unit-I :-

Evolution of Java :- Object oriented programming, Two paradigms, The three OOP principles, evolution, Types, Java buzzwords, Java program structure, Implementing a Java program, JVM Architecture, data types, variables, Type conversion and casting, I/O Basics, Reading console input, writing console input, output, Operators, Control statements.

Class, methods, objects, and constructors :- Introducing classes, objects, methods, constructors, garbage collection, finalize() method, Overloading methods and constructors, Argument passing recursion, Static and Kernel final keywords, Nested and inner classes, Arrays. Exploring string and string buffer class, command-line arguments.

Prerequisites :-

Evolution of the 'C' language → It was the miracle in the world of Programming

- It was an structured oriented high level language
  - It would efficiently replace the assembly code when creating system programs
- It changed the way Programming was approached and thought about
  - The aspects of programming languages comparision
    - * Ease of use vs power
    - * Safety vs efficiency / Performance
    - * Rigidity vs extensibility

Fortran → was used for writing scientific applications but was not suitable for the 'system code'.

Assembly language → was used to write efficient Programs but was not easy to learn (or) debug easily

BASIC → Was very easy to learn but is not that Powerful & structured

All early computer languages such as Basic, COBAL, FORTRAN were not structured and relied upon GOTO as Primary means of Program control → Which tended to Produce 'Sphageti' code' — a mass of tangled jumps & conditional branches that make program hard to understand.

While pascal's language were structured they were not designed for efficiency.

The Zeal for creating an efficient computer programming language & A better hardware was on rise as the computers were no longer behind locked doors but were open to all for experimenting.

Created by dennis ritchie

C on DEC PDP-11 running on UNIX OS

↓

ANSI 'C' in 1989

"C language Synthesized the many attributes that had troubled earlier languages"

↓

BCPL martin richards

↓

Influenced by B Ken thompson

Evolution of 'c++' → The necessity for a new language to evolve is complexity and having real-life solving skills.

↓

Throughout the history of programming, the complexity of the programs have driven need for the better ways to manage the complexity.

↓

So the concept of OOP's has been introduced (Object oriented Programming)

↙

A program after reaching an particular level, exceeding it always there is a threshold at which program is unmanageble

↙

C++ broke this threshold, allowing the programmers to comprehend and manage the larger programs

→

C++
invented by
Bjarne stoustrup
1979 in Bell laboratories

C++ blended the high efficiency and stylistic elements of 'C' with object oriented paradigm.

What is Java?

↓ It is widely used Programming language.

↘ It is broadly defined as a general purpose, Object oriented, Platform independent 'concurrent' Programming language that runs very fast.

general purpose :- It can be worked on many domains

Object oriented :- object orientedness helps to solve problems in the real world scenarios

Platform Independent

Java's mantra is

WRITE ONCE RUN ANYWHERE (WORA)

Concurrent :- multi threaded

Very fast :- Java language is almost indistinguishable from 'C' and 'C++'

Familiar syntax → most of the programmers during early 90's using C & C++ hence creaters of Java kept syntax of Java similar to C & C++

Simple & safe → Java developers wanted it to be simple and not complex which makes it safe.

↘ In C, C++, User directly manipulates the memory to free up space

↙ If it is not done correctly results in crashing of the program

→ But in contrast Java has AUTOMATIC MEMORY MANAGEMENT which restricts the direct manipulation of the memory.

↓

By using Garbage collection

Secure → Java programs can be downloaded across the internet and shouldn't cause any kind of damage to the system.

* Java comes with a Rich library, which offers extensive Predefined functionality
* The Java library can be called 'JAVA API', where API is Application program interface.
* Java is Free
* Java was initially designed to run on the embeeded Systems, and later on the web browsers in the form of Java programs called applets

* Java is also used in the Mars rover project called Spirit.

Java's History

Sun microsystems → A silicon-valley based company

↓

upto 1991 it was selling unix-Base computers

↓

And wanted to see next wave in computing!

↓

So they assembled a small team known as "The green team".

↙

Conclusions of green team
* a network of heteregenous devices communicating with each other
* And these devices are small, reliable, distributed, real time embeedded systems.

→ Based on initial conclusions they thought to build a prototype as a part of the new Project

Green project

↙

The vision was to build an interactive handheld device which would communicate with TV and VCR

And software developed by green team would be installed on all these devices

{which is common to change a tv channel, play songs using device in these days but not in early 90's

Goals of Project

- Consume less memory
- delivery of Software components
- Platform Independence
- multithreaded
- security

Initially C++ was considered for this Project

no platform Independency ← But it was rejected as it failed to meet the goals of the project

↓ So a brand new technology was ready to be evolved

One of the team members ← JAMES GOSLING

↓ Created entirely new language and named it as (OAK)

LATer it was renamed to JAVA due to trademark issues.

so James gosling is regarded as the father of Java.

In Sept '92 they came up with a working prototype Called Star 7 (or) goslings Star 7

Sun, initially targeted the cable-Tv industry and the rejected it because the technology was way to Advanced

But the company identified WWW (Worldwide web) was identical to green project i.e, network of heterogenous systems communicating with each other

Before the Java, the internet contained only the Static HTML

And the company thought that by integrating the Java into the HTML they can make the webpages more interactive

So, At Sun world conference it showcased a browser called HOt Java browser

And the browser allowed the Java programs to be embedded in the HTML called as applets which made web pages more interactive

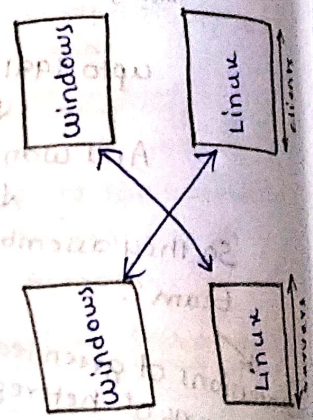
But for this to work the clients Browser Should install Java

↓ An Html page would have html tags Corresponding to applets and applets would have reside in some remote server and when browser Processes an html page and when it encounters tags, then it would download corresponding applets then run

Applets are Platform Independent

* he named it as Oak as there was an Oak tree outside its office.

* goslings star 7 was similar to that of a handheld device with touch screen and features similar to smartphone.



network of heterogenous system communicating with each other.

* HOt Java browser was Showcased in Sunworld conference in 1995.

↓ And netscape browser in the same conference, their announced that their webbrowsser ship with Java

Platform dependency and How Java acheives platform Independency

Compilation → Computer understands Instructions

→ Program is a set of Instructions → which help a computer to perform Something meaningful

Instruction is basically a sequence of 0s & 1s → Binary format

This the language that computer understands → which is called machine code (or) machine langg(or) native code.

So computer sceintists came up with other language called as the assembly language

But it is tedious to write programs in machine language

which was much more easier and expressive

ex:
```
ADD A,B
MOVE C,A
```

But computer understands only 0's and 1's so a program called assembler translates Assembly language into machine language

⊛ Both machine language & Assembly language are referred to the low-level languages. As they deal with low level details. eg: acessing and using memory location

And writting the programs in the Assembly language is Still tedious.

So High level languages evolved gradually.

eg: FORTRAN, c++, Java, c#, C → They used english-like words, math notations, punctuations

They hide the low-level details on the implementation details.

Source code → is the term which is used to refer code written in the Programming language.

Even the source code is not understood by computer.

So a program called compiler translates the source code into machine code

assembly language → Assembler → machine language

* Hiding the implementation details is called as the Abstraction

Source code → Compiler → machine language (or) Target language

Target language
- C, C++ → machine code
- Java → Bytecode
- Other → another Programming language.

Core Compilation operations → Generates machine code
→ code optimizations

↓

Verifying syntax & semantics of source code

Platform Dependency → ON LINUX

↓

ON windows

Hello.c

↓

Compilation (compile & link) ← command to compile: Cl Hello.c

↓ machine code

0100100 Hello.exe

↓

Output

ON LINUX

↓

Hello.c

↓ cc -o Hello Hello.c

Compilation (compile & link)

↓ Hello.exe

0100101

↓

Output

linker links the required header files to the Program to form an final executable file

> Hello.exe (generated on windows machine)

> Hello.exe (generated on linux machine)

NOTE :> Hello.exe generated on windows machine, cannot be run on a linux machine

> Hello.exe generated on linux machine cannot be run on a windows machine

> * And also the processor must be similar if we are going to run the .exe file in another system. If we want to run on a different processor then it should be recompiled on that machine again to run.

This is called as the PLATFORM DEPENDENCY

How interpreter Helps to solve platform dependency

★ Interpreter is a program that "directly executes the source code." without requirement of the cpu execution like in the case of Compiler

★ Intrepreter is nothing But a Virtual machine

★ It is similar to cpu's "Fetch & execute Cycle"

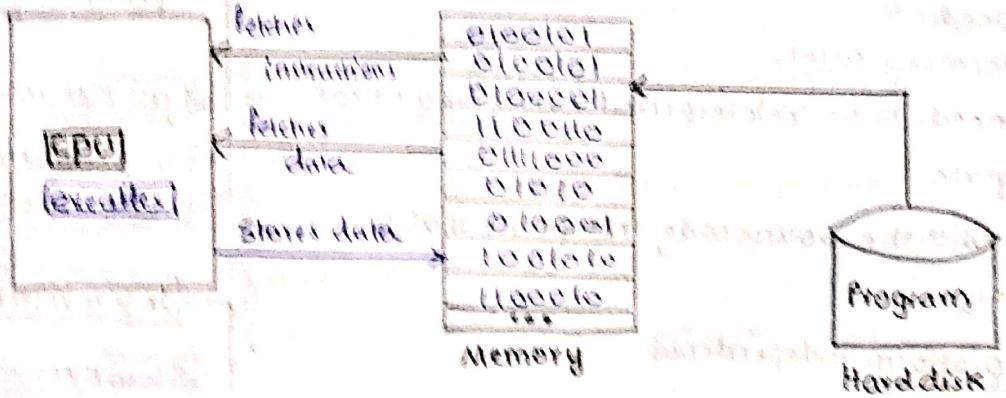
Source code → Compiler → machine code → CPU → Results

Source code → Interpreter → Results

> A 'cpu' turns machine code into results

> An Interpreter turns Source code into results

CPU's fetch and execute cycle → is most fundamental operation



- > when program is started for execution, it is loaded into the memory from harddisk
- > And program is a set of instructions, mainly containing 0's & 1's
- > Then the cpu fetches the instructions to the cpu and executes it. And in the process it also fetches any data required for execution.
- > Any data generated by the execution of program will be written back to the memory.
- > Cpu then fetches the next instruction & executes it.
- > This cycle continues till all the instructions in program are executed.

Interpreters Fetch-and-execute cycle

* Interpreter maintains a library of precompiled machine code.
* Interpreter acheives platform independence.
* Interpreter does →
  The ①, ② steps same as the Compiler, but interprets into its particular interpreted language which can be executed in any platform without any issues.

Pros of interpreter :-

* Platform independence
* no compilation step → so program execution starts very fast
* The code is easier to update → as code is interpreted. (So no need of compiling again).

* According to interpreter An instruction is basically "(An statement in the source code)"
  ① It then understands what is necessary to carry that statement
  ② execute precompiled machine code in its library

Cons of Interpreter :-

* Slow execution speed.
  > due to costly memory access
  > Source code needs to be reinterpreted every single time we run the program.
* Everytime along with the source code, interpreter is also loaded into memory.

Compilation
Fast execution
no platform independence

Interpretation
Slow execution
Platform dependence

How Java acheives platform independence

Java uses both compilation and interpretation to get fast execution and platform independency

The source code is first compiled with compiler and produces Java Bytecode → on Interpretation of this code

Source code → Java compiler → Java Bytecode → JVM → Output

It can be run on any platform using JVM (Java virtual machine). → JVM is platform dependent.

Java runtime environment

Hello.Java → sourcecode file saved with extension (.java)

Hello.Java → compilation → Hello.class (Java bytecode)

Javac Hello.Java → Javac is command used to compile the Java Program

Syntax:-
Javac filename.Java

Hello.class → JVM windows, JVM Linux, JVM Solaris, JVM OS X → Output

To execute the program we use → Java filename

* In case of interpreter we directly execute the source code
* But in case of Java Java byte code is executed by the Java virtual machine

** Note :- In Java filename should be same that of the class name

Key Facts of Java Interpreter → Bytecode interpretation is much faster

- Java bytecode is already compiled
- bytecode is optimized
- Bytecode is compact.

To further speed up the compilation JVM uses JIT compilation → Just in time compilation

Java virtual machine → It is so sophisticated that programs written in other languages like scala and groovy are also executed using JVM.

- JVM is called virtual because it is Abstract computing machine
- JVM is the cornerstone of the Java platform

→ For a computer machine code acts as the instruction set

But for the JVM ← Instruction set

Java bytecode acts as the instruction set. → Same as the real computing machine

JVM manipulates memory at runtime.

Features of JVM → loading and interpreting Java bytecode

- Security
- Automatic memory management.

