* Java provides a way to create a new classes from the existing classes with the help of a keyword called 'extends'.
* The extends keyword is used to establish a parent-child relationship between classes.

The following is the syntax for creating a child class from an existing class:

```
class childclassName extends ParentclassName {
        //class-body
}
```

Calling sequence of constructors in the context of inheritance:

* When there is inheritance relationship (parent-child) between two classes, the execution of constructors will follow the top down order.
* When we create an instance of the child class, the constructor of child class will be automatically executed.

But before the execution of child's constructor, the execution of parent's constructor will take plan.

* That means the execution of constructors in the context of inheritance follows the order of inheritance.

The following program explains well:

```
class A{
    A(){
      System.out.println("In A's constructor");
    }
}
class B extends A{
    B(){
      System.out.println("In B's constructor");
    }
}
class C extends B{
    C(){
      System.out.println("In C's constructor");
    }
}
class Demo{
    public static void main(String args[]){
      C c=obj.new C();
    }
}
```

When we create an object of C, C's constructor will be invoked. Since 'C' is the subclass of B, C's constructor will invoke the constructor of B. B's constructor inturn invokes the constructor of A. So A's constructor will be executed first and then B's constructor later the constructor of C.

Hence the above program produces the following output:

In A's constructor!
In B's constructor!
In C's constructor!

Overriding methods:—

*It is possible for a chid class to re-define the methods of its parent.

*When a chid class has a method with the same name and signature as its parent, that method is said to be overridden in the child.

*This phenomenon/property of redefining base class methods in the subclasses is termed as "method overriding."

*Method overriding is a feature in Java that implements polymorphism like method overloading.

*Method overloading is a way to implement static polymorphism where as overriding is a way to implement dynamic/runtime polymorphism in Java.

*When a method is overridden inside a sub class and you called that method with the reference of sub class object, the sub class version of that method will be executed.

The following example illustrates method overriding:

```
/*Program to demonstrate method overriding*/
class Shape{
    double d1;
    double d2;
    double area(){
        return d1*d2;
    }
    double perimeter{
} }     return, d1+d2;
```

```
class Square extends Shape{
    Square(double d){
        d1=d2=d;
    }
    double perimeter(){
        return 4*d1;
    }
}
class Rectangle extends Shape{
    Rectangle(double l, double b){
        d1=l;
        d2=b;
    }
    double perimeter(){
        return 2*(d1+d2);
    }
}
class Triangle extends Shape{
    double d3;
    Triangle(double s1, double s2, double s3){
        d1=s1;
        d2=s2;
        d3=s3;
    }
    double area(){
        double s=(d1+d2+d3)/2;
        double a=math.sqrt(s*(s-d1)*(s-d2)*(s-d3));
        return a;
    }
    double perimeter(){
        return d1+d2+d3;
    }
}
```

```
class Circle extends Shape{
    Circle(double r){
        d1=d2=r;
    }
    double area(){
        return 3.14*d1*d2;
    }
    double circumference(){
        return 2*3.14*d1;
    }
}
class OverloadingD
class OverridingDemo{
    public static void main(String args[]){
        Shape s=new Shape();
        s.d1=10;
        s.d2=13;
        System.out.println("Area of shape s:"+s.area());
        System.out.println("Perimeter of shape s:"
                +s.perimeter());
        Square sq=new Square(10);
        System.out.println("Square object is created!");
        System.out.println("Area of square:"+sq.area());
        System.out.println("Perimeter of square:"
                +sq.perimeter());
        Rectangle r=new Rectangle(25,40);
        System.out.println("Area of rectangle:"+r.area());
        System.out.println("Perimeter of rectangle:"
                +r.perimeter());
        Triangle t=new Triangle(10,10,10);
        System.out.println("Area of triangle:"+t.area());
```

```
        System.out.println("Perimeter of triangle: "
                                                    (15) +t.perimeter());
        Circle c=new Circle(5);
        System.out.println("Area of circle is :" +c.area());
        System.out.println("Circumference of circle is: "
                                                    +c.circumference());
    }
}
```

Output:

```
Area of shape 5:130
Perimeter of shape 5:23
Square Object is created!
Area of square:100
Perimeter of square:40
Area of rectangle: 1000
Perimeter of rectangle:130
Area of triangle:
Perimeter of triangle:
Area of circle is:
Circumference of circle is:
```

Dynamic method dispatch (or) Dynamic binding (or) Runtime polymorphism:

* Binding means connecting a method class to it's definition.
* This binding can be done either at compile time (or) at run time.
* If the binding happens at compile time, it is known as early binding. If the binding is done at runtime it is termed as "late binding."

*In case of method overloading, all the overloaded method have the same name but different signature, so at compile time itself the compiler resolves the overloaded method call by looking at the no. of (or) type of parameters and return type.

*Since binding of overloaded methods is done at compile time, overloaded methods are said to be early bound.

*In case of method overriding, the methods overridden in all the sub classes not only have the same name but also have signature. Hence when a overridden method is called, the binding to it's body cannot done at compile time.

*Also a super class reference variable can hold an object of the sub class. Hence the resolution of a call to an overridden method can be done only at runtime. Therefore overridden methods are said to be late bound.

* Late binding is also known as dynamic method dispatch.

