INDEX

NAME: Misba ·k STD.: TSpiders SEC.: June-2024 ROLL NO.: SUB.: JBA - JFFCJD - M2

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
		Java		

01-07-24

JAVA

Program:

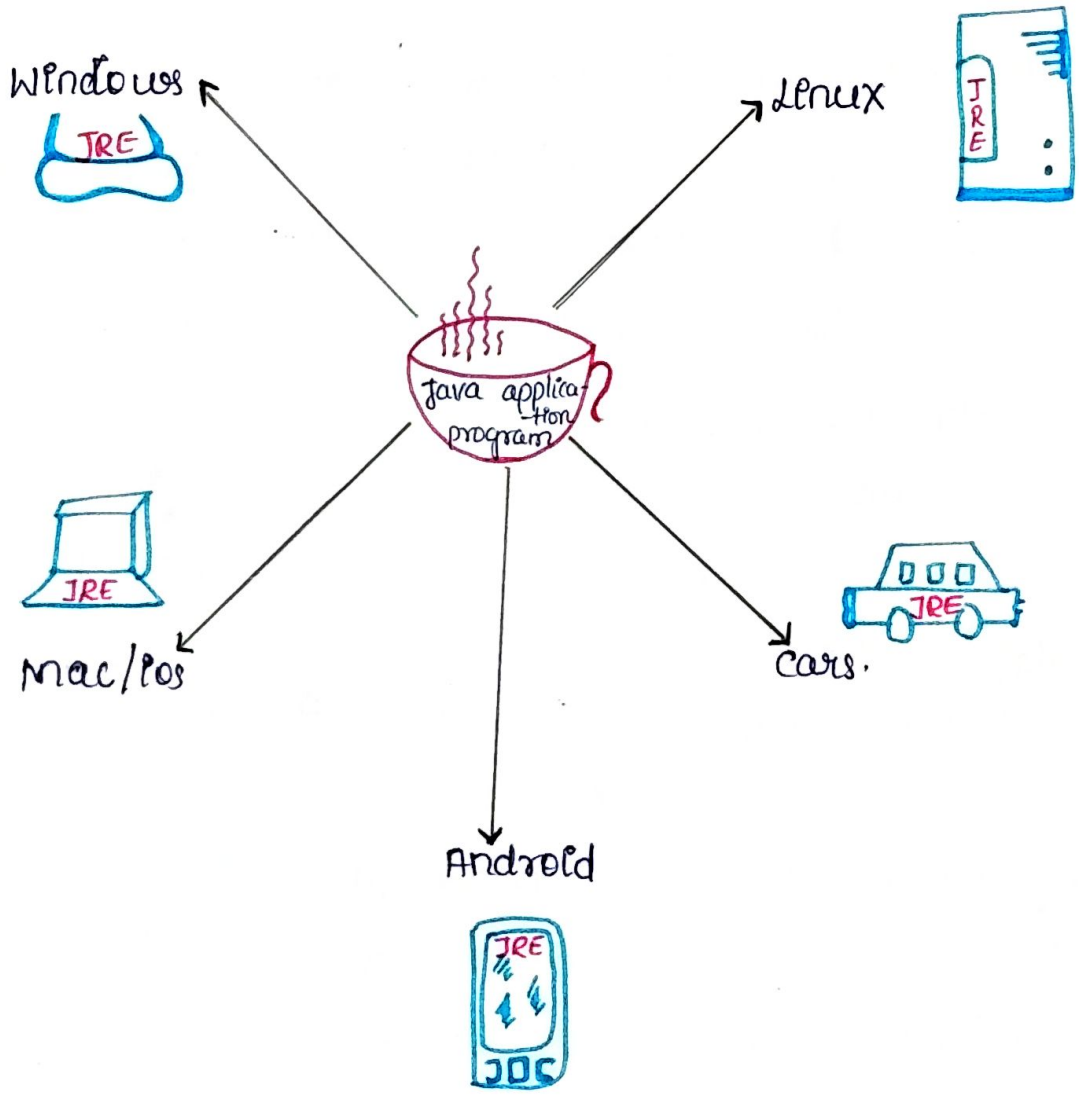
A program is a set of instructions to perform some specific task

Java:

(*) Java is a High-level, Object-oriented, and platform-independent programming language.

(*) If we develop a software using java, then it can run on different platforms because Java follows WORA-architecture.