Components of JVM → Abstract JVM specification

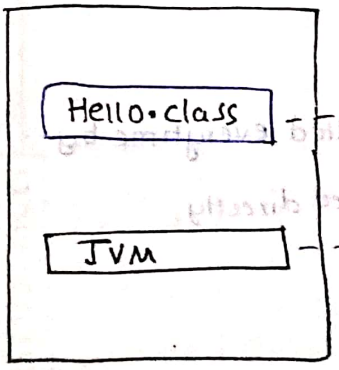
↓ It contains documentation about the JVM and also bytecode instruction set.

- Java language specification → It gives the details of the syntax and semantics of the Java language.
- Concrete implementation
  - Oracle's Hotspot JVM
  - IBM's JVM
- Runtime instance is concrete implementation of JVM

Runtime instance → JVM implements one Java program at a time

↓ runtime instance ↑

java.hello → when executed



→ Then it loads bytecode into memory

→ An instance of JVM is created in the memory

Just-in-time compilation

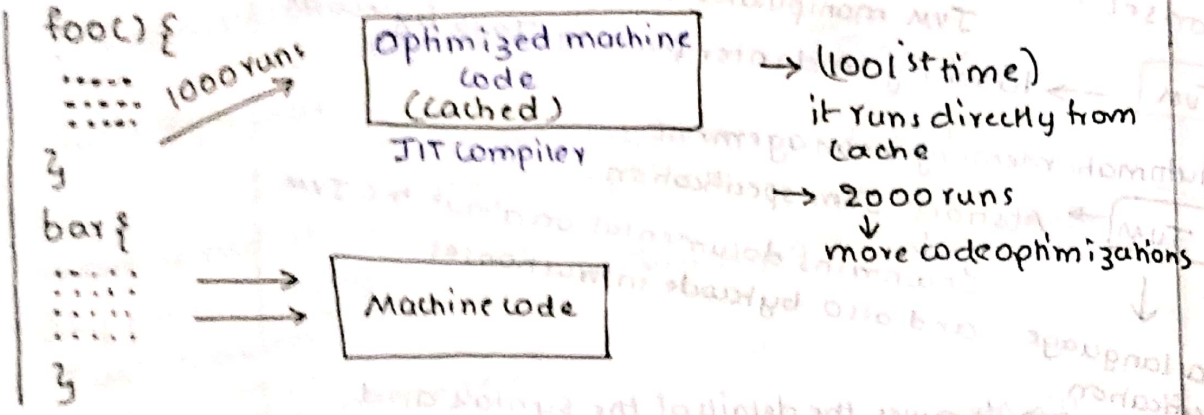
frequently executed bytecode are called 'hotspots'

↳ It identifies the frequently executed code, Precisely frequently executed bytecode → hotspots are given to a Subcomponent of JVM called JIT compiler

JIT compiler then converts these hotspots to machine code ← And this machine code is cached

And next time the cached machine code is directly executed resulting in much faster performance → And the rest of the code is still interpreted by the Java interpreter

ex:- using 2 methods foo() and bar()

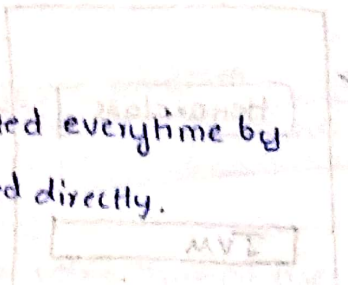


★ for the 1000 runs the code is Interpreted and then executed and the optimized machine code is cached.

★ for the 1001st time the cached code is directly executed. And it is already in machine code format it executes very fastly.

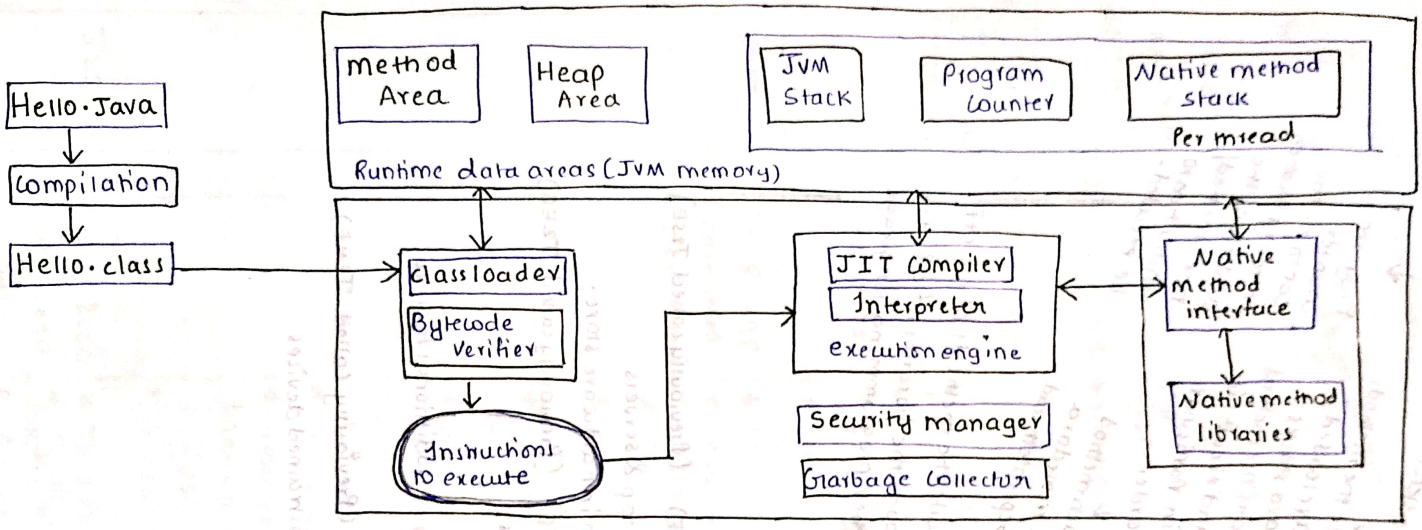
★ further after the 2000 runs the optimized machine code is updated with more optimizations from the identified hotspots.

★: So the cached code need not to be executed everytime by Interpreting everytime, But can be executed directly.



★★★
★ The frequently executed code IS NOT INTERPRETED every single time.

The JIT compilation is also referred as the dynamic compilation
↓
But dynamic compilation is not defined in JVM's specification



JVM ARCHITECTURE.

★ when the Java code is compiled, an instance of Jvm is created. Jvm then invokes a subcomponent called classloader which loads the bytecode. Then it is sent into bytecode verifier, whether the bytecode contains the proper structure and complies with the Java specifications. This is used to check the integrity of Jvm, because .class file can be downloaded from internet which may contain malicious code. After the verification is complete, it can be safely executed by the execution engine. (includes JIT compiler along with interpreter). 'Garbage collector' is responsible for the automatic memory management. Security manager allows the user to run untrusted bytecode, As long as the bytecode doesnot perform any dangerous operations. eg:- We can restrict the bytecode from accessing our file system. And this is done by the security manager by running it in a more restricted environment called Sandboxing environment.

★ And there are different memory areas allocated by the Jvm.

Work of the different runtime areas

- **Method area**
  - It loads all the class code, methods & variables
- **Heap area**
  - It is a temporary memory area and this area creates memory for the object
  - JVM loads method area and heap area as soon as .class file is created.
- **Java stacks**
  - Once the methods are loaded, Java stacks calls the method and executes it in particular order.
  - each method is treated as a seperate thread.
- **Program counter**
  - Program counter contains the memory location address of the next method instruction to be executed.

Native method → If we want to execute the methods of C & C++ in Java the native method loads the libraries of C & C++ by which we can execute other programming language methods in Java.

Java software Family

> Java standard edition (Java SE) (Previously called J2SE)
- o Standalone applications for desktop & servers
  eg: inventory management system in hardware store.

> Java enterprise edition (Java EE) (Previously called J2EE)
- o enterprise applications for servers
  eg:- e-commerce websites
- o It also includes Java SE and other additional tools to develop enterprise applications

> Java micro edition (Java ME) (Previously called J2ME)
- o Applications for resource - constrained devices
  eg: Cell phones and mobiles.

Java SE → Java runtime environment (JRE)

↓

Java Development Kit (JDK).

↓

It is also referred as SDK (software development kit)

↓

Used to develop and run Java applications

→ It contains additional tools such as Java compiler.

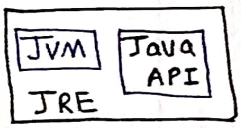
→ is basically a runtime environment

↓

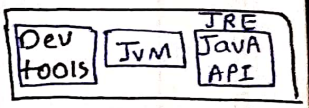
To run the Java applications we need to install JRE

eg:- Webservers using Javascript, (or) servers using applets install JRE

JRE contains



JDK contains.



Sno.	JDK version	relase data	New features updated.
1.	JDK 1.0	Jan 23, 1996	Initial release (codenamed OAK).
2	JDK 1.1	Feb 19, 1997 [codenamed Playground].	* JDBC (Java database Connectivity) * Inner classes * Java beans * RMI (Remote method Invocation). * Reflection (introspection only).
3.	J2SE 1.2	Dec 8, 1998 [codenamed Playground]	* Collections framework. * Java string memory map for constants. * JIT (Just in time) compiler. * Jar signer for signing Java Archive (JAR) files. * Policy tool for accessing grant Permission to system resources. * Java Foundation Classes (JFC) which consists of swing 1.0, Drag and drop, and Java 2D class libraries. * Java plugin. * Scrollable result sets, BLOB, CLOB, batch update, user defined types in JDBC. * Audio support in Applets.
4.	J2SE 1.3	may 8, 2000 [codenamed Kestrel]	* Java sound * jar indexing * A huge list of enhancements in all areas of Java.

Sno	JDK version	release data	New...
5	J2SE 1.4	[First release under JCP] Feb 6, 2002 [codenamed merlin]	* XML processing * Java print service * logging API * Java webstart. * JDBC 3.0 API * Assertions * Preferences API * Chained exception * IPV6 Support. * Regular expressions * Image I/O API.
6.	J2SE 5.0	Sept 30, 2004 [code named Tiger]	* Generics * enhanced for loop * Autoboxing/unboxing * Typesafe enums. * Varags * Static import * metadata (Annotations) * Instrumentation.
7.	Java SE 6	Dec 11, 2006 [code named MUSTANG]	* Scripting language support * JDBC 4.0 API * Java Compiler API * pluggable Annotations. * Native PKI, Java Gss, Kerberos, and LDAP support * Integrated web support. * Lot more enhancements.
8.	Java SE 7	July 28, 2011 [Codenamed Dolphin].	* Strings in switch statements * Type inference for Generic instance creation. * multiple exception handling * Support for dynamic languages * Try with resources * Java Nio package. * Binary literals, underscore in literals * Diamond syntax * Automatic null handling

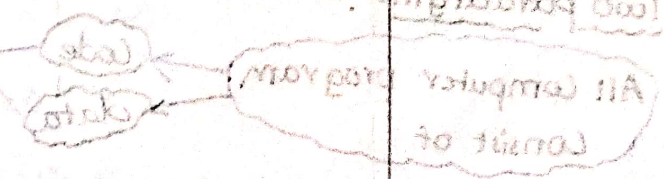
sno	JDK version	release date	new features updated
9.	Java SE 8	18 march, 2014 [codename culture is dropped with Java SE 8]	* Lambda expressions * pipelines and streams * Date and time API * Default methods * Type Annotations * Nashhorn Javascriptengine * Concurrent Accumulators. * parellel operations * permgen error removed * TSL SNI
10.	Java SE 9	upcoming July 27, 2017	* Java + RPEL = Jshell * micro benchmarks * G1 Default garbage collector * HTTP 2.0 * Process API

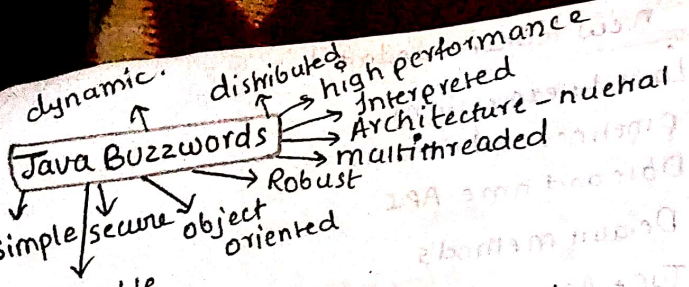
Installing the JAVA :-

Step I :- Install Java. (Download Java JDK 8 (or 7).

Step II :- make Command prompt aware of Java executables
* set Path & JAVA_HOME environment variables
[make command prompt aware] ....

Step - II may be skipped if we use a Java IDE
↓
Integrated development environment.





⇒ OBJECT-ORIENTED language :-

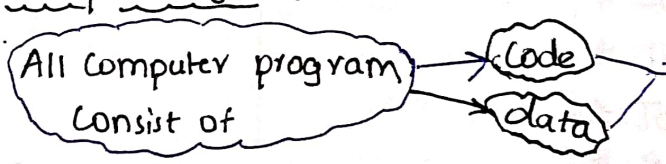
The main problem associated with the larger programs in the structure-oriented language is the 'under evaluation of data'. There is no built in mechanism to ensure the security of data. In the object-oriented programming language emphasis is on data and not on procedures.