The following program illustrates dynamic method dispatch:

```
/*Program to illustrate dynamic method dispatch*/
class Shape{
    double d1;
    double d2;
    double area(){
        return d1*d2;
    }
    double perimeter(){
        return d1+d2;
    }
}
```

```
class Square extends Shape{
    Square(double d){
        d1=d2=d;
    }
    double perimeter(){
        return 4*d1;
    }
}
class Rectangle extends Shape{
    Rectangle(double l, double b){
        d1=l;
        d2=b;
    }
    double perimeter(){
        return 2*(d1+d2);
    }
}
class Triangle extends Shape{
    double d3;
    Triangle(doube s1, double s2, double s3){
        d1=s1;
        d2=s2;
        d3=s3;
    }
    double area(){
        double s=(d1+d2+d3)/2;
        double a=Math.sqrt(s*(s-d1)*(s-d2)*(s-d
        return a;
    }
    double perimeter(){
        return d1+d2+d3;
    }
}
```

```
class Circle extends Shape{
    Circle(double r){
        d1=d2=r;
    }
    double area(){
        return 3.14*d1*d2;
    }
    double perimeter(){
        return 2*3.14*d1;
    }
}
class DynamicMethodDispatch{
    public static void main(String args[]){
        Shape s; // Reference variable of Shape class
        System.out.println("----***----");
        s=new Square(15); // Super class reference holding sub class object
        System.out.println("Area of square:"+s.area());
        System.out.println("Perimeter of square: "+s.perimeter());
        System.out.println("----***----");
        s=new Rectangle(10,10);
        System.out.println("Area of rectangle: "+s.area());
        System.out.println("Perimeter of rectangle: "+s.perimeter());
        System.out.println("----***----");
        s=new Triangle(10,10,10);
        System.out.println("Area of triangle:"+s.area());
        System.out.println("Perimeter of triangle:"+s.perimeter());
        System.out.println("----***----");
        s=new Circle(10);
        System.out.println("Area of circle:"+s.area());
        System.out.println("Circumference of circle:"
                           +s.perimeter());
    }
}
```

The 'super' keyword:

* The 'super' keyword in Java references to the parent class of the class in which the keyword is used.
* The 'super' keyword is used inside a subclass for the following 3 purposes:

1. To call the super class constructor
2. To access super class variables
3. To call super class methods

Calling super class constructor from child class:

* It is possible to call the constructor of the base class within the constructor of the subclass using the 'super' keyword.
* We can also pass the parameters for the super class constructor using the 'super' keyword as shown below:

```
subclassConstructor(parameters){
    super(parameters); // calling super class
                       constructor by passing parameters
}
```

* When you are using the super keyword inside the subclass constructor make sure that it must be the first statement.
* The following program explains calling super class constructor within the child class constructor:

```
/* Program to illustrate calling base class constructor
from sub class constructor using super keyword */
class Employee{
    int empId;
    String empName;
```

```
    Employee(int eid, String name){
        empId=eid;
        empName=name;
    }
}
class Teacher extends Employee{
    int exp;
    String course;
    Teacher(int eid,String name, int e, String c){ //sub class constructor
        super(eid, name); // calling super class constructor
        exp=e;
        course=c;
    }
    void display(){
        System.out.println("Employee ID:"+empId);
        System.out.println("Name:" +empName);
        System.out.println("Years of exp:"+exp);
        System.out.println("Course:" +course);
    }
}
class SuperDemo1{
    public static void main(String args[]){
        Teacher t=new Teacher(123,"Komali",11,"Software
                                                Engineering");
        System.out.println("----TeacherDetails----");
        t.display();
    }
}
```

To accessing super class methods in the sub class using super keyword:

*When a super class method is overridden in the subclass and the subclass still want to execute the superclass version of the same method, it is done through 'super' keyword.

*Whenever the child wants to invoke the base class version of an overridden method, the method class should preceed the 'super' keyword as shown below:

```
super.methodname(parameters);
```

*The following program illustrates calling base class version of an overridden method inside the sub class using the super keyword.

```
/*Program to illustrate calling super class method from
child class using super keyword */
class Employee{
    int empId;
    String empName;
    Employee(int eid, String name){
        empId=eid;
        empName=name;
    }
    void display(){
        System.out.println("from super class...");
        system.out.println("Employee ID:"+empId);
        System.out.println("Name:"+empName);
    }
}
class Teacher extends Employee{
    int exp;
    String course;
```

```
    Teacher(int eid, String name, int e, String c){ //subclass constructor
        super(eid, name); //calling superclass constructor
        exp = e;
        course = c;
    }
    void display(){ //overriding display method
        super.display();
        System.out.println("From child class...");
        System.out.println("Years of exp: " +exp);
        System.out.println("Course: " +course);
    }
}
class SuperDemo2 {
    public static void main(String args[]){
        Teacher t=new Teacher(123, "Komali", 11, "Software Engineering");
        System.out.println("... Teacher Details.... ");
        t.display();
    }
}
```

Accessing super class variable using super keyword:

*When the sub class and super class have a variable with the same name, name collision occurs.

*When a sub class defines a variable with the same name as one of it's parent's variable then the subclass variable 'shadows' the super class variable that means if you try to access the variable inside the subclass only the subclass version of the variable can be accessed. This is known as "shadowing of variables".

*When a variable is shadowed by a subclass and the sub class still wants to access the same variable of the super class, it is possible 'super' keyword as shown below:

```
super.variableName;
```

Accessing superclass variable using super keyword:

* When the sub class and super class have a variable with the same name,

The following program explains accessing a shadowed variable using super keyword:

```
/* Program to illustrate accessing super class variable
from base class using super keyword */
class A{
    int a=10;
    int b=20;
    void show(){
        System.out.println("From superclass ---");
        System.out.println("a = "+a);
        System.out.println("b = "+b);
    }
}
class B extends A{
    int b=30;
    int c=40;
    void show(){
        super.show();
        System.out.println("From sub class ---");
        System.out.println("a = "+a);
        System.out.println("b = "+b);
        System.out.println("c = "+c);
        System.out.println("Using super keyword b = "
                                    +super.b);
    }
}
class SuperDemo{
    public static void main(String args[]){
        B obj=new B();
        obj.show();
    }
}
```

Output:

From superclass....

a=10

b=20

From sub class....

a=10

b=30

c=40

Using super keyword b=20

The 'final' keyword:

*The final keyword can be used for the following purposes:

1. To define constants in Java
2. To prevent method overriding
3. To prevent class inheritance

1. To define constants:

*In Java, constants are defined as the variables that are declared 'final'. When we declare a variable as final, we must assign a value to it otherwise compiler throws an error.