[Write Once Run Anywhere -WORA]



Java Runtime Environment: (JRE)

(*) JRE is a special software required to execute java programs (or) java applications.

(*) Without JRE, we cannot execute java-applications.

(*) JRE is platform-dependent.

That is, JRE of windows is different from JRE of Linux, which is again different from JRE of Mac and others.

02/07/2024

Classes and Objects:

(*) Class is a blueprint (or) a logical entity which is used to create an object.

(*) A class can also be considered as the 'description of an object'. [components of class: fields (instance variables), methods or constructors]

(*) Object is a real world physical entity which is an instance of the class.

(*) Object cannot be created without the class. Hence, first step is to create the class and then the second step is to create the object.

(*) We can create one (or) multiple objects using a single class, which are technically called as identical or similar objects.

(*) Every object works independently. If one object is destroyed (or) damaged, then it neither affects the other objects nor the class. [i.e.,] If an object is destroyed then it will not effect the class.

03/07/2024.

Syntax:

"Grammar" of a programming language is technically called as Syntax. Every programming language has its own syntax.

Keywords:

(*) Keywords are reserved words. Keywords are the pre-defined words. Every keyword has a pre-defined meaning in the programming languages.

(*) There are approximately 53-keywords in Java.

(*) Java is a Case-sensitive programming language. All the keywords are in Lowercase in Java.

(*) Keywords are building blocks of Java's syntax.

Keywords / Reserved words:

package	import	static		
public	private	protected		
class	new	this	super	
final	abstract	interface		
int	long	double	float	boolean
if	else	while	for	
true	false	null.		

..... etcetra .....

Java members: (7)

- class
- variable
- Method
- constructor
- Interface
- enum
- package.

Eg:

```
class   dinga
{

}
```

dinga.java.

(Any program is saved with classname'.java' extension)

Here, the number of space between 'class'-keyword & 'class-name' doesn't matter how lengthy it is. But one space is always mandatory between them.

Identifier:

Identifier is the one which is used to identify a java member like a class, variables, method, etc.....

Rules of Identifier:

(*) Identifier cannot have 'the space'.

Eg:
```
class alia Bhatt
{                  (x) error appears.
}
```

(*) Identifier cannot have any special characters, except (-) underscore as well as ($) dollar.

Eg:
```
class marker-pen
{          (✓)
}

class cancel$button
{          (✓)
}

class product.life
{          (x)
}

class account-info
{          (x)
}
```

```
class _student
{          (✓)
}

class -                    class --
{          (✓)   (or)      {          (✓)
}                          }

class $
{          (✓)
}

class $employee_
{          (✓)
}
```

(*) Identifier cannot be a java keyword.

Eg:

```
class new          |  class this
{                  |  {
}   (X) ↓ keyword  |  }   (X) ↓ keyword
```

```
class This
{   (✓)  { Because the 'This'-classname is in upper-
}            -case & keywords are always in
                lower-case }
```

```
class _Int
{   (✓)      { with (_), 'Int' can be used as classname,
}              but, it is better to avoid such names }
```

(*) Identifier can have numbers but it should not start with a 'number'.

Eg:

```
class 2bahubali
{       (X)
}
class $7
{       (✓)
}
class Kgf2
{       (✓)
}
class _8$
{       (✓)
}
class 9_$
{       (X)
}
```

Note:

- Good practices to be followed for standard coding.
- This is not any rule but follow for good-coding.

Class - Naming Conventions:

(*) Upper Camel Case / Lower CamelCase

Eg:

- class BigBasket
  {
  }
- class bigBasket
  {
  }

(*) It should never be a plural word. Always it should be a Singular name.

Eg:

```
class Pens
{
}
```
(X)

```
class Pen
{
}
```
(✓)

(*) Classname should be a Noun. It cannot be any verb or other word.