> In object-oriented programming, instead of deciding the Procedures first and then deciding about data to suit these procedures, the oop approach is to design a data that corresponds to the essential features of the problem.
> In oop, a class is the specification of such a data form.
> class is a specification of the data entity.
> This data entity is called object.
> An object is an instance of a class
> The relationship between class and object is same as that of a built in data type and a variable of that type.
> A class is an data type and an object is the instance of the data type.
> A class's data is called member data or class data
> The functions that operate on these data are called methods or member functions.

Two most power features of the oop language :-

⇒ Encapsulation (one of the oop-principle).
⇒ Data Hiding. (Abstraction)

Two paradigms :-



→ A program can be conceptually organized around its code or around its data.

Paradigm's me
A typical exam
Pattren, or patte
model.

oop dates back
to 1960's

oops was used to
implement large
Projects in a simple

Some programs are written around → what is happening
→ who is being effected

These are the two paradigms that governs how a program is constructed.

- Process-oriented model
- Object-oriented model

* Process oriented model can be thought as code acting on the data.
* Object-oriented model organized the program around its data and a set of well-defined interfaces to the data. (or) data controlling access to the code.

⇒ The three OOP principles → encapsulation
→ Inheritance
→ Polymorphism.

encapsulation → is the mechanism that binds the code and the data it manipulates, and keeps both safe from outside inheritance and misuse.

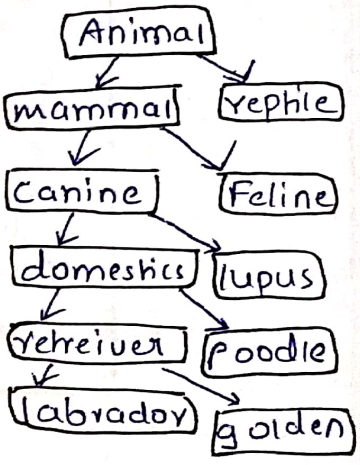
→ The basis of the encapsulation is the class.

→ "A class defines the structure and behaviour (code & data) that will be shared by set of objects".

→ "Each object of a given class contains the structure & behaviour defined by class, in shape of class".

→ "Thus class is a logical construct and Object is a physical reality".

→ "When you create a class, you will specify the code and the data constitute the class". These elements are called the members of the class. The data defined by class is referred to as member variables or data members



Inheritance → is the process by which one object aquires the property of another object. It supports the concept of hierarchial classification

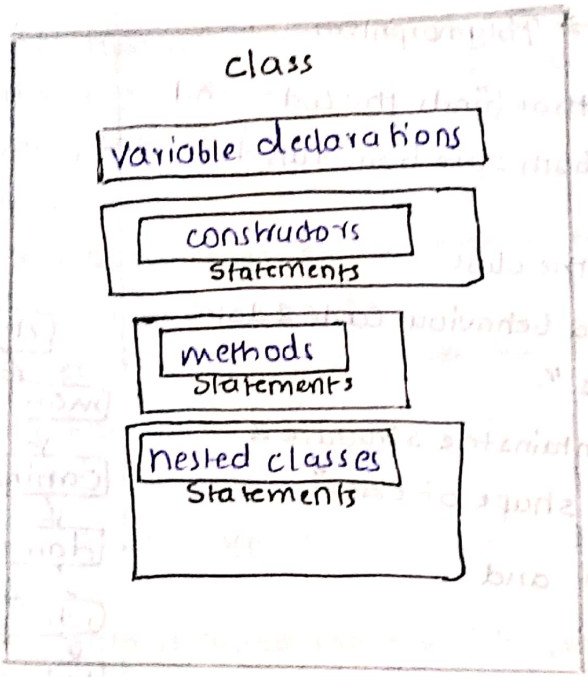
> Inheritance interacts with the encapsulation

> By using encapsulation, inheritance public methods can be used to protect private data

Polymorphism :- It is a feature that allow one interface to be used for a general class of actions. It is commonly referred to as "one interface, multiple methods". This greatly helps to reduce the program's complexity by allowing by allowing same interface to specify the general class of action".

Note :- In the object oriented programming language Inheritance, encapsulation, polymorphism are worked together to produce a robust programming environment

Structure of a Java program :-



Class
member variables
Toget
Members

> In Java program, we declare atleast one or more class.
> And for an objects we write in class, we have to create one or more objects

Simple Java program to print the Hello-world : (Guide)

every class begins with keyword 'class'.
And every class hadan beginning brace and an ending brace

```
// Program 1 : Helloworld.
Class Helloworld
{
  Public static void main(String[] args)
  {
    System.out.println("Helloworld");
  }
}
```

Save it in the directory and with extension .Java

To Compile use the command Javac

Syntax:-

Javac <filename>.Java

ex:-

Javac Helloworld.Java

↓

Then .class file will be generated with Java bytecode

ex: Helloworld.class. — It can be executed on the any Platform now.

To execute it use Java command

Syntax:- Java <filename>

ex:- Java Helloworld.

M.Imp

Note:- If we again edit the program to Helloworld1, i.e, change the classname and then compile it then it produces Helloworld1.class. so you need to delete your old class file Helloworld.class. (as we dont need it anymore).

> Typically we can use any name as classname and different name for the filename. in the above case of program 1

> But consider following program (modified of program 1)

```
// Program 1.2 :- Hello world.
Public class Helloworld1 {
  Public Static void main(String[] args) {
    System.out.Println("Hello world");
  }
}
```

> If we compile the above program 1.2 it shows an error.

```
Helloworld.Java:1: error: class helloworld is public,
Should be declared in a filenamed Helloworld.Java
  Public class Helloworld{
         ^
1 error
```

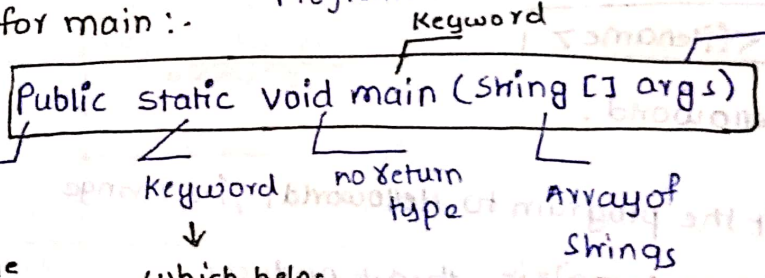
> So if you have the modifier public, then both of class name and filename should be same.

> "But in Java it is preferred to write classname and filename same for every program".

> we can have more than one class in a Java program but it generates 2 .class files. But generally we shouldn't do so.

⇒ main() method :- → Program starts with main()
→ main method must be declared Public static and void.
→ Program ends with the main

Syntax for main :-



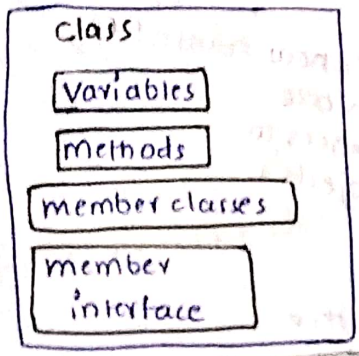
access specifier → which helps the Jvm to invoke the main method. If not return the Program would compile but not run.

Keyword → which helps the Jvm to call the main function. If not, the main function is not called until unless the object is created (which happens actually inside the main()) inside the main function

no return type

Array of Strings

class data:



comments → are used for the documentation purposes

double slash (//) it means to ignore the rest of the line

→ /* */ (block quotes)

Ignore everything between /* and */

*** Everything In the Java is case sensitive **

→ ex variable testing is different from variable Testing

⇒ naming rules :-

> classes, methods and variables names
- must start with letter, underscore or $
- Other charecters can be letters, underscores, $ or numbers

> we cannot use the reserved keywords

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

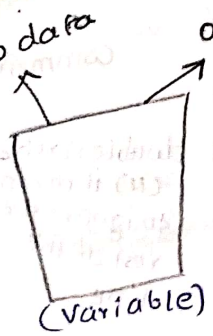
* not used in Java
** introduced in Java 1.4
*** Introduced in 1.2 version
**** not used in Java

⇒ Variables → defines the state of an object

eg:- name of student, age of student, Id of an student.

So basically variables are containers that store the data

variables can hold two kinds of data



eg:- stud s = new student();
(which are references to objects)

↓

A variable has always a type associated with it and basically it indicates what kind of data can be stored in it → once declared we cannot change the type of the variable, hence it is called as the statically typed language. (Java)

so the Java compiler wouldn't allow us to assign one data to the another type data variable → which is called 'static type checking'

Static typechecking is helpful during method overloading

It allows earlier checking of Programming mistakes, Instead of dynamically type checking in dynamically typed languages eg: Javascript.

→ It also helps better developer experiences in IDE's such as eclipse

Refractoring:- Changing the / restructuring the entire code without change in the external behaviour.

so some beleive that maintainability is easy in statically typed languages than dynamically typed languages

Variable declaration:-

<type> <name> = [literal or expression]

Assignment statement

<name> = [literal (or) expression]

used to change the value of variables

is the 'raw data'
eg:-
int count = 25
boolean flag = true;

which is evaluated to a single value
eg:-
int count = getcount();

declaration statements (can be anywhere in the class

Variable types (3 types)

- Instance / static variables
  - referred to as fields or attributes
- local variables

* static : dictionary meaning : lack in movement or change

Instance variables

- are declared directly within the body of the class
- we cannot declare them in the methods
- They are declared at class level but not at method level
- They represent each specific Object state of the class
- They always have a default value when not initialized
- cannot be reinitiallized directly within the class
- reinitiallization can be done in methods.

Static variables

- They are same as the instance variables which are declared directly within the class But, with a Keyword 'static'.
- They are also referred to as the class variables
  - which is declared within class and shared by all the objects in the class
- → whereas 'Instance variables' "are specific to each object" and represent their Object state
- For the static variables only one copy is maintained per class regardless of the Instances of variable in that class
- → static variables value are shared among the objects of class whereas the Instance variables are unique to each object in the class
- like the Instance variables, Static variables are also gets the default value if they are not explicitly Initialized.
- → They are also cannot be reinitiallized directly within the class " like that of the Instance variables.

local variables → They are declared within the methods

They also contains method parameters (within parenmesis).

→ These variables are temporary and can't be accessed outside the methods

They stay in the memory as long as the control of Program is in method.

→ unlike the instance, & static variables they don't get default values

So they must be initialized before they are used

eg:-

```
class Student {
  // Instance variable
int id;  // as not initialized it takes default value '0'.
id = 34; // Illegal statement**
Static int count = 95; // Staic variable*

boolean updateProfile(String newName)
{ // newName is also local variable
   id = 22; // legal statement
   boolean result = true; // result is local variable
   name = newName;
   return result;
  }
}
```

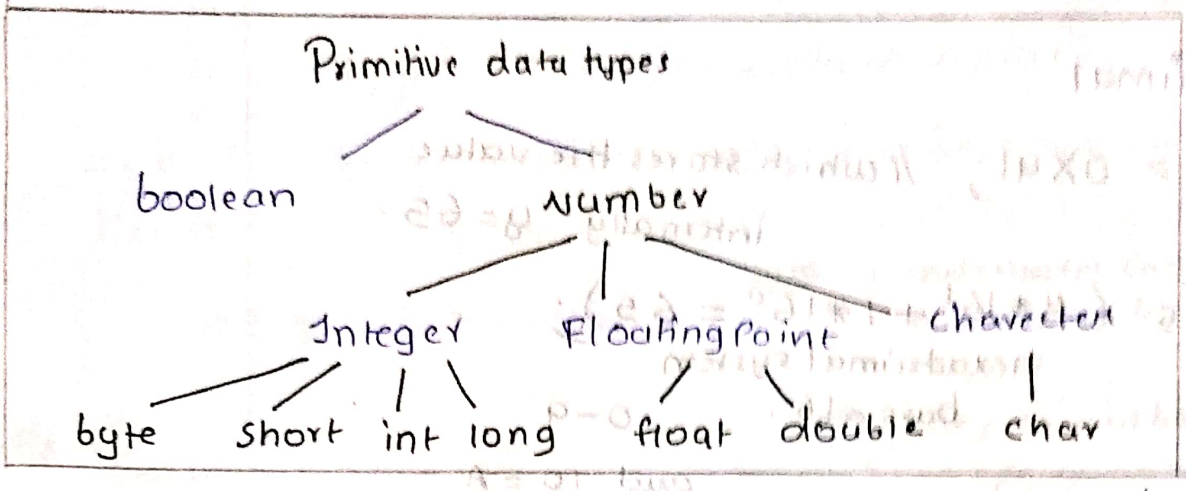
** variables of local, static, instance, BASED ON VALUE TYPE THEY STORE THEY ART TWO TYPES (DATATYPES)

* Primitive variable types (datatypes)
* non-primitive variable types (Object references/datatypes)

** The state... is illegal as... can reinitia... the variable... within men... but not with... the clas...

* even the... variable can... be reinitia... at class lev...

Java defines 8 primitive data types for variables :-



★ except for the boolean the rest of the types are numeric datatypes.