*The syntax for creating constants using 'final' keyword is given below:

```
final datatype constantName=value;
```

*We cannot change the value of a constant one's after it has been defined. By convention the name of a constant must contain all upper case letters.

Eg: final int X=20;

*If we try to change the value of a constant, the program won't compile

```
X=X+10; //Illegal statement
```

The following program illustrate the creation of constants using final keyword:

```
import java.util.Scanner;
class Circle{
    static final double PI=3.14;
    double rad;
    Circle(double r){
        rad=r;
    }
    double area(){
        return PI*rad*rad;
    }
    double circumference(){
        return 2*PI*rad;
    }
}
class FinalDemo1{
    public static void main(String args[]){
        Scanner s=new Scanner(System.in);
        System.out.println("Enter radius: ");
        double r=s.nextDouble();
        Circle c=new Circle(r);
        System.out.println("Area of the circle:"+c.area());
        System.out.println("Circumference of the circle:"
                +c.circumference());
    }
}
```

Output:

```
Enter radius:
5
Area of the circle:
Circumference of the circle:
```

2. Defining final methods:

*When we declare a method as 'final', it is prevented from overriding in an of the subclass.

*When we are certain that a method in the base class must be used as it is even in the subclass without an change, we can declare such method as 'final'.

*'final' methods of a base class are not allowed to be redefine in the subclasses.

* To define a final method, use 'final' keyword as shown below:

```
final datatype methodName(parameter list){
        //Statements
}
```

* The following code illustrates the final method:

```
class A{
    int x=10;
    final void show(){
        System.out.println("x= "+x);
    }
}
class B extends A{
    int y=20;
    void show(){ //This is illegal
        System.out.println("y= "+y);
                //show() is a final method of A
    } //It can't be overridden in B
}
```

3. Defining final classes:

* When the 'final' keyword is applied to a class, the class is prevented from subclass that means we can't have the subclass of a 'final' class.
* The classes that are declared 'final' can't be inherited.

To declare a class as final use the following syntax:

```
final modifier class className{
    //class body
}
```

Note:

The modifier can be public, private, protected or default.

* The following code segment explains:

```
final class A{
    int x=10;
    final void show(){
        System.out.println("x= "+x);
    }
}
class B extends A{ // This is illegal
    =
}
```

abstract class:

* In Java, the keyword 'abstract' is used to create abstract methods and abstract classes.
* The term 'abstract' refers to a concept or idea that is not associated with any specific instance. The keyword 'abstract' also indicates the same in Java.
* An abstract method is a method which has no implementation or body, for an abstract method we can have return type and parameters as shown in

Syntax given below:

```
abstract returntype methodName(parameter_list);
```

*A class becomes abstract when it posses atleast one abstract method.

*An abstract class cannot be instantiated i.e., we can create objects of an abstract class.

*But abstract class can have sub class.

*An abstract class lets it's subclass(es) to implement or define the body for it's abstract methods.

*Hence it is the responsibility of the subclass to provide implementation for all the abstract methods present in the abstract base class.

*If the sub class fails to provide implementation for any of the abstract methods of the base class then the sub class also becomes abstract, we should declare it as abstract.

The following is an example illustrating the concept of an abstract class.

```
abstract class Shape{
    double d1;
    double d2;
    abstract double area();
    abstract double perimeter();
}
class Square extends Shape{
    Square(double d){
        d1=d2=d;
    }
    //Defining abstract method of base class.
    double area(){
    }    return d1*d2;
```

```
        double perimeter(){
            return 4*d1;
        }
    }
class Rectangle extends Shape{
    Rectangle(double l, double b){
        d1=l;
        d2=b;
    }
    double perimeter(){
        return 2*(d1+d2);
    }
}
class Triangle extends Shape{
    double d3;
    Triangle(double s1, double s2, double s3){
        d1=s1;
        d2=s2;
        d3=s3;
    }
    double area(){
        double s=(d1+d2+d3)/2;
        double a=Math.sqrt(s*(s-d1)*(s-d2)*(s-d3));
        return a;
    }
    double perimeter(){
        return d1+d2+d3;
    }
}
class Circle extends Shape{
    Circle(double r){
        d1=d2=r;
    }
    static final PI=3.14;
    double area(){
```

```
        return PI*d1*d2;
    }
    double circumference(){
        return 2*PI*d1;
    }
}
class AbstractClassDemo{
    public static void main(String args[]){
        System.out.println("----***----");
        Square s=new Square(15);
        System.out.println("Area of square:"+s.area());
        System.out.println("Perimeter of square:"
                                    +s.perimeter());
        System.out.println("----***----");
        Rectangle r=new Rectangle(10,20);
        System.out.println("Area of rectangle:"+r.area());
        System.out.println("Perimeter of rectangle:"
        System.out.println("----***----"+r.perimeter());
        Triangle t=new Triangle(2,2,2);
        System.out.println("Area of triangle:"+t.area());
        System.out.println("Perimeter of triangle:"
        System.out.println("----***----"+t.perimeter());
        Circle c=new Circle(5);
        System.out.println("Area of Circle:"+c.area());
        System.out.println("Perimeter of circle:"
    }                                   +c.perimeter());
}       Shape s; //Reference of abstract class
        s=new Triangle(12,13,9);
        System.out.println("Area of triangle:"+s.area());
        System.out.println("Perimeter of triangle:"+s.perimeter());
```

*The following are some key points about abstract class:

1, They can do instiantiated but they can have reference variable.