Eg:

```
class Friend
{
}
```
(✓)

```
class Write
{
}
```
(X) — Write: verb

(*) It is better to avoid (_) underscore & ($) dollar in classnames for maintaining Good practice

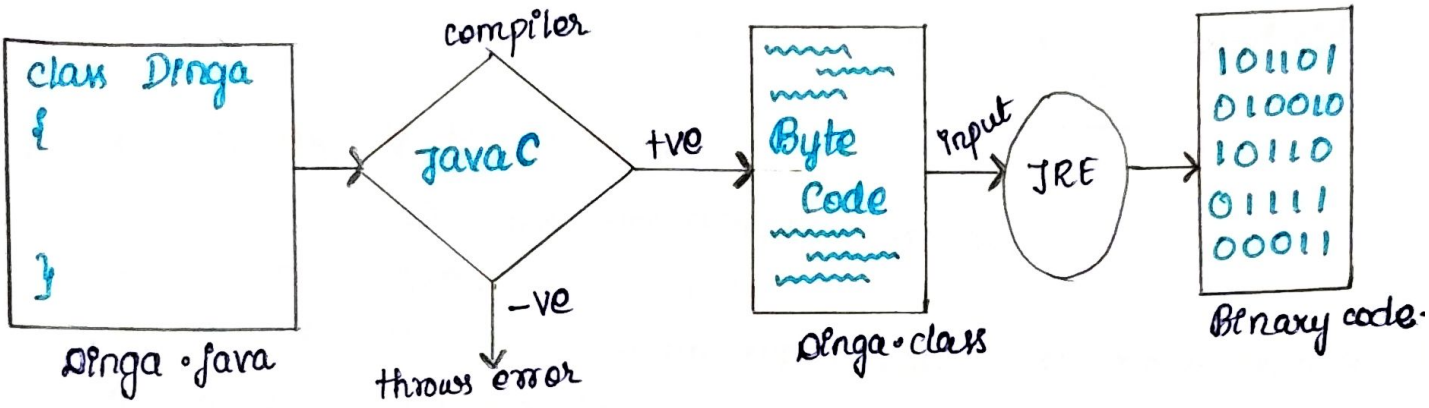
Compiler:

(*) compiler is a special software which is used to check the correctness of the java program.

(*) JavaC is the name of the java compiler.

Java Compilation:

In Java, Compilation is a two-step process.



Step-1: check the syntax of the java program.

Step-2: If the program is syntactically correct, then "Byte code" is generated.

Note: If a program is having any syntactical mistakes, then compiler throws compilation error. In this case, the compiler will not generate the Byte Code.

Byte-Code:

(*) "Byte code" is an instruction which is generated by the compiler (or) it is an inter-mediate code generated by the compiler, which is neither a high-level language nor a low-level language.

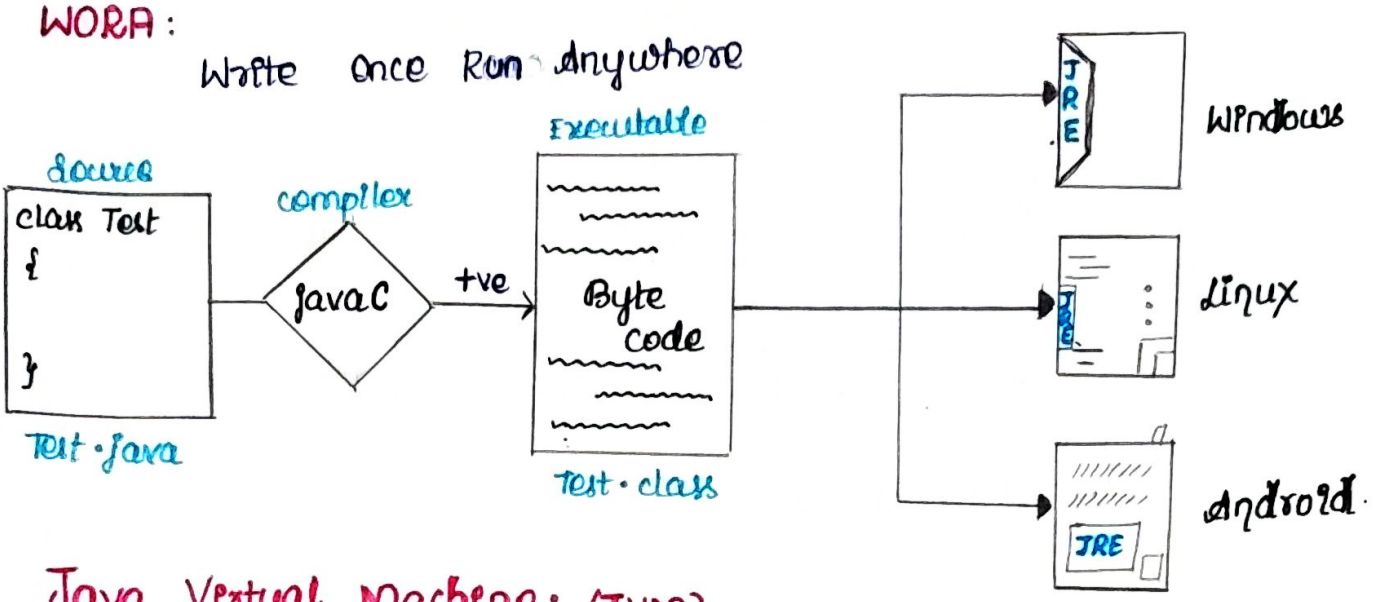
(*) Only a special software called "JRE-Java Runtime Environment" can understand the byte code.