★ "Even char which is used to represent a character, as it is represented internally as unsigned integer" RND

So char is an numeric data type.

★ To reduce the memory constraint and memory wastage there are 4 types of integers declared, from ability to store small value to the largest integer value.

Integers → They are whole or fixed point numbers

They can be represented as byte, short, int & long

★★ "Internally Java uses "signed two's complement" scheme to represent integers".

Type	Bit size	value range	default value
byte	8 bits	-2^7 to 2^7-1	0
short	16 bits	-2^{15} to $2^{15}-1$	0
int	32 bits	-2^{31} to $2^{31}-1$	0
long	64 bits	-2^{63} to $2^{63}-1$	0

eg: byte b = 100;
short s = 1000;
int i = -10000;
long l = 1000000L → small 'l' or capital 'L' should be placed always for a long integer ★★

★★ If the value falls off range, we get a compile error ★★

RND :-
It is said everything in Java is internally represented as "strings".

byte :-
-128 to +127
short :-
-32,768 to +32,76

★★ 'L' is only needed at end if the value is outside int's rang Otherwise no need to write 'L'.

"Literal is the value passed to an variable".

⇒ Other integer literal forms :-

* hexadecimal

int y = 0X41; // which stores the value internally y = 65.

y = 65 (4 * 16^1 + 1 * 16^0 = 65);

hexadecimal system

base = 16. 0-9 and 10 = A, 11 = B, 12 = C, 13 = D, 14 = E, 15 = F

0X — Should be return before hexadecimal number

Zero + letter (small/capital)

* Binary (Introduced in Java 7)

int y = ob 01000001 ;

ob — Should be written before hexadecimal number

Zero + letter (small or capital)

ob 01000001

2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0

64 + 1 = 65.

* Octal.

int y = 0101 // starting always with zero.

(base 8)... converted to decimal gives = 65.

⇒ underscores for readability :-

> In the Java 7 version, a new feature is introduced which is used to represent large numbers easily and read easily

for example we generally write

1, 23, 456 (Basically in maths)

So in Java we can represent large numbers something like this

```
int y = 1_23_456;
```

Note: 1. underscores can only be used for integers
2. They cannot be at beginning or ending but only middle.

eg:-
Consider

```
int x = y = 99
```

99 is first assigned to y and later then to x
and this way we can reinitialize the instance variables
as above is an declarative statement.

eg:-

```
Class Basicsdemo {
Public static void main (string args[])
{
Primitives();
}
Static void Primitives() {
System.out.println("In Primitives.....");
int intHex = 0x0041;
System.out.Println("intHex: "+intHex);
int intBinary = 0b01000001;
System.out.println("intBinary: "+intBinary);
int intUnderscore = 1_23_456;
System.out.println("intUnderscore: "+intUnderscore);
}
}
```

Output

```
Primitives
intHex = 65
intBinary = 65
intUnderscore = 123456
```

ev: from: http://www.ntu.edu.sg/home/ehchua/Programming/Java/J1a_introduction_1.html

Floating-point data types → They are real numbers

→ They can be represented by

- 32-bit float
- 64-bit double → as more bits double is more precise than float.

* Internally Java uses "32 & 64 bit IEEE 754 floating point" scheme to represent the floating point internally.

Type	Bit size	value range	default value
float	32 bits	−3.4E38 to 3.4E38 *	0.0f
double	64 bits	−1.7E308 to 1.7E308	0.0d

eg: float f = 3.18f

⇒ Rules for thumb (both integers and float) :-

> Item 48: Avoid float and double if exact answers are required.

* float and double are ill-suited for monetary calculations (i.e, e-commerce applications)
* And if uses a BigDecimal (which is a class which comes with the Java library).
* We can use int or but we need to keep track of the decimal ourself.

> Floating point is not as fast as integer arithemetic

> double is more precise than float → used in 'nueral networks'

> We can use byte, short, float if memory saving is important.

* here exp... notation is... ∴ −3.4E3... = −3.4... and 3.4E3... = 3.4... "E can be... upper case... lower case...

Charecter & Boolean types → would consist of single letter charecters eg: 'a', 'A', '0'

→ In Java they are represented by 16 bit char data type

** "Internally Java uses "16-bit unsigned integer scheme to represent the charecter". ** (only +ve unsigned int)

Type	Bit size	value range	default value
char	16 bits	0 to $2^{16}-1$	'\u0000' (null charecter)

Char range 0 to 65535

Charecter → 16-bit unicode → 16-bit unsigned integer → stored as string

* here \u → unicode escape sequence, and number following it is a hexadecimal number.

> Java uses 16-bit Unicode (UTF-16) scheme to represent charecters

Unicode → represents all the charecters in all the languages

→ UTF-16 encoding scheme → even C#, Python uses (UTF-16) scheme

of unicode used by Java is one of implementation

eg: 'A' → 0041 → 00000000 01000001

eg:-
```
Char c = 'A'; // Always in single quotes
Char c = 65; // Prints A, in Unicode 65-A
Char c = '\u0041'; // (4*16^1 + 1*16^0) = 65 = A.
```
↓ we can use this representation for emojis and other typical languages such as Japanese.
```
Char c = 0X41; // 4*16^1 + 1*16^0 → 65 → A
Char c = 0b01000001; // 65 → A.
```

unicode table at http://unicode-table.com

```
class Basicsdemo {
Public static void main (string args[])
{
  Primitives();
}
Static void primitives();
{
  char charA = 'A';
  System.out.Println ("charA :" +charA);
  char charInt = 65;
  System.out.Println(" charInt : " +charInt);
  char charUnicode1 = '\u0041';
  System.out.println ("charUnicode1 : " +charUnicode1);
  char charUnicode2 = 0x41;
  System.out.println ("charUnicode2 : " +charUnicode2);
  char charBinary = 0b01000001;
  System.out.Println ("charBinary :" +charBinary);
}
}
```

output

```
charA : A
charInt : A
charUnicode1 : A
charUnicode2 : A
charBinary : A
```

Boolean → It is a binary data type
→ which is either true or false

Type	bit size	value range	default value
boolean	JVM specific	true or false	false

eg:- boolean result = false;

Variables & Typecasting :-

Even though Java is a statically typed language i.e, a variable of one type declared once, cannot be used to store values of other data types

> But sometimes we need to use other data to be stored in variable of one data type.

> so we cast / convert the value.

Type casting :- Assign variable or literal of one type of variable to another type

eg: int ← long
int ← byte

> Type casting is only possible in numeric to numeric Primitive types

> cannot cast anything to boolean or vice versa.

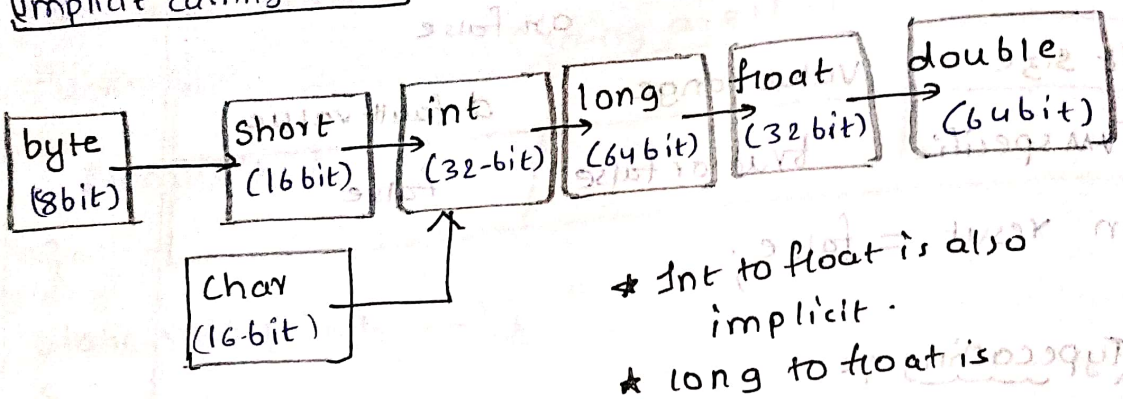
> Type casting is two types.

* Implicit casting (widening)
* Explicit casting (narrowing)

⇒ Implicit casting : when converting from smaller data type to larger data type.

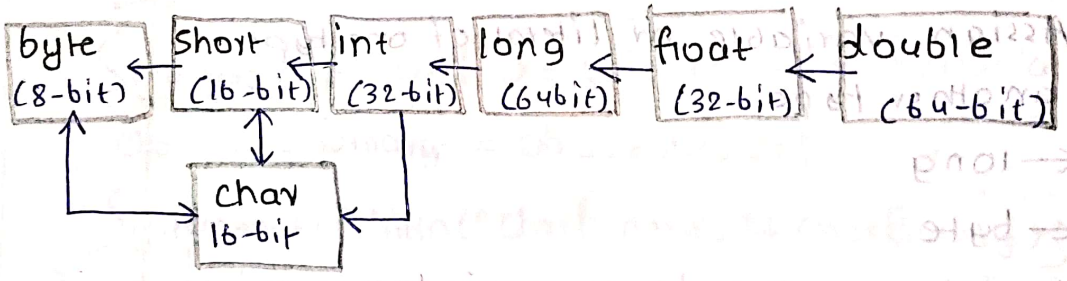
eg: int x = 65
long y = x; (Implicit casting, done by compiler)

Implicit casting order



* Int to float is also implicit.
* long to float is also implicit.
* char to int is also implicit and char to long, float, double also implicit.

Explicit casting order



⇒ larger to smaller type is called explicit conversion

eg: 1. long y = 42;
int x = (int) y;

2. byte b = 65;
char c = (char) b; // c = A. (both widening & narrowing)
byte → int → char

3. char c = 65; // c = A
4. short s = 'A'; // s = 65

} no cast needed.

Information loss in explicit casting:

> out of range assignments

- byte narrowedByte = (byte) 123456 // (64) → loss of data

> Truncation

when we try to cast a floating-point variable to the integer/char will always truncate.

↓ Because JVM discards all lower bits of number except lower 8 bit.

eg: int x = (int) 3.14f ; // x = 3
int y = (int) 0.9; // y = 0
char c = (char) 65.5 // c = 65 = A

Note: To be precise for some implicit castings, there will be data loss

eg: int → float
long → float.
long → double
as it looses least significant values

eg: int oldval = 1234567890;
float f = oldval // implicit cast
int newVal = int f; // 1234567936.

↓ loss of Precision

Cases where we use Casting

Implicit casting: In the functions

explicit casting: double avg = (2+3)/2 ⇒ 2.0

So double avg = (double) (2+3)/2 ⇒ 2.5

```
class Basicsdemo
{
  Public Static void main(String args[])
  {
    typecasting();
  }
  Static void typecasting()
  {
  System.out.println("typecasting...");
  //explicitcasting
  long y = 42;
  int x=y;  //int x = (int) y; → To avoid error*
  // Information loss due to over-range assignment
  byte narrowedByte = (byte) 123456 **  //or 127
  System.out.println("narrowedByte : " +narrowedByte);
  //Truncation
  int itruncated = (int) 0.99;
  System.out.println("itruncated : " +itruncated);
  // implicit cast (int to long)
  y=x;
  // implicit cast (char to int).
  char cChar ='A';
  int iInt = cChar.
  System.out.println("iInt : "+iInt);
  // byte to char using an explicit cast
  byte bByte = 65;
  cChar = (char) bByte; // special Conversion (widening
from Byte → int followed by narrowing from int → char)
  System.out.println("cChar : "+cChar);
}
```

** error : incompatible types : possible lossy conversion from long to int int x = y.

** If we dont insert a (byte) caste it shows an error

Same as above

byte narrowedByte = 12345 6

error : lossy conversion

when executed it gives

(64) → unknown number

So we should give between - 127 to 128

∴ byte narrowedByte = 127

so output

narrowedByte : 64
itruncated : 0 // as it takes (0)
iInt : 65
cChar : A.

Variables Object references

> Consider a statement

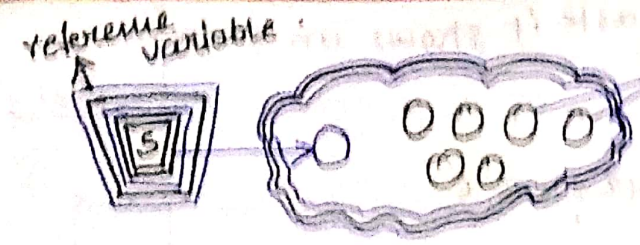
Student S = new student();

- Student S → Allocates the space for the reference variable
- S ← Student Objects adress
- new student() → Allocate space for new student object

> So we need to allocate and assign memory location address to the object and reference variable, so we use assignment operator "=".

> whenever the Jvm gets started it gets a chunk of memory from its operating system for the programs to be executed.

> All objects are stored on the heap



** > Bit depth of the object references are JVM specific i.e, Bit depth of one object reference may be larger/smaller than other object reference. on one kind of JVM the bit depth will be same regardless of references.

> Default value is NULL

eg: Student S; // S is null initialized

Consider

S.updateProfile(); // If you use a dot operator on a null reference, will lead to the null pointer exception.