2, A class can inherit only one abstract class as multiple inheritance is not allowed among classes.

3, They can have abstract methods as well as non-abstract methods.

4, Overriding all the abstract methods of an abstract base class is mandatory for the subclass. Overriding non-abstract methods is upto the requirement of subclass.

5, Abstract class can have constructors and variables just like any other classes.

Interfaces:

*An interface in Java defines a set of data members and methods just like a class with the exception that all the members of an interfaces must be 'final' (constants) and all the methods are 'abstract' by default.

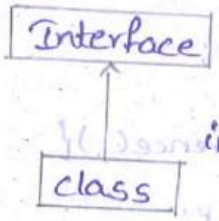
*The members of an interface are public by default. The data members are 'public static final' and the methods are 'public abstract';

*An interface can be defined with the help of the keyword interface.

```
interface InterfaceName {
        //data members
        //methods
}
```

*Classes can implement interfaces by providing implementation (or) definition for all the methods of the interface.

*



*Interfaces cannot be instatiated but we can create reference variable for an interface.

*A class can implement an interface using the 'implements' keyword.

```
class className implements InterfaceName{
        //class body
}
```

*A class that implements an interface mandatory provide definition for the interface methods.

*If the class fails to implement any of the method of the interface that is implements then the class must be declared abstract.

*The following program illustrates defining an interfaces and implementation interfaces.

```
/*Program to demonstrate defining an interface*/
import java.util.Scanner;
//defined an interface
interface CircleInterface{
    double PI=3.14; // public static final double PI=3.14;
    double area(); //public abstract double area();
    double circumference(); //public abstract double circumference();
}
class Circle implements CircleInterface{
    double r;
    Circle(double rad){
```

```
        r=rad;
    }
    public double area1(){
        return PI*r*r;
    }
    public double circumference1(){
        return 2*PI*r;
    }
}
class InterfaceDemo{
    public static void main(String args[]){
        Scanner s=new Scanner(System.in);
        System.out.println("Enter the radius of the circle:");
        double radius=s.nextDouble();
        Circle c=new Circle(radius);
        System.out.println("Area=" +c.area1());
        System.out.println("Circumference=" +c.circumference1());
    }
}
```

Output:

```
Enter the radius of the circle:2.6
Area=21.2264
Circumference=16.328
```

*It is mandatory that the methods of an interface must be defined as final inside a class that implements the interface.

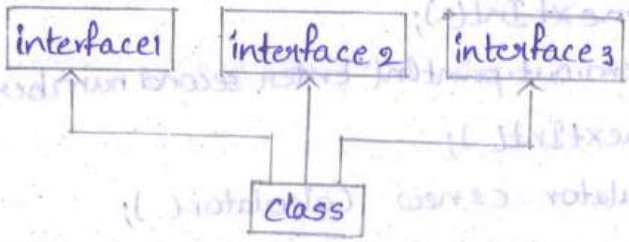
Implementing multiple interfaces:

*A class can implement more than one interfaces this is how java achieves multiple inheritance.

*When a class implements multiple interfaces it has to implement the methods of all those interfaces.

Syntax:

```
class className implements InterfaceName1, InterfaceName2,...
{
  //class body
}
```

*The following program illustrates a class implementing multiple interfaces:

```
import java.util.Scanner;
interface Interface1{
    int add(int a, int b);
    int subtract(int a, int b);
}
interface Interface2{
    int multiply(int a, int b);
    int divide(int a, int b);
}
//class implementing multiple interfaces
class Calculator implements Interface1, Interface2{
    public int add(int a, int b){
        return a+b;
    }
    public int subtract(int a, int b){
        return a-b;
    }
    public int multiply(int a, int b){
        return a*b;
    }
    public int divide(int a, int b){
        return b/a;
    }
}
```

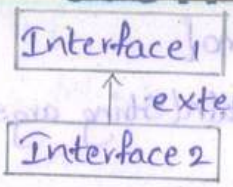
```
class MultipleInterfaceDemo{
    public static void main(String args[]){
        int a,b;
        Scanner s=new Scanner(System.in);
        System.out.println("Enter first number a: ");
        a=s.nextInt();
        System.out.println("Enter second number b: ");
        b=s.nextInt();
        Calculator c=new Calculator();
        System.out.println("Sum of given numbers:"+c.add(a,b));
        System.out.println("Difference of given numbers:"
                +c.subtract(a,b));
        System.out.println("Product of given numbers:"
                +c.multiply(a,b));
        System.out.println("Division of given numbers:"
                +c.divide(a,b));
    }
}
```


Output:

```
Enter first number a:
2
Enter second number b:
4
Sum of given numbers:
6
Difference of given numbers:
2
Product of given numbers:
8
Division of given numbers:
2
```

Extending Interfaces:

* An interface can extends another interface just like a class extends another class in inheritance.