(*) ".class" is an extension of Byte-code.

06/07/24

WORA:

Write Once Run Anywhere



Java Virtual Machine: (JVM)

(*) JVM is the specification for JRE.

Java Runtime Environment: (JRE)

(*) JRE is the physical representation or implementation of java virtual machine.

- JRE can be created & modified for any required 'OS'-operating system.
  for eg: - Android developed its own 'JRE' for Android OS.
  - This is created in a way that any user can create their own JRE, according to the requirements.

08/07/24

(*) JRE is a special software required to execute a java program (or) a java application.

(*) JRE is platform specific (or) platform dependent. (i.e., JRE of windows is different from JRE of Linux).

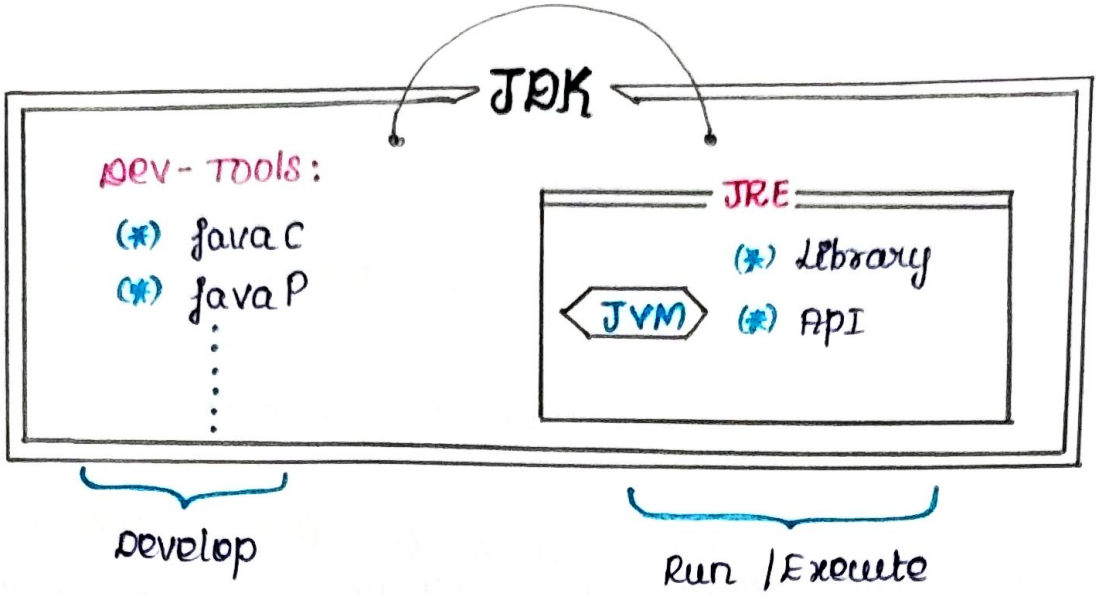
(*) Without JRE, we cannot execute the java applications.

(*) JRE can only understand the Byte code and cannot understand the java program.

Java development Kit: (JDK)