Statements: → is Basically a command executed by Java.

→ The statement is ended with a semicolon ;

Command can be used → declare a variable

→ change a variable value

→ Invoke a method

> statement basically changes program state

> A statement is made of one or more expressions → which is evaluated to a single variable

> And the evaluation happens at the runtime

An expression can be dealt using literals, variables, operators (+, -, *, /) and method calls

eg: int count = x * getCount();

getCount returns a value, then multiplied by x and stored in count variable

> Compound expressions are constructed from the smaller expressions.

> every expression whether it is an simple expression or an compound expression gets evaluated to a single variable

Statements are of three types

① declaration statements eg: int count = 25;

② expression statements

```
eg: count = 25; //assignment statement
    getCount(); // method invocation statement
    count++;    // increment statement.
```

③ Control flow statements :- The regulate the order in which the Program gets executed

```
eg: if(count<100)
    {
    ----
    }
```

Note :- expression statements & control flow statements cannot be defined within the class level as declaration statements, which are defined at class level.

> we get an error if we try to define or declare ② & ③ statements directly within the class.

Array :- A software gets a lot of its power through the datastructure it uses

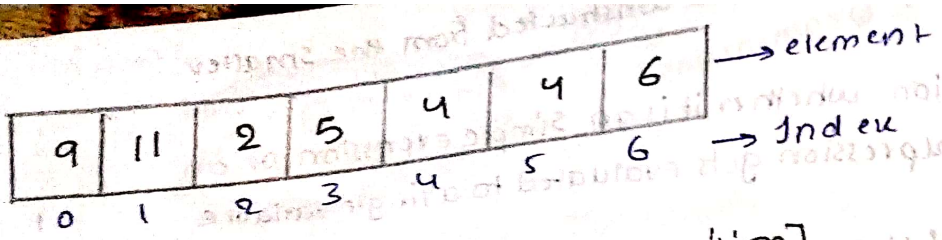
And an data structure is an organized collection of related data.

** Array is basically a container object that holds fixed # values of single type".

- no. of elements is fixed at creating.
- In Java basically array is an object and stored in heap memory
- single data type

> So arrays can store primitives and also object references

eg:-



To access we use <arrayname>[index_position]

ex: a[3] ⇒ 5.

Creating an array :- There are three ways to create an array

① eg:- int[] array = new int[7];

int[] <Arrayname> = new int[size];

* default value is zero (0)

or. we can initialize individually

Array[0] = 9; array[2] = 2; array[4] = 4; array[6] = 6;

array[1] = 11; array[3] = 5; array[5] = 4;

note: If there is only object, in the array declaration

Since array is an object, it gets default value as any other object reference gets which is NULL.

② eg:

int[] myArray = new int[]{9, 11, 2, 5, 4, 4, 6}

int[] <Arrayname> = new int[]{-, -, -, ----}

* element initialization is done in the declaration itself

③ eg.

int[] myArray = {9, 11, 2, 5, 4, 4, 6}

int[] <Arrayname> = {-, -, -, ---}

Note :- It is perfectly legal to even have subscripts after arrayname also

```
int <ArrayName>[];
```

→ length feild in array object :- we can use length in the array object to find length of arrays

```
<Arrayname>.length
```

eg: myArray.length → 7

- accessing array outside boundary gives runtime error

eg: int item = myArray[7]; // runtime error

⇒ Array's storing object references :-

> similar to the primitive arrays, we can also create an array of object references

eg:

```
Student[] students = new Student[2];
```

↓

student array which can store 2 object references

*default object reference value → NULL

> so we can assign each object reference to new student object

```
Students[0] = new Student();
Students[1] = new Student();
```

> we can also assign using dot operator

```
Students[0].name = "john";
Students[1].name = "raj";
```

→ how Arrays are stored in memory:-

Jvm dosent specify anything how objects are stored in memory

> most Jvm implementations use linear layout to store arrays → linear layout → fast random access at @ O(1) speed

> They are stored in contigious memory locations

i.e, accessing array[1000] would take same time accessing array[5] elements only.

> But searching would take linear time ~ O(n)

2-dimensional arrays :- elements are arranged in a grid or matrix form.

> In 2-d array elements are arranged in rows & coloumns.

↓ coloumns \ → rows	0	1
0	9	11
1	2	5
2	4	4
3	6	13

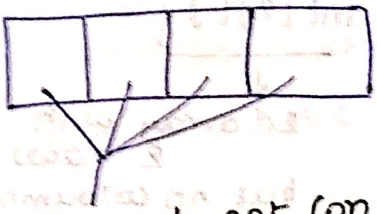
array[0][1] = 11;

Creating a 2-d array :- There are three ways

① eg: ✗ int [][] Array = new Array int [4][2];

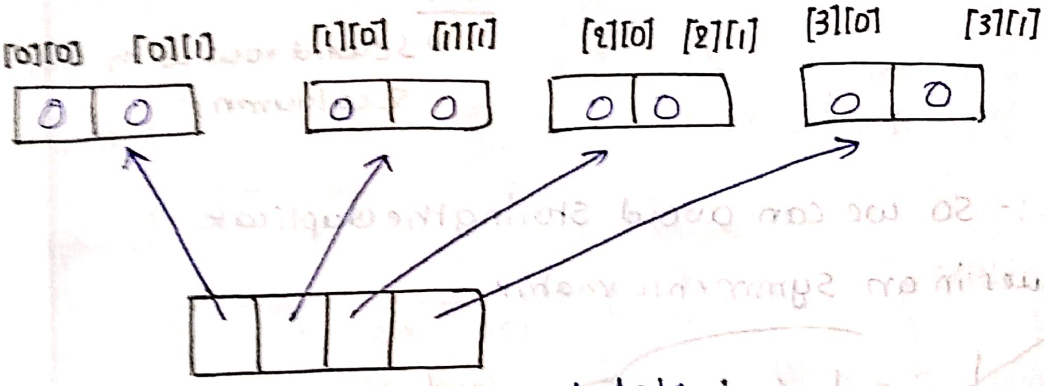
✓ *** int[][] <Arrayname> = new int[row][coloumn];

> internally, first JVM creates a one dimensional array with 4 elements



These do not contain any raw data

⇒ But each place has an object references to the other array of list elements



> and those contain the actual data.

* default values are zero.

> 2d arrays are implemented using 1-d array

> so we call

type[] <Arrayname> → Array of <Arrayname>

int[] arrayKav → Array of arrayKav

int[][] myArray → array of array of int.

> we can also initialize them individually

myArray[0][0] = 9;

② eg: int[][] myArray = new int[][] {
{9,11}, //row
{2,5}, //row
{4,4},
{6,13}
};

⇒ Array with 'irregular rows' :-

eg:

```
int[][] myArray = new int[2][];
```

↓ 2d array with 2 rows but no columns specified

so later we can write

```
myArray[0] = new int[5];
```

↓ first row with 5 columns

```
myArray[1] = new int[2];
```

↓ second row with 2 columns

Area of use :- so we can avoid storing the duplicate values in an symmetric matrix

int[1]	7	-2	6	2
int[2]	-2	3	20	11
int[3]	6	20	9	5
int[4]	2	11	5	-4

→ duplicates

so storage can be saved

> we can also use the 'length' feild for 2d arrays but it only returns the row value

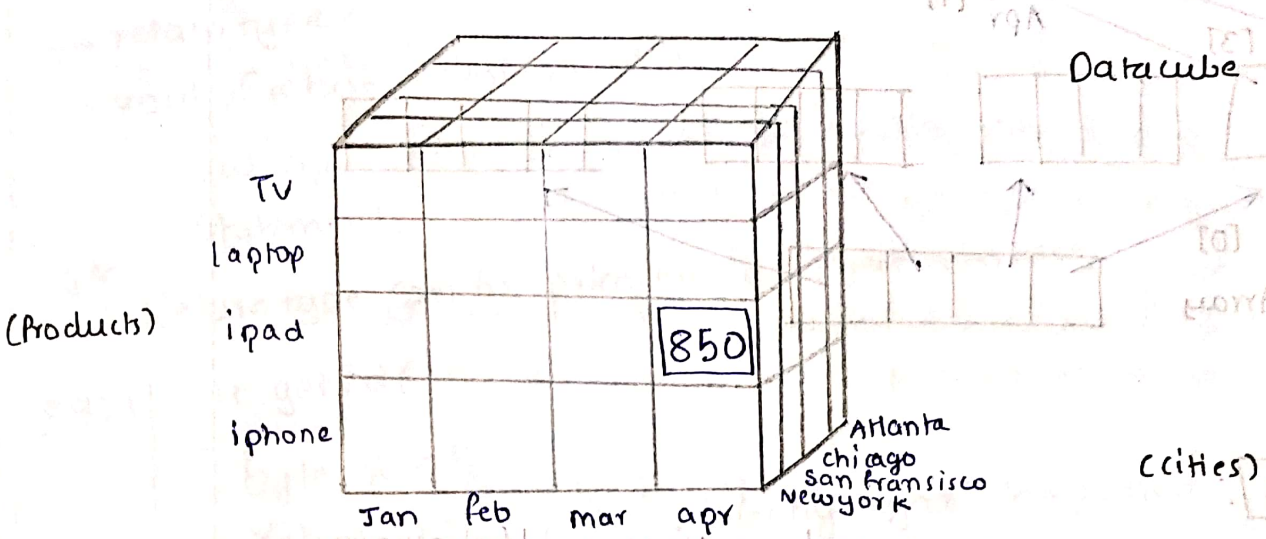
∴ eg: myArray.length → 4 // row

myArray[0].length → 2 // column

3D Arrays :- Other than 2d, higher dimensions are rare in practice

Usually data warehousing software stores data in such data cubes

> Consider an electronic store sales data which is represented in a data cube



> data cube has 3 dimensions cities, months, products

> In here 850 means 850 ipds were selled in april in the newyorkcity

eg:. myArray [0][3][1] = 850
↓
Plane

→ The two ways to create & initialize

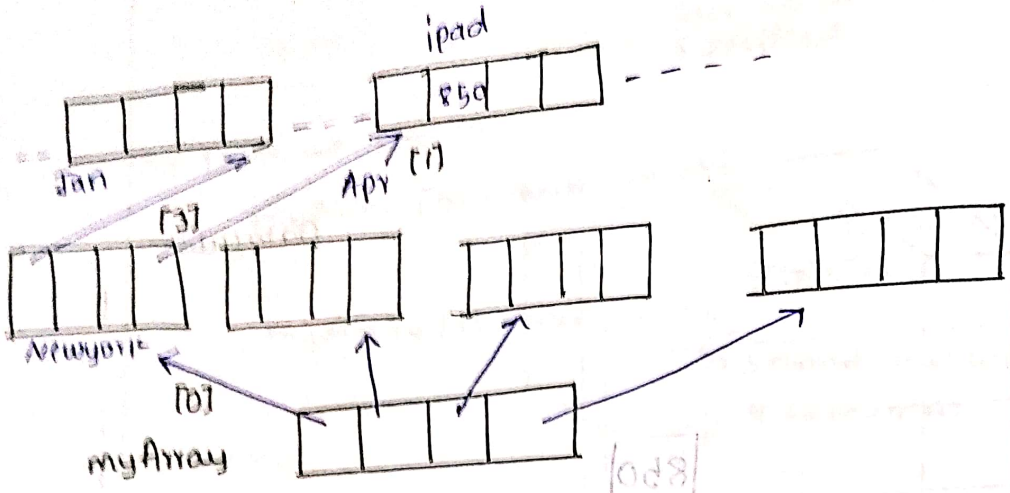
① int[][][] myArray = new int[4][4][4];

② myArray [0][3][1] = 850;

⇒ 3-d array illustration :

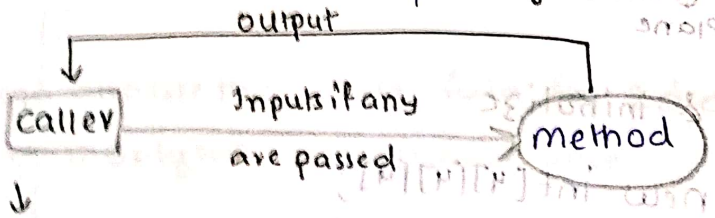
To be continueed.....

here, the second dimension represent the ...
→ The 3rd dimension contains products and items sold.



METHODS

- As previously discussed objects have state and behaviour
  Methods define behaviour.
  " methods are self-contained logics that can be used any number of times "
- methods can receive input & generate some output.



Syntax :-

```
<returntype> <method name>(parameters / formal arg)
{
  -----
  return <somevalue>;
}
```

optional

↓ It has some type

↓ It maybe primitive or object reference

↓ This can be Primitive or object reference.

Note: In Java the methodname and parameters are together called as the method signature. but not return type

caller syntax:

type <var> = <methodname> (arg1, arg2 ...)
→ inputs to method
↓ actual parameters

→ return type
> void (returns nothing)
but optionally we can write return; as last statement.