* This can be done using the extends keyword as shown below:

```
interface SubInterfaceName extends BaseInterfaceName
{
  //Interface definition
}
```

* When a sub interface extends a base interface, all the methods and members of base interface will get inherited into the sub interface.
* The following program demonstrates extending interfaces:

```
//program to demonstrate extending interfaces
import java.util.Scanner;
interface Interface1 {
    int add(int a, int b);
    int subtract(int a, int b);
} //An interface extending another interface
interface Interface2 extends Interface1{
    int multiply(int a, int b);
    int divide(int a, int b);
}
class Calculator implements Interface2 {
    public int add(int a, int b) {
        return a+b;
    }
    public int subtract(int a, int b){
        return a-b;
    }
    public int multiply(int a, int b){
        return a*b;
    }
    public int divide(int a, int b){
        return a/b;
    }
}
```

```
class MultipleInterfacesDemo{
    public static void main(String args[ ]){
        int a,b;
        Scanner s=new Scanner(System.in);
        System.out.println("Enter first number a:");
        a=s.nextInt( );
        System.out.println("Enter second number b:");
        b=s.nextInt( );
        Calculator c=new Calculator( );
        System.out.println("Sum of given numbers:"
                                    +c.add(a,b));
        System.out.println("Difference of given numbers:"
                                    +c.subtract(a,b));
        System.out.println("Product of given numbers:"
                                    +c.multiply(a,b));
        System.out.println("Division of given numbers:"
                                    +c.divide(a,b));
    }
}
```

Eg:

Write a JAVA program that illustrates defining a class using both 'extends' and 'implements' keywords.

```
import java.util.Scanner;
class Base{
    int x;
    int y;
    Base(int x, int y){
        this.x=x;
        this.y=y;
    }
    void show( ){
        System.out.println("x="+x);
        System.out.println("y="+y);
    }
}
```

```
interface BaseInter1 {
    void add(int a, int b);
    void sub(int a, int b);
}
interface BaseInter2 {
    void mul(int a, int b);
    void divide(int a, int b);
}
class Subclass extends Base implements BaseInter1, BaseInter2
{
    Subclass(int a, int b) {
        super(a, b)
    }
    public void add(int a, int b) {
        System.out.println("Sum :" +(a+b));
    }
    public void sub(int a, int b) {
        System.out.println("Difference=" +(a-b));
    }
    public void mul(int a, int b) {
        System.out.println("Product=" +(a*b));
    }
    public void divide(int a, int b) {
        System.out.println("Quotient=" +(a/b));
        System.out.println("Remainder=" +(a%b));
    }
}
class Demo {
    public static void main(String args[]) {
        Scanner s=new Scanner(System.in);
        System.out.println("Enter first number:");
        int a=s.nextInt();
        System.out.println("Enter second number:");
        int b=s.nextInt();
        Subclass obj=new Subclass(a, b);
```

```
        obj.show();
        obj.add(this.x, this.y);
        obj.sub(this.x, this.y);
        obj.mul(this.x, this.y);
        obj.divide(this.x, this.y);
    }
}
```


Packages:

*Java packages provides a mechanism for organizing java classes into groups.

*Infact a java package is a directory or folder for holding the java files.

*Package can be defined as a collection used for grouping a variety of class and interfaces based on their functionality

*A package declaration resides at the top of a ~~source~~ java source file. All source files placed in a package have common package name.

Need for packages:

*Two classes in two different packages can have the same name, which is not possible without package mechanism

*Packages provide a way to hide it's classes from being accessed by other programs or classes which donot belong to same package.

*Packages provide a unique name space for all the classes it contains.

*Packages can contain classes, interfaces, enumerated types and annotation (meta data).

Creating packages:

*Packages in Java fall under 2 categories i.e., built in packages (Java APIs) and user defined packages.

* Some of the commonly used Java built-in packages are java.util, java.io, java.lang, java.math, java.awt, java.SQL, javax.swing etc,
* for creating a user defined package, we use the package statement as shown below:

```
package PackageName;
```

Here 'package' is the keyword used to define a java package and packageName is the identifier (or) name of the package.

* Package names must contains all lowercase letters.
* The package statement must be declared at the top of the java file. Anything else may only be followed by the package declaration.

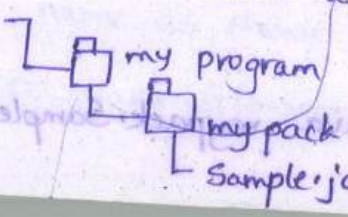
Eg:

```
package mypack;
class Sample {
    int x;
    Sample(int a){
        x=a;
    }
    void show(){
        System.out.print("Value of x=" +x);
    }
}
```

1, Open notepad and type the above code.
2, Save the file as 'Sample.java' in the following location.

E:\myprogram\mypack

3, The directory structure looks as shown below:



* Remember that the file must be saved, name of the class (Eg: Sample.java) and placed in the directory named exactly the same as packages (Eg: my pack)

* There are 2 ways to compile and run the java files stored in a package:

1, At command prompt move to the same directory in which your java file resides and use javac command to compile.

```
E:\>E:
E:\>cd myprogram
E:\myprogram>cd mypack
E:\myprogram\mypack>javac Sample.java
```

For executing the file, move to the directory that is just above your package and execute the file using the syntax:

```
java packageName.className
```

```
E:\myprogram>java mypack.Sample
```

Note:

Classes that reside inside a package cannot be referred by their own name alone. The package name has to precede the name of the class.

2, To compile the file, you can specify '-d' option to the 'javac' command as follows:

```
E:\>javac -d "E:\myprogram\mypack" Sample.java
```

For executing the class the same steps need to be followed

```
E:\myprogram>java mypack.Sample
```

CLASSPATH Variable:

* By default 'javac' & 'java' tools/commands search the file in the current directory only.
* If the classes are not in the current directory then we need to set the classpath.
* Classpath can be set in 2 ways:

1. Setting the value of the environment variable 'CLASSPATH' at the command prompt.

already exist path add with myprogram

```
E:\>SET CLASSPATH=%classpath%;E:\myprogram;
```

2. Use classpath option (-classpath or -cp) with javac (or) java commands to override the user defined classpath.

```
E:\>javac -cp "E:\myprogram" Sample.java
```

(or)

```
E:\>javac -classpath "E:\myprogram" Sample.java
```

Sub packages:

* A subpackage is similar to a sub folder or sub directory within another directory.
* To create a java file in a subpackage, the package statement must contain the parent package name & subpackage name as shown below:

```
package mypack.subpack;
```

Using package:

* To use the class and interfaces of a package, we use the 'import' keyword.
* The keyword import is followed by the package name and the item name as shown below:

Eg:

```
import java.util.Scanner;
```

Here 'java' is a built in package and 'util' is a subpackage of 'java' and 'Scanner' is a class defined in the util package.