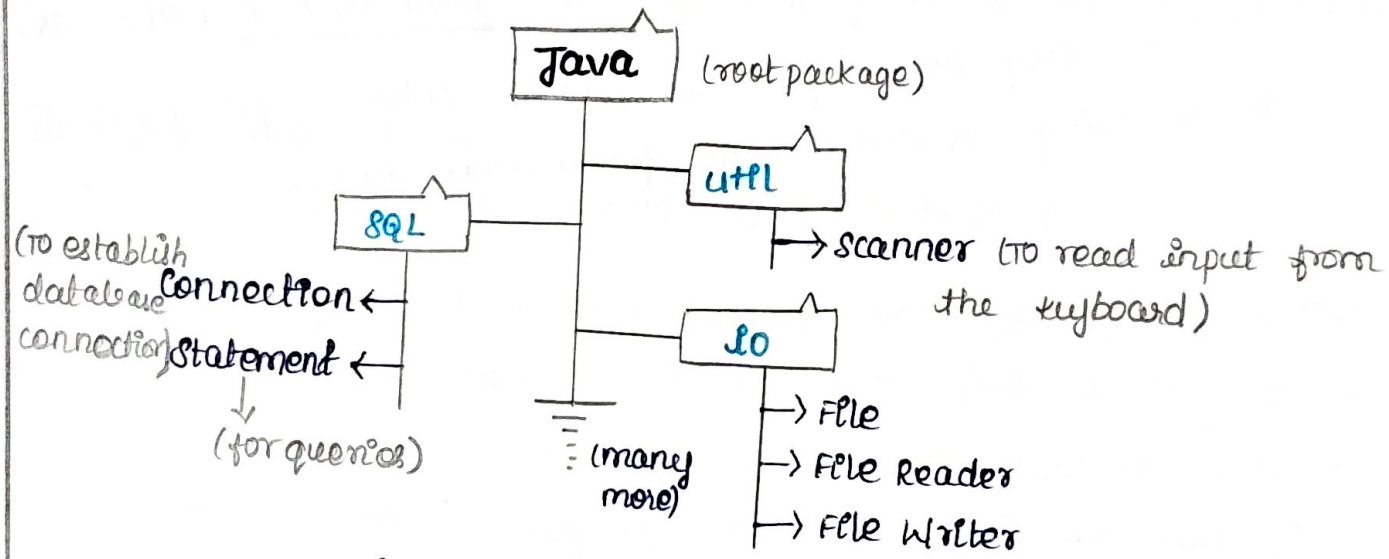
(*) JDK is a type of SDK (Software development Kit), which internally has both the compiler as well as the JRE.

(*) using JDK, we can develop as well as execute (or) run java applications.



Libraries:

(*) 'Library' is a collection of in-built java packages along with in-built programs.



Eg:- import java.io.filewriter

(*) In java, Libraries are found in JRE and JRE is found in JDK.

(*) Technically, 'package' is nothing but a 'folder' in java.

Note: We must use "import" statements to use the programs present in java Library.

Memory:

(*) In java, objects are stored in heap memory.

(*) In java, memory management is automatically taken care by JVM.

[Heap memory stores all the objects & arrays created by the application]

(*) object address is represented in Hexa-decimal format.

[Number formats (for understanding)]

Binary (2) : 0 and 1 ⇒ 01001110 1

Octal (8) : 0 to 7 ⇒ 0430 7510

Decimal (10) : 0 to 9 ⇒ 7810 286431

Hexa-Decimal (6, 10) : a-f and 0-9 ⇒ 98a 31fc6e

For memory storage - We have many more like heap, stack,

Jdk 1.8 Installation:.

Download jdk 1.8 and install it.

Open command prompt and type javac -version, then java -version; [make sure both versions are same].

In case, the version doesn't show up, instead some other msg appears then go to → program files
↓
Java
↓
bin
↓
Select & copy path

Open System settings, Go to Advanced system settings then .

10/07/24.

Objects.

(*) An object has states and Behaviours.

(*) State of an object is the property or the information which is used to describe an object. The state of an object is called as "data member" programmatically.

(*) Behaviour of an object is the action of the work performed by an object. The behaviour of an object is programmatically called as "method".

(*) A class can have multiple datamembers and multiple methods in any order.

Eg:

Considering "mobile" is an object.

States of Mobile	Properties of Mobile
- Brand	- call()
- Colour	- sendText()
- price	- takePhoto()
- RAM capacity etc.....	- playMusic() etc.....

Default Constructor.