> Return type can be primitive or object reference.

eg:1
```
int getId()
{ byte x = 5;
  return x;   // It is perfectly legal: Compatible
}
```

eg:2
```
byte getId()
{ int x = 5;
  return x;   // int is down casted to byte before returning
}
```

Advantages of methods:-

> Avoid duplicate code
> divide & conquer
  * software reuse
  * clean & readable code

Note: A method can be called by
<class_name> Object = new ~~method~~ <class_name>();
(then) Object.method();
If method static then no need to create object.

Types of methods → Instance methods
→ static methods

Instance methods :- They are methods of instances of a class i.e, objects of a class (object-level methods)

Invocation :- we use an dot operator

```
Objectref.methodName();
```

> After performing certain tasks it affects the object state
> Instance variable affects the object state by directly manipulating instance variables of that class
> object state can also be affected by one instance method invoking other instance method of same class.

Static methods :- They have the keyword static in the method declaration.

```
static void go {
---
---
}
```

> Static methods are class level methods
> They dont have access to instance variables / methods
> They can only access static variables / methods
> Invocation :-

```
classname.methodName();
```

> main method is an example of static method.
> A static method can access other static methods in the same class.

> unlike static methods, for instance methods we need to create an object first in order to access it

How data is passed to methods in java?

> Passing the data to a function can vary from one programming language to other

```
void updateId (int newId)
{
    newId = 1001;
}

int id = 1000;
updateid (id);
```

// program assigning 1001 to newId

// then answer = 0

so output = 1000

> In Java id would be 1000, but in other languages it would be 1001

> For same if we pass object reference

```
void updateId (Student s1)
{
    s1.id = 1001;
}

Student s = new Student();
s.id = 1000;
updateId(s);
```

so output = 1001

> when primitive was passed calling environment was not affected, but whereas in Object reference calling environment is affected.

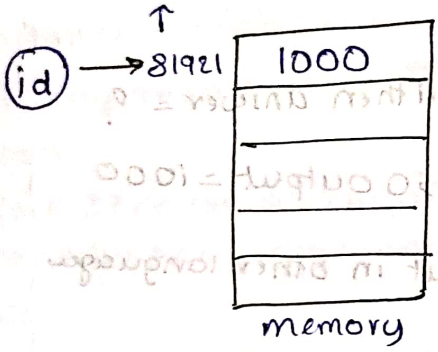
Passing data:- → Pass by value
→ Pass by reference

Pass by value :-

> Passing the primitives

Consider int id = 1000

id → contains → logical name → variable name
→ memory address
And the value

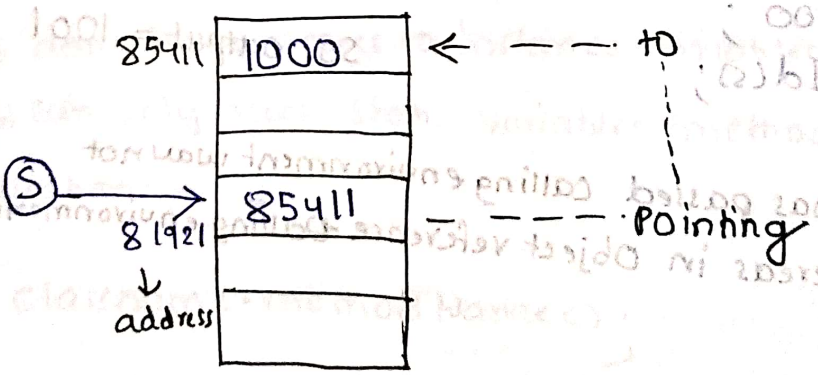


> At the runtime we have only memory address and the variable data.

> Object reference in memory

Student s = new Student();

Object — Object reference



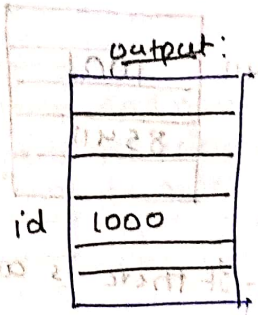
As the name implies Pass by value means, value of an argument is passed to the parameter.

i.e, if primitive argument → value is primitive

and if object reference argument → value is memory address

eg1:-

```
void updateId(int newId)
{  newId = 10001;        //step 2
                         //step 3
}
int id = 1000;           // step 1
updateId(id);            //step 4
```



Step 1 : id 1000

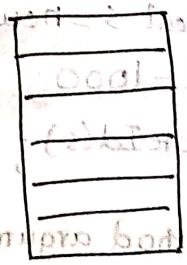
Step 2 : newid 1001

Step 3 :- passing 1000 to function

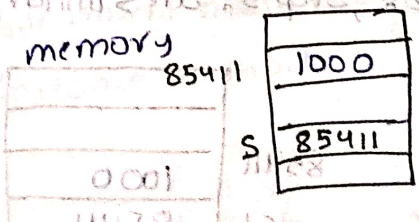
Step 4 :- the newId will be discarded and finally 1000 is stored

eg2:-

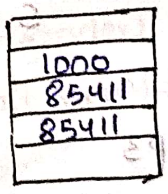
```
void updateId (Student S1)
{      // Step 2
   S1.id=1001;
        //step3
}
Student s = new Student();   //step1
s.id=1000;
updateId(s);                 //step4
```



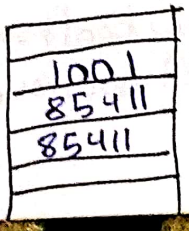
Step 1: S.id is loaded into memory



Step 2:- function invoked and then



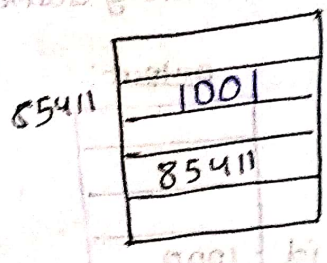
Step 3:-



*** M.Imp

when changed, it even affects the change in the value of 'S' also

Step 4:



after it comes out local variable s1 is discarded and s value when called gives 1001

Conclusion:- If there is a change in the value, inside the method. Then that modification is also reflected in the calling environment also. And this do not happen with the primitive arguments.

Consider the following example:-

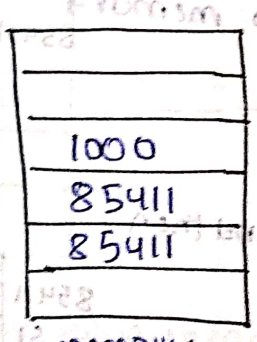
```
void updateId (Student s1)
                // step 2
{
    s1 = new Student ();  //** reinitialization **//
    s1.id = 1001;
        //step 3
}

Student s = new Student ();
s.id = 1000;  // step 1
updateId(s);  // step 4
```

** here method argument and parameter are completely independent. So change in the value inside method, will not be reflected in the calling environment **

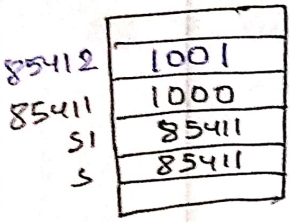
Step 1:-, Step 2:-, Step 3:- are similar

in Step 3:



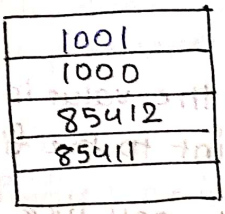
memory

Now at step3



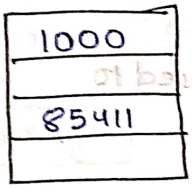
→ new object is created

then



and now s1 referencing the new object variable value '1001'.

Step 4:-



And in here, s1 is discarded from the memory.

Conclusion:- In here s is unaffected because, we added a reinitialization statement

Note:- "Java is always uses PASS BY VALUE" (It may be Primitive or object reference and as there is no concept called pointers in Java).

method overloading:- Sometimes we need to maintain the various instances of methods of (same) identical method names, but would take different input data, same return type.

> multiple methods with
  - same name
  - Different parameters
    - # The change must be in no. of parameters or their return types. or both
  - changing only return type doesn't matter. (does not work)

> It applies to both static & instance methods

Valid examples:

eg:-1
```
boolean updateProfile(int newid) { }
boolean updateProfile(int newid, char gender) { }
boolean updateProfile(char gender, int newId) { }
```

eg:-2
```
void overLoad(int i) { }
void overload(byte b) { }
```

Consider eg: 2

If we pass

> Overload(23); // the value is defaultly passed as int to the first method

But if we explicitly pass like

> byte b = 23;
Overload (b); // then the value is passed to Second method

> Assume there is another method

void overload (short s) { } and if value is passed

byte b = 23;
Overload (b);

** even though there is a method with byte parameter the compiler chooses most specific higher order parameter and passes to that method ** **

when byte is not there then it chooses method with next high order method (based on parameter)

Invalid examples:-

eg ① void updateProfile (int newId) { }
void updateprofile (int newId) { } X // same
int updateprofile (int newId) { } X // Just Change in return type

And compiler reports that we have a duplicate method.

eg:② void updateProfile(int newId) { }
void updateProfile(int id) { } ✗
As changing just parameter name
dosent work.

eg:③ Static void updateProfile(int id) { }
void updateProfile(int newId) { } ✗
even adding 'Static' to method signature wont work.

Method varagu :-

> Before Java 5, a method could only be invoked with a fixed number of arguments.
i.e if you have a method with 3 parameters then we need exactly 3 arguments to invoke it

> Java 5, introduced a special kind of arguments called Varags ~ variable-length arguments.
i.e, it can take variable number of arguments.

> Last parameter* can take variable number of arguments it can be 0, 1 or more than 1 arguments
- It could be the only parameter of an method.
- or if there are many parameters, then the varag parameter must be the last one.

Syntax & invocation:-

> Syntax involves three dots following parameter type.
eg: updateProfile(int id, char gender, int ... items)

> And to the above method, the argument can be an array of any size
updateProfile(101, male, new int[]{1,2,3});

> Or we can also write as comma seperated arguments.
updateProfile(101, male, 1,2,3);

> If passed with comma's then compiler automatically converts them into an array.

> It can also be omitted

updateProfile(101, male); //so empty array is passed to varag;

> varags just give the illusion that the method is infinitely overloaded.

> If it is not last parameter it is illegal

updateProfile(int...items, int id, char gender);
// Illegal.

Note:- A method cannot have more than one varag Parameter.

Why use varags?

> we can actually pass arrays altogether
> But varags provide simpler & flexible invocation.
> But we can use an array if there are many number of elements

Printf() in Java:- method printf() which was introduced to print formatted text, which is introduced in Java 5.

Syntax:

Printf(String format, Object...args)
→ varags Parameter

eg:- System.out.println(" ");

System.out.printf("DOB:%d/%d/%d", 1, 1, 1978);
→ placeholders.

// it prints 1/1/1978.

main method with varags parameter

> Public static void main (String [] args)
{
}

> we can replace the arguments of strings with varags

Public static void main (String ... args)
{
} //perfectly legal

varags in overloaded methods:

> Invalid example:
updateProfile (int id, char gender, int... items)
so we cannot have an another method overloaded like this
updateProfile (int id, char gender, int[] items);
//illegal & Compiler shows error.

> If we have many methods written, then the method with the varags parameter will be matched last.

Constructors: It constructs an object an initializes its state.

> It has a code which runs when you create an object.
> It is mostly used to initialize the object state.
i.e, to initialize the instance variables of that object.

eg: Student s = new Student();

- Student s: Allocates space for reference variable's.
- =: assigns address of object to reference variable
- new Student(): Allocates space for new 'student' object
- ----> Invocation of constructor

Syntax:-

> A constructor must have the same name that of the class

```
classname (type parameter1, type parameter 2, ---)
{
  ---  ---
}
```

> A constructor donot have a return type.

> Constructor can also have varags parameter.

eg:-

```
class Student
{
  int id;
  Student (int newId)        // constructor
  {                          // which initializes
    id = newId;              // id value with
  }                          // newid value.
}
```

Default Constructor:- If any constructor is not provided, the compiler inserts a default constructor with no parameters.

> default constructor is also referred to as no-args constructor.

> This will be the case if class definition donot include a constructor.

eg:

```
class Student
{
  int id;
}
```

As no Constructor initialized

```
class student
{
  int id;                    // The compiler automatically
  Student() { }              initializes a default constructor
}                            into java bytecode
```

So we can create an object inovoking no-args constructor

```
Student s = new Student();
```

eg 2:

```
class Student
{ int id;
  Student(int newId)
  { id = newId;
  }
}
Student s = new Student(1001);
Student s = new Student(); // Illegal
```

as already a constructor is defined.

> If we want to do both then we must add

```
Student()
{ ---
}
```

to the code

Parameterized constructor :- The constructor containing an 1 or more parameters.

Constructor overloading:- Same as the methods:

Constructors can also be overloaded. Having more than one constructor in a class implies that they are Overloaded.

> They have the same overloading rules as methods.
  - Parameter list must be different

> we can create objects using any of the constructors.

Why Constructor Overloading is required?