* The above import statement import only the Scanner class of util package.

* If you require all the items in a package are to be imported into your program, the import statement should be as shown below:

```
import packageName.*;
```

Eg: import java.util.*;

the above statement makes all the classes and interfaces of the util package available to your program.

Program to illustrate using packages:

Example1:

Step1: Open notepad and type the following code.

```
package mypack;
public class Factorial{
    public int fact(int n){
        if(n==0) return 1;
        else return n*fact(n-1);
    }
}
```

Step2: Create a folder named 'mypack' in the following location.

D:\18525\

Step3: Save the above file as 'Factorial.java' in the following location.

D:\18525\mypack\Factorial.java

Step4: Compile the above file as shown below:

```
C:\>d:
```

```
D:\>cd 18525
D:\18525>cd mypack
D:\18525\mypack>javac Factorial.java
```

Step5: Open a new notepad file & write the following code.

```
import java.util.Scanner;
import mypack.Factorial;
class FactDemo{
    public static void main(String args[ ]){
        Scanner s=new Scanner(System.in);
        Factorial f=new Factorial( );
        System.out.println("Enter a number:");
        int n=s.nextInt( );
        System.out.println("factorial of "+n+"is" +f.fact(n));
    }
}
```

Step6: Save the above program as FactDemo.java in the following location.

```
E:\myprogram\
```

Step7: Compile the above file

```
E:\myprogram>javac -cp D:\18525 FactDemo.java
```

Step8: Run the above program by following the steps shown below:

```
E:\myprogram>SET CLASSPATH=%.classpath%;D:\18525;
E:\myprogram>java FactDemo
```

Output:

```
Enter a number:
6
Factorial of 6 is 720
```

Example 2:

Step1: Open notepad and type the following code.

```
package mymathpack;
interface Circle{
    double PI=3.14;
    double areal );
    double circumference( );
}
```

Step2: Save the above file as

C:\Circle.java

Step3: Compile the above file as follows:

```
c:\>javac -d D:\18525 Circle.java
```

This statement compiles Circle.java and creates a folder named "mymathpath" in D:\18525 and stores Circle.class file in it.

Step4: Open notepad and write below code.

```
package mymathpack;
import mymathpack.Circle;
public class MyCircle implements Circle{
    double r;
    public MyCircle(double rad){
        r=rad;
    }
    public double areal ){
        return PI*r*r;
    }
    public double circumference( ){
        return 2*PI*r;
    }
}
```

Step5: Save the above file as MyCircle.java in the following location.

Step6: Compile MyCircle.java as follows:

D:\18525\mymathpack>javac -cp D:\18525 MyCircle.java

Step7: Again open notepad and type the below code.

```
import java.util.Scanner;
import mymathpack.MyCircle;
class Demo{
   public static void main(String args[ ]){
      Scanner s=new Scanner(System.in);
      System.out.println("Enter the radius: ");
      double radius=s.nextDouble( );
      MyCircle c=new MyCircle(radius);
      System.out.println("Area= "+c.area( ));
      System.out.println("Circumference= "+c.circumference( ));
   }
}
```

Step8: Save this file in E:\myprogram\

Step9: Compile Demo.java

E:\myprogram>javac -cp D:\18525 Demo.java

Step10: Run the Demo.java file as follows:

E:\myprogram>set CLASSPATH=%.classpath%;D:\18525;

E:\myprogram>java Demo

Step11: The program will run properly and produces the output.

Enter radius.

Enter the radius:

10

Area=314

Circumference=62.8

Static Import:

* If you require only static members of a class to be imported to your program then you can use another form of import statement as shown below:

```
import static packageName.ClassName.item;
                    (or)
import static packageName.ClassName.*;
```

* This form of import statement imports only the static member of the specified package (or) class into your program.
* The following program illustrates static import statement

```
package mymathpack;
import static java.lang.Math.*;
class StaticImportDemo{
    public static void main(String args[]){
        System.out.println("Value of PI= "+PI);
        System.out.println("Value of e= "+E);
    }
}
```

* Save the above file as follows:

D:\18525\mymathpack\StaticImportDemo.java

* Compile the above file as follows:

D:\18525\mymathpack>javac StaticImportDemo.java

* Run the file as follows:

D:\18525>java mymathpack.StaticImportDemo

* The output as follows:

Value of PI= 3.141592653589793
Value of e= 2.718281828459045

Access Modifiers / Specifiers:

*Access specifiers define how much a class (or) a method (or) a variable is exposed to other classes & packages.

*Java supports 4 specifiers (or) modifiers for access protection of elements. There are:

1. public
2. private
3. protected
4. default (no modifier)

1. public:

*This modifier is applicable for class, constructor, methods, variables.

*When we declare an element as public it can be accessible everywhere.

2. private:

*This modifier is applicable for coristructor, methods, variables & inner classes.

*Top level classes should not be declared as private.

*When we declare an element as private, it is accessible only within the class where it is defined.

3. protected:

*This modifier can be applicable for constructors, methods, variables & inner classes.

*Top level classes should not be declared as protected.

*When we declare an element as protected, it is accessible by the members of the same package and only by the subclasses of other packages.

4. default (No modifier):

*When we don't use any of the 3 modifiers (no modifier),

default access protection will be applied to the element.

* This modifier can be applied to class, constructors, methods, variables.

* When we don't specify the access modifier for an element, it is only accessible by the members of the same package.