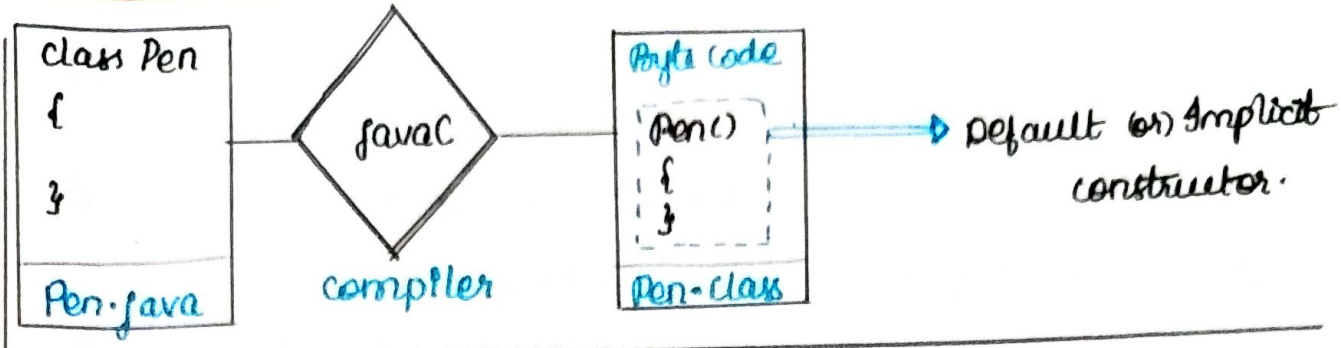
(*) After successful compilation of every java program, the compiler generates a default constructor as the part of Bytecode.

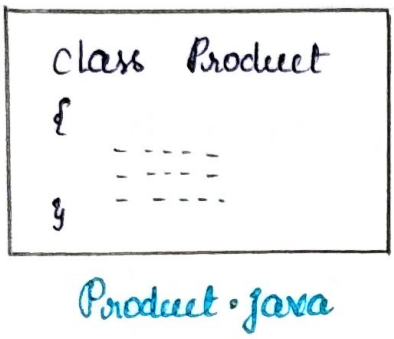
(*) Constructor name must be as same as the class name. (It is case-sensitive).

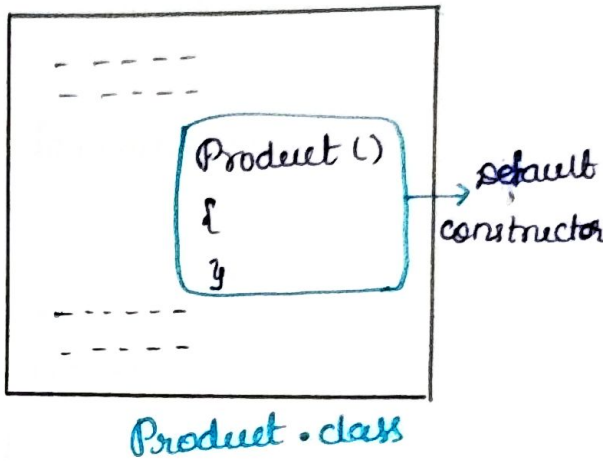
details - [In java, a default constructor is a no-argument constructor which the Java compiler automatically provides if no other constructors are defined a class. It initializes the object with default values.

Numeric fields are set to zero, object fields are set to null, Boolean fields are set to false.

- If a class has atleast one constructor with (or) without parameters, the compiler doesn't create default constructor







Object creation using "default Constructor":

(*) When we invoke a constructor using 'new'-keyword, then JVM allocates the memory in the heap area and the object gets created

```
class Envelope
{
}
```

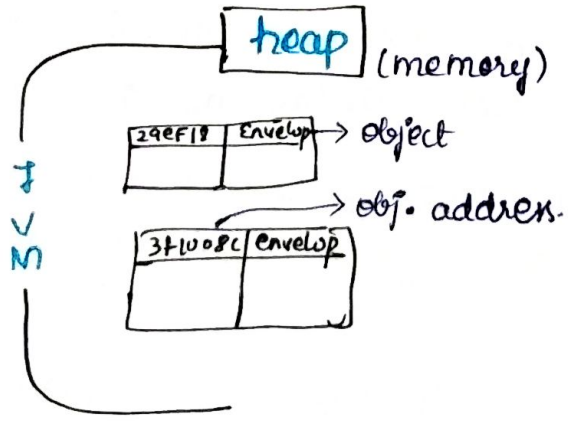
=> Envelope.java

=> envelope.class

```
new envelop();
new envelop();
```

→ This process of calling is called as "constructor invocation" "constructor call".

(*) A constructor can be invoked multiple times.



Reference Variable (or) Object reference:

(*) It is a container (or) a data holder which is used to store object address (reference) of an obj, rather than obj. itself.

(*) A reference variable refers to an object present in the heap area.

Syntax:

```
ReferenceType  refVar = new  