> To create the objects with the different capabilities

eg:- we can assume the class FileOutputStream

- FileOutputStream(String name, boolean append) ✓
- FileOutputStream(String name) ✓
- FileOutputStream(File file) ✓
- FileOutputStream(File file, boolean append) ✓
- FileOutputStream(FileDescriptor fdObj)

> first two are overloaded constructors with same name with different parameters

eg 2:-

```
class User {
int id;
String name;
int Salary;
    User (int UserId, String UserName)
    { id = userId;
      name = UserName;
    }
User (int userId, String userName, int userSalary)
{  id = userId;
   name = UserName;
```

```
    salary = userSalary;
  }
}
```

This keyword :- we can also use the this() keyword to access the repetetive/same local parameters in one constructor from other constructor

```
class User {
int id;
String name;
int Salary;
User(int UserId, String userName)
 { id = userId;
   name = userName;
 }
 user (int userId, String userName, int userSalary)
 {
     this(userId, userName);
     salary = userSalary;
 }
}
```

** Note :- This keyword must be the first statement in the Constructor otherwise the compiler gives an error.

We can only have one this statement per constructor.

We cannot use instance variables as arguments to this statement. A constructor cannot invoke itself using this invocation statement.

Recursive Constructor invocation :

```
class User
{ int id;
  String name;
  int salary;
  User(int userId, String userName)
  { this(userId, userName); //compile error
  }
  User(int userId, String userName, int userSalary)
  { this(userId, userName, userSalary);
                    // compile error.
  }
}
```

This reference :-

> To access a variables or methods of an object

```
objectRef.someVariable
(or)
objectRef.someMethod();
```

> If an object wants to access it's own members, then we use a this reference.

```
this.variable          // Basically is an object reference
(or)
this.method()
```

> It can be also called as reference to current object.

> But, An object can also access it members directly.

why this reference needed ?

" when we want to access the instance variables hidden (or shadowed) by the local variables in the method or Constructor ".

eg:-

```
class Student
{
  int id;                          // Instance variables
  String name;

  void updateProfile(int newId, String name)
  {
    id = newId;                    // directly accessed
                                   // Instance & local variables
    this.name = name;              // Share the same name
  }
}
```

** Within the method updateProfile, the method parameter 'name' is hiding (or shadowing) instance variable 'name'

** And any reference to the name within the updateProfile method is actually a reference to the method parameter and not instance variable

> To access the instance variable within the method we use this reference.

Note:- It can be also used in the constructors.

** we cannot use this reference in a static methods.

Operator → It performs operations on the operands and produces a result

eg: x+5 (operator: +, operands: x, 5)

Types of operators

- Assignment
- Arithmetic
- Comparision
- Logical
- Bitwise
- Bit shift
- instance of

> An operater can handle unary, binary & Ternary operands

unary :- ++x Prefix
x++ postfix

Binary :- for the two operands the operator mostly appears between two operands
eg: x+3 Infix

Ternary :- Conditional operator uses Ternary operands
eg: (x>3) ? x : 0

Arithmetic operators :-

> Addition (+) ~ int i = 5+2;
+ is also used as unary plus int i = +5
and also for string concatenation

> Subtraction (-) ~ also used as unary int i = -5

> multiplication (*)

> division (/)

> modulus (%) eg: int i = 5 % 2 // i = 1

> pre on post increment (++)

```
int x = 5;
y = x++
```

first y is assigned to x then later incremented

y=5, x=6

If y = ++x

first x is incremented then assigned to y

∴ y = 6, x = 6

> Post and predecrement (--) also work in the same way

> Arithmetic operators are only applicable to Primitive Numeric datatypes except boolean

> Compound Arithmetic assignment operator

eg: x = x+5

then we can write

x += 5;

using '=+' will be simply treated as an assignment operation

Similarly we can also write -=, *=, /=, %=

Arithmetic operation Rules:-

Consider 5+9-3+2*5 (which is confusing)

To evaluate this expression java have a predefined set of rules called operator precedence

Rule 1:- multiplicative/divisible operators (*, /, %) have higher precedence over additive operators (+, -)

A has even higher prior than *,

So our expression would be

5+9 − 3+(2*5)

Rule 2 :- operators in same group are evaluated from left to right (operator precedence)

So our expression would be

((5+9)−3) + (2*5)

Rule 3 :- we can use the parenthesis to change the order of evaluation

((5+9) − (3+2) *5)

Rule 4 (operand promotion) :- operands smaller than int are promoted to int

127 + 1 → 127 + 1 → 128
(byte) (byte) (int) (int) (int)

they are converted into int before addition option is applied

97 98

eg: 'a' + 'b' = 195
(char) (char) (int)

Rule 5 :- If both operands are int, long, float or double, then operations carried in that type and evaluated to a value of that type

eg: 5 + 6 = 11

1/2 = 0 not 0.5 → truncated to 0

Rule 6 :- (for mixed type operations) :- If the operands belong to the different types, then smaller type is promoted to the larger type

Order of promotion

int → long → float → double

eg: $1/2 \rightarrow 0$

$1/2.0 \rightarrow 0.5$ (or) $\frac{1.0}{2} = 0.5$ (or) $\frac{1.0}{2.0} = 0.5$

char-float evaluation:-

char + float → int + float → float + float → float

eg:-

$9/5 * 20.1 \rightarrow (9/5) * 20.1 \rightarrow 1 * 20.1 \rightarrow 1.0 * 20.1$

$\rightarrow 20.1$

Comparision operators :- Compares one operand with another

eg: $x > y$

> Basically always a condition is being tested
> and output obviously generates a boolean value
> These are mostly used in control flow statements

```
eg: If (x>y)
    {
    ---
    ---
    }
```

> Comparision operators are also referred to as relational operators

operators	Description	eg: (let x=5)	Primitive support
<	less than	(x<7) → True	No Boolean
<=	less than or equal to	(x <= 7) → true	No boolean
>	greater than	(x > 7) → false	No boolean
>=	greater than or equal to	(x >= 7) → false	No boolean
==	equal to	(x == 7) → false	All
!, =	Not equal to	(x !, = 7) → true	All

Object reference comparision:-

> Student s1 = new Student();
  Student s2 = new Student();
  when we compare s1 and s2
  > (s1 == s) → false (as it contains different memory addresses)
  > (s1 != s2) → True

> So using '==' and '!=' we can validate the object references

```
boolean foo(student s)
{   if(s==NULL)
    {
      return false;
    }
    // do something
    return true;
}
```

Logical operators:- used to test the multiple conditions

eg: (x<7) and (y<3) ?

using comparision operators

```
if (x<7)
{
  if(y<3)
  {
  }
  ---
  ---
}
}
```

> But using logical operators this can be done in a more elegant way.

```
if (x<7 && y<3)
{
  ---
}
```

operator	Name	(let x=5, y=6)
&&	AND	((x<7) && (y<3)) → false
||	OR	((x<7) || (y<3)) → True
!	NOT	!(x<7) → false. (actually True) (Returns opposite of result)

> They even generate boolean value
> They are also used in the control flow statements
> Also called as conditional operators

Note :- && operator prevents the NULL pointer exception

Operator precedence :-
> ! operator has the highest precedence
> && has next highest precedence
> || has the least precedence.

(! ← && ← ||

> we can use parenthesis to change the evaluation order.
> && , || are also referred to as short circuit operators.

Bitwise operators :- operate on the operands

> Operands can be integer primitives (operand promotion rule applies)

> They can also be boolean operand, but very rarely used

> Bitwise operators work at the bit level. i.e, they would depend on the binary representation of bits & then operate on them

Applications of bitwise operators:-

① used mostly in the embeeded systems Applications i.e, mostly in the resource constraint environments.

② They are also used in the hash tables, Java hash map's hash function

③ They are used in encryption and compression

Bitwise operators

& → Bitwise AND

| → Bitwise OR

^ → Bitwise XOR (exclusive OR)

~ → Bitwise NOT

Bitwise AND (&):-

> Returns 1 if both of the inputs are 1, otherwise it returns 0.

eg: let x=1, y=3

∴ (x&1) → 1

```
00000000 00000000 00000000 00000001
00000000 00000000 00000000 00000001
-----------------------------------
00000000 00000000 00000000 00000001
```

∴ returns 1

Note: These operators are booleans instead of integers

∴ they treat true = 1, false = 0

Bitwise OR (|) :-

> returns 1 if either of input bits is 1, otherwise it returns 0

let x = 1, y = 3

(x | y) → 3

```
00000000  00000000  00000000  00000001
00000000  00000000  00000000  00000011
-----------------------------------------
00000000  00000000  00000000  00000011
```

∴ returns = 3

Bitwise XOR (^) :-

> Returns 1 only if one of the input bits is 1, but not both.

eg. let x = 1, y = 3

(x ^ y) → 2

```
00000000  00000000  00000000  00000001
00000000  00000000  00000000  00000011
-----------------------------------------
00000000  00000000  00000000  00000010
```

∴ returns 2.

Bitwise NOT (~) :-

> It inverts Bits (0 → 1 and 1 → 0)

let x = 1 ∴ ~x = -2

~(00000000 00000000 00000000 00000001)

11111111 11111111 11111111 11111110

∴ returns −2

Note :- & and | are called NON short circuit operators

They always check both operands

Compound bitwise Assignment

> x &= y instead of x = x & y

eg: boolean b = true

b &= false // Assigns false

- Similarly x |= y and x ^= y

Bitshift operators :- Likewise Bitwise operators they even work on the individual bit of operands

> They are simply shift bits

> operands are integer primitives

They can shift the bits in either left or right side

<< → left-shift

>>> → unsigned right shift

>> → signed right shift

Left-shift operator (<<) :- left shifts left operand by number of bits specified on right

let us consider ex: 6

00000000 00000000 00000000 00000110 → 6

If 6 << 1 (1 bit left)

then

00000000 00000000 00000000 00001100 → 12

So it is 12

> As bits get shifted, it inserts zero in the lower order bits (empty slots)

> left shift operator is same as multiplication of powers of 2

$\therefore$ number * $2^{(shift\_number)}$

$\boxed{n * 2^x}$ //

unsigned - right shift operator (>>>) : right shifts the left operand by number of bits specified on right.

> It inserts 0's at higher order bits (empty slots)

ex: 12

00000000 00000000 00000000 00001100 → 12

12 >>> 1

thus

00000000 00000000 00000000 00000110 → 6

> Right shift operator is same as division by powers of 2

$\frac{number}{2^{(shift\_number)}}$ $\boxed{\frac{n}{2^x}}$ //

Signed-right shift operator (>>) : It is same as unsigned right shift operator, but it is padded with MSB instead of 0's.

> so sign is preserved

ex: −2,147,483,552

10000000 00000000 00000000 01100000

If number >> 4

11111000 00000000 00000000 00000110

$\therefore$ −2,147,483,552 >>4 → −134,217,722 → sign is preserved

Applications:-

① can be used in compiler optimizations, to replace multiplications & divisions with Bitshift operations which are much more faster.

② They are also used in the hash tables, Java hash map's hash function

③ They are heavily used in embeeded systems

④ They are used in games programming

⑤ They are used in systems with no-floating point support.

> like bitwise, bitshift operators can also be used in compound bitshift Assignment

```
x <<=y
x <<=y
x >>>=y
x >>=y
```

Control flow statements:- They control the flow of the Program execution (or) controls the order of statements execution

If statement:- execute if test passes

```
if (condition =true)
{
//execute block of code;
}
```

If else statement:- along with if and only one condition

```
if (condition = true)
{ //code;
}
else //if false
{ //code 2;
```

> we use if-else-if for long conditional tests

```
if (condition)
{
  // code
}
else if(condition 1)
 {
   // code
 }
 else if (condition 2)
  {
    // code
  }
  else                    // else block is optional
   { // code              // else block should be last
   }
```

> Nested-if

```
if( c )
{
 if(c)
 {
 }
}
```

Switch statement :- It can be an alternative to the if statement

It is extensively used if we want to use & select options

ex:-

```
Switch (condition/option)
{
  case '1' :   //code break;
  case '2' :   //code break;
    .
    .
  case 'n' :   //code break;
  default :  //default option
}
```

└ keyword

> If there is no break the case statements execute one by one until an break is encountered.
> default statement need not to be the last block.
> Switch expression may be 7, x, xty --
> Condition may be byte/short/char/int
> And the condition may be an object references to the byte/short/char/int which are called wrapper classes (or) boxed primitives (Instance of these classes)
> The final value of expression taken should be evaluated to an integer.

→ A Switch expression can also be a String (since Java 7)
→ A switch expression can also be an enum.
→ If switch expression evaluates to null at runtime, then we get an null pointer exception.

⇒ Case label restrictions:-
> The value of the case label must be within the range of the data type of switch expression.
> The value of the case label must be known at compile time itself
> case label value must be unique
> It cannot be null.

Where we
wrapped
the Compile
is
and Compa
the primi

Tip: Alw
the break

⇒ when it is not feasible to use a switch statement.

> Switch statement is not feasible if we have more than one condition to test.

> when condition is to test other than equality (>=)

> when condition/switch expression is other than integer, String or enum.

> A switch is not usable if atleast one case lable is not available.

⇒ when switch is preferred?