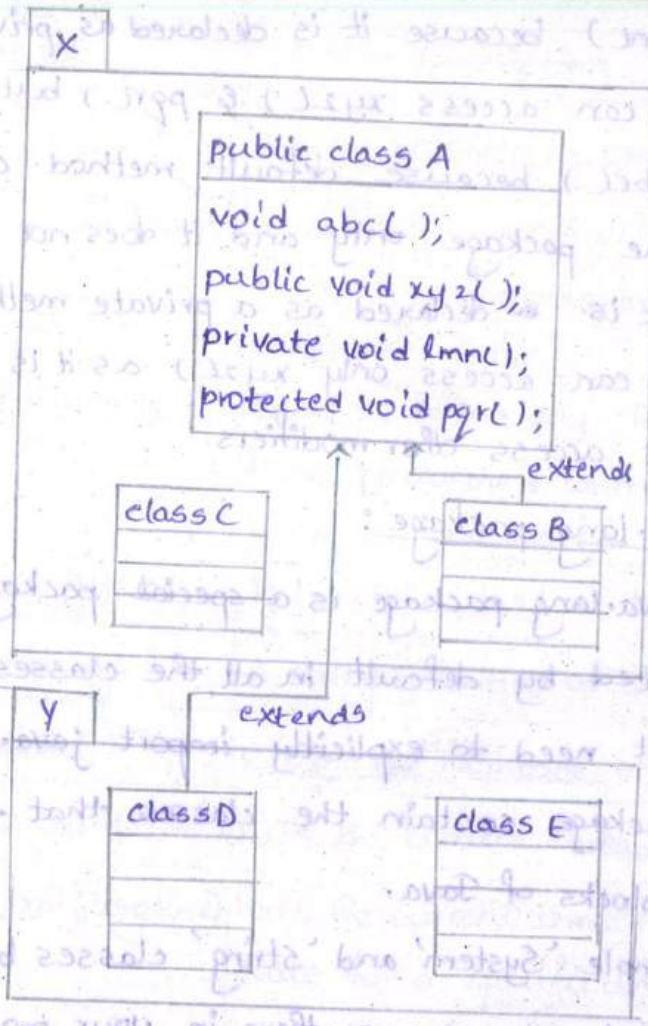
* The following table gives a clear idea about access specifiers of Java.

Access Location	Access Modifier			
	public	private	protected	default
Within the same class	Yes	Yes	Yes	Yes
Subclass in the same package	Yes	No	Yes	Yes
Non-subclass in the same package	Yes	No	Yes	Yes
Subclass of different package	Yes	No	Yes	No
Non-subclass of different package	Yes	No	No	No

Table: Access protection in Java

* The following example illustrates the access protection in Java:

Consider x, y are two packages



*In the above diagram there are two packages one is 'X' and other one is 'y'.

*Package 'x' contains 3 classes i.e., A, B & C i.e., class A contains 2 subclasses B & C. class B extends A but class C does not extends class A.

*Class A contain 4 methods with different access specifiers.

*Package 'y' contains 2 classes i.e D & E

*class D extends the class A which is in the package X.

*Class E does not extends and it is non-subclass.

*class A can access to all the 4 methods.

*class B can access to abc( ), xyz( ) & pqr( ). It does not access lmn( ) because it is private.

*class C can access to abc( ), xyz( ) & pqr( ). It does not

access lmn( ) because it is declared as private.

*Class D can access xyz( ) & pqr( ) but it does not access abc( ) because default method can access only within the package only and it does not access lmn( ) because it is declared as a private method.

*Class E can access only xyz( ) as it is public and it cannot access other modifiers.

The java.lang package:

*The java.lang package is a special package in Java. It is imported by default in all the classes that we create.

*We don't need to explicitly import java.lang package.

*This package contain the classes that form the basic building blocks of Java.

*For example 'System' and 'String' classes belongs to lang package that why we use them in your programs without having the java.lang package imported.

*Some of the important classes of this package are System, Object, String, StringBuffer, StringBuilder etc.,

*All wrapper classes belongs to java.lang package.

The 'Object' class:

*The 'Object' class, the supreme class in Java.

*It is the super class of all built-in and user-defined classes in Java.

*For all the classes that we create, the Object class is the parent by default.

*The following table shows the methods of java.lang.Object:

S.No	Method	Description
1.	Object clone( )	A copy of Object is created & return
2.	boolean equals(Object o)	It compares two objects & return true if both references to the same object otherwise false.
3.	void finalize( )	Used to define any clean up activity to be performed when an object is dead (or) terminated.
4.	final Class getClass( )	Return the class of the object.
5.	String toString( )	A string definition of an object is return
6.	int hashCode( )	Return the hashcode of the object.
7.	final void wait( )	Puts the current thread in waiting state
8.	final void wait(long time)	Puts the current thread in waiting state for a specified time.
9.	final void notify( )	Used to wake up a thread that is in waiting state.
10.	final void notifyAll( )	Used to wake up all the threads that are in waiting state.

Eg:

```
class Demo {
    String toString( ) {
        return "I am Demo object! ";
    }
    public static void main(String args[ ]) {
        Demo d1 = new Demo( );
        System.out.println("d1: " + d1);
        System.out.println("Hash code of d1: " + d1.hashCode( ));
    }
}
```

Output:

>javac Demo.java

>java Demo

: I am a Demo object!

Hashcode of d1:

Wrapper classes:

* Primitive type of Java are non-objects that is why we cannot term Java as a pure object oriented language.
* To bring pure object orientation wrapper classes are designed for each primitive type.
* As the name indicates, the wrapper class is wrapper around a primitive data type.
* The following table list the primitive data type and corresponding wrapper classes.

S.No	Primitive type	Wrapper class
1.	int	java.lang.Integer
2.	long	java.lang.Long
3.	char	java.lang.Character
4.	short	java.lang.Short
5.	byte	java.lang.Byte
6.	float	java.lang.Float
7.	double	java.lang.Double
8.	boolean	java.lang.Boolean
9.	void	java.lang.Void

* For example an 'int' data type can be represented as an instance of 'Integer' class.

Eg: int x=5;

Integer obj = new Integer(x);

Methods of wrapped classes:

a) Converting wrapper objects into primitives

* The methods that are used to convert numeric wrapper object into its corresponding primitive type are: byteValue(), shortValue(), intValue(), longValue(), floatValue(), doubleValue()
* These are all non-static methods.

Eg: Integer iobj = new Integer(5);
Int x = iobj.intValue(); // x contain 5

b) Converting primitive type to string object

* There is one method defined in all wrapper class to convert a primitive type into its corresponding string object representation.
* The name of the method is 'toString()'. It is a static method.

Eg: 1, double y = 235.76;
String s1 = Double.toString(y); // s1 contain "235.76"
2, float y = 235.76f;
String s1 = Float.toString(y); // s1 contain "235.76"

c) Converting string object to primitive type

* There are 6 parse methods corresponding to 6 numeric types. These parse methods are used to convert a string object to the primitive type.
* Those are parseInt(), parseFloat(), parseDouble(), parseShort(), parseLong(), parseByte().

Eg: String s = "595";
int y = Integer.parseInt(s); // contains value 595

d) Converting string representation of primitive type into wrapper object

* There is a method called 'ValueOf()' in all the wrapper class which helps to convert a string representation of a primitive type into its corresponding wrapper object.
* It is a static method.

Eg: String str = "527.37";
Double dobj = Double.ValueOf(str);
System.out.println("Value wrapped in dobj: "
+dobj.doubleValue());

e) Binary and hexadecimal conversions method.

* The 'toBinaryString()' method of 'Integer' class converts an integer to its binary equivalent and returns it as string object.

Eg: System.out.println("Binary Equivalent of 10 is: "
+Integer.toBinaryString(10));

* The 'toHexadecimalString()' method of 'Integer' class converts an integer to its binary equivalent and returns it as string object.

Eg: System.out.println("Hexadecimal Equivalent of 10 is: "
+Integer.toHexadecimalString(10));

Autoboxing and unboxing of wrapper classes:

* the convertion from its primitive type to its corresponding wrapper is known as "autoboxing."
* the convertion from its wrapper type into its corriesponding primitive type is known as "unboxing."
* this convertion to and flow between wrappers and primitives is done automatically from the version java 5.0
* The following example shows autoboxing and unboxing.

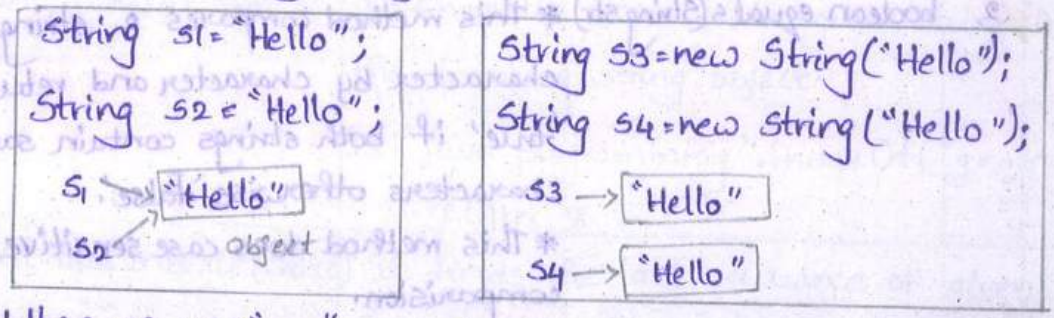
Eg: Integer obj=5; // autoboxing
int x=obj; //unboxing

The 'String' class:

* Strings are immutable objects in Java. A string can be defined as a collection of characters which is treated as a single unit.
* In Java, the strings are treated as objects of 'java.lang.String' class.
* The object of String class are immutable which means that they cannot be changed one after they have been created.
* In Java, string objects can be created in 2 ways:

1. String s1="Hello";
2. String s1=new String("Hello");

* The following figure shows the difference between these 2 ways of creating strings.



When we use "==" operator to compare 2 string objects, it returns 'true', if both the strings references the same object in the memory otherwise returns 'false'.

Eg: s1==s2 returns 'true'
s3==s4 returns 'false'

* Consider the following statement

```
String s1="Hello";
String s2= s1+"World!";
```

In the above 2nd statement, we tried to concatinate s1 and "World!" as s1 is a mutable object, the string literal "World!" cannot be directly appended to s1. But the

statement will not produce any error.

Because it gets converted into the following:

```
String s2 = new StringBuffer().append(s1).append("World!").
                                        toString();
```

where StringBuffer object is used for mutable set of characters.

Methods of 'String' class:

The following table presents some of the commonly used methods of String class.

S.No.	Method Name	Description
1.	int length() (through object reference we use all these methods)	It returns the length of the string as integer.
2.	boolean equals(String str)	* this method compares 2 strings character by character and returns 'true' if both strings contain same characters otherwise 'false'. * This method does case sensitive comparision. Eg: "Hello".equals("hello"); returns false
3	boolean equalsIgnoreCase(String str)	Same as equals() method except the comparision is not casesensitive Eg: "Hello".equalsIgnoreCase("hello"); returns true
4.	int CompareTo(String str)	This method is used to find whether the invoking string is greater than or less than or equal to the argument string. s1.CompareTo(s2) returns: i) a value less than zero, if s1<s2