> If you want to improve readability

> If you deliberately state that only one variable is involved

> Speed // faster than if

> If there are 'n' conditions to test

time taken by if is O(n)

but by switch is O(1)

> we can use an profiler to identify which part of program is slowing down your program and for eg:

If #conditions >100, if to switch conversion will bring a huge decrease in execution time.

> one of the profilers used are 'jprofiler'.

⇒ Ternary operator:-

result = (boolean-expression) ? trueexp : false_exp;

If boolean_expression is true then true_exp will be evaluated

Otherwise if it is false then false_exp will be evaluated.

> Ternary operator is a shorthand representation of if-else statements.

> It improves code readability

eg: Smart String Construction

greeting = "hello" + (user.isMale() ? "Mr." : "Ms.") + user.getName();

> Ternary statements are used in

eg:

```
System.out.println ("John is" + (s.isMale() ? "male" : "female"));
```

Note: Ternary statement cannot be an expression statement

→ for loop statement :- It is an iteration statement.

Rules:

> Initialization part is optional.

eg:

```
int Array = {0,1,2,3,4,5,6,7,8,9};
int i=0;
for (; i < Array.length; i++)
{ System.out.println (Array[i]);
}
```

> eg:

```
for (int i=0;;)          //VALID
for (int i=0, j=1;;)     //VALID
for (int i=0, int j=1;;) // INVALID
for (int i=1, double d=1.0;;) //INVALID
```

> we can even reinitialize the values.

```
int i=0; j=1;
for (i=1, j=2;;)          //VALID
for (i=1, double d=10.0;;) //INVALID
for (i++;;) //VALID
for (System.out.println(i);;) //VALID
for (System.out.println(i), i++;;) //VALID
```

> condition expression must be evaluated to a boolean value.
> If omitted, a true is assumed then it results in an infinite loop.

eg:
```
for(int i =0; ; int itt)
{
  System.out.println(i);
}
```

To avoid this we can use a break statement in above example.

> we can have one or more expression statements.

eg:
for(int i=0 ; i<Array.length; system.out.println(Array[i]), itt);
> In here semicolon is valid and required as there is no explicit body.
> we can make above statement more simple
for(int i=0; i<Array.length; System.out.println(Array[itt]));

⇒ for-each statement :- it is introduced in Java 5! It is an improvised version of for loop.

eg:
```
for (int i : Array)        → Array
{ System.out.println(i);
}
```

> And read as for-each element.
> It provides a cleaner syntax (index variable is hidden).

⇒ where traditional for loop is preferred:
> Transforming or replacing
> Parellel iteration
> Backword iteration
> filtering (removing selected items)

> But if we want to access array index then we can use traditional for loop.

⇒ while loop

```
while (condition)
{
. . .
. . .
}
```

⇒ dowhile : To run atleast runs once

```
dowhile
{ . . .
. . .
}do(condition);
```

⇒ Break Statement :-

> exists immediately enclosing switch (or) loop (for/while)

eg:1
```
for ( ) {
break;
}
```
exists for

2
```
for() {
  if(){
   break;
  }
}
```
exist for

3
```
for() {
 for() {
  break;
 }
}
```
exists inner for

4
```
for () {
 switch() {
  case: break;
```
exists switch

Invalid example :-

```
if()
{ break;
}
```

⇒ Labeled break statement :-

> label : block statement. (identifier)

> Block statements

> 0 or more statements in curly braces

```
if() { //block 1
} else { //block 2
}
```

> statements are executed sequentially

> can be nested

eg. Outermost : for(int i=0; i<10; i++)

```
{ for(int j=0; j<10; j++)
    { if(i==5 && j==5)
        { break outermost;
        }
    } num++;   //if breaks outermost for loop
}              //instead of inner one.
```

⇒ continue statement : used with only loops not switch.

> used to continue with next iteration of innermost loop.

> It is invalid for if/else, same as break statement

> we can use even labelled continue statement

eg: Same as for the break statement.

⇒ Scope of the variable :-

> every variable has a scope

> i.e, it is the part of the program where variable can be used.

⇒ Scope of class level variables :- Entire class

> A class level variable cannot be assigned to variables declared before it.

```
class Test
{ int i;
  int j = i;
  void see() {
    j=k;  //INVALID
  }
  j=k;  //INVALID
  int k;
}
```

> even j=k in method see() is valid because even before see method is invoked int k is initialized.

⇒ Scope of local variables :- (within methods / constructors / blocks)

Scope: from declaration to end of the block.

⇒ Shadowing class-level variables

```
{ int x=10;
  void see();
  {
    int x=0; //shadow class variable x
    x++;
    this.x++;
  }
}
```

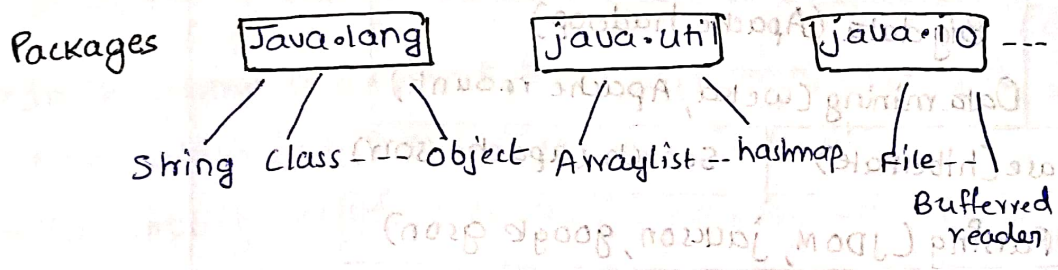
> methods invoked from current block will get a new scope

```
{ int x=10;
  volac();
}
```

```
volac()  //method
{ x++;  //INVALID
}
```

Java API (Application programming interface):-

- is a library of hundreds of well-tested classes
  eg: Java 8 has 4240 classes
- In the Java API classes are grouped into packages
  * (are directories of the file system) *



⇒ Why do we need packages?

- Instead of browsing all the 4240 classes, they are meaningfully organized into packages based on their functionality
- Name scoping: Consider java has two similar Packages
  java.util.Date ≠ java.sql.Date
  which are not same, without using these packages it is not possible.
- packages also enable security.

name scoping ⇕ naming conventions avoiding.

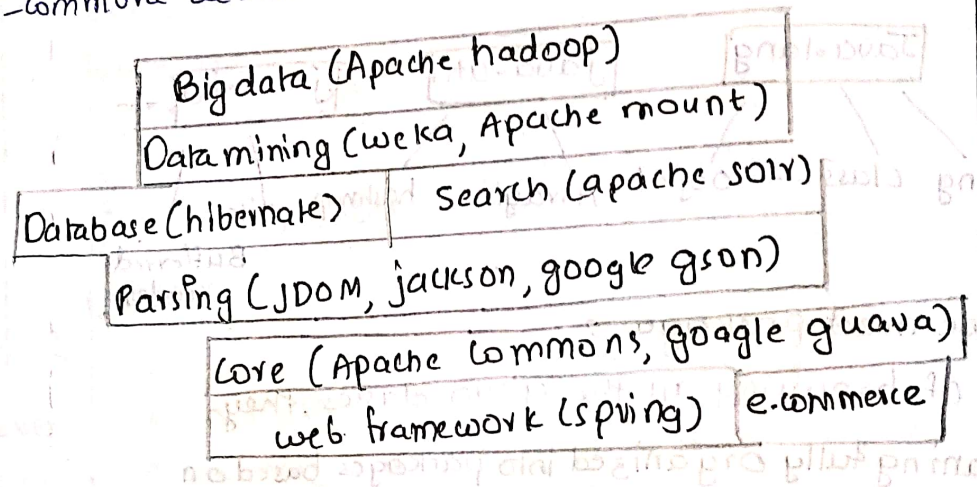
⇒ Important packages:-

- java.lang → Fundamental classes
- java.util → data structures.
- java.io → Input and output to a file, stream, buffer
- java.net → networking
- java.sql → Databases

In Java 8 there is a new package java.nio ↓ new input output

> we can explore different packages and classes within it from link
" https://docs.oracle.com/javase/8/docs/api ".
> It also shows several interfaces

⇒ 3rd party API's : To develop some advanced projects you need some additional packages offering more functionality than the traditional packages
> Consider an open source components that make an e-commerce website



> Api's can be used to write new code/functionality to improve performance

⇒ Accessing packages :-
> It is very common for the classes within a package to access other classes in the package, then classes can directly reference other packages in the same class using the class name
> To access a different package within same class there are two ways.

Note: We can customize 3rd Party API's

Note: For any the superclass Object class inh
> every class the properties object class au tly

(i) import
(ii) Fully - classified class name (rarely used)

eg: //import statement//

```
import java.util.ArrayList;
class FClass { void F() {
    ArrayList list = new ArrayList();
    ---
}
}
```

Note: This must be the first statement in program.
It cannot be within the class

> In the above example we are explicitly importing a single class (or single-Type import).

> To import multiple classes, there are two ~~ways~~
- Seperate explicit import
- Import on demand (or *import.

→ using '*' we can import all the classes present in the package
ex:- import java.util.*;

Note: Import '*' can break code

```
import java.util.*;
import java.sql.*;
---
Date date; //from util.    //Compile error.
```

⇒ Fully classified class name:-

> Alternative to import

java.util.ArrayList list = new java.util.ArrayList();

⇒ Invalid Imports:

```
import java.util.date;
import java.sql.date;
```

even this gives a compile error

Note: 1. An Import Statement do not make your class bigger
i.e it doesnot import bytecode
2. An Import statement doesnt affect the runtime performance
3. It saves us from typing fully qualified name
4. java.lang is imported by default.

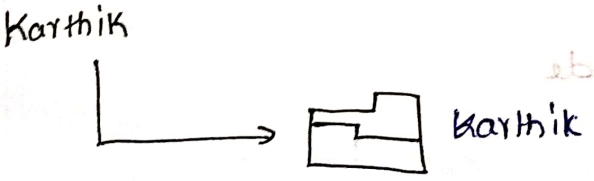
⇒ Creating packages:

> So to create a package (directory) named "karthik
It looks something like this

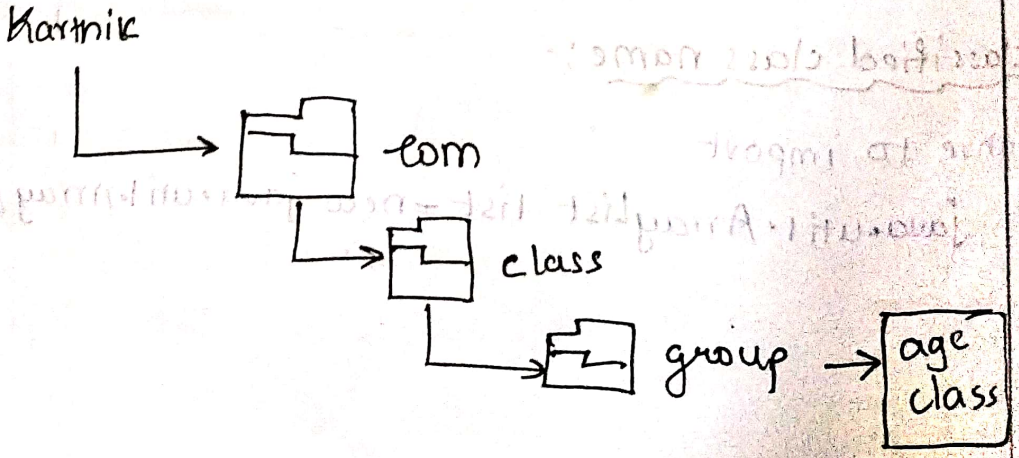
com.karthik

> So we need to create some directories corresponding to the package name (The directory name would be Package name)

> If the package is unique and single



> If it contains more components inside it, then we use sub packages



→ so we place the program / class in last directory group
→ And we need to have a package statement in the class

Syntax:

Package Packagename ;

↓
Keyword

ex: Package com.karthik ;

ex: Package com.class.group ;

Note: Package / Import statements must be first in the class
If not we get an compile time error.

```
Package com.class.group;
Import java.util.ArrayList;
class age
{ display(int age)
  { System.out.printf("%d", age);
  }
  static
  { display(10);
  }
}
```

★ PENDING
creating & using packages

Note:
1. we must always ensure the matching directory structure with the package exists
2. checking package statement.

Once class is compiled

Package name becomes Part of class name → So you need to use the fully classified class name

ex: java age ✗

java com.class.group.age ✓

⇒ **Strings**:- A string is basically an object of class

java.lang.String

> It stores a set of characters

Initialization:-

```
String s = new String();  // empty
String s = new String("hello");
```

> String literals are enclosed in double quotes
> A String literal is also a String object
> Internally a string literal is passed as a String object
> you can also pass an object reference of type String

→ Alternative way

```
char[] ch = {'h','e','l','l','o'};
String s = new String(ch);
```

→ normal way:- (Best way)

```
String s = "hello";
```

↙ It is class (String)

↓ Object reference

NOTE: String class uses Character Array internally to Store text

But for characters, java uses UTF-16 encoding

∴ A string in java is a sequence of unicode characters

> A String in java is also immutable
i.e, when a String object is created, its value can never be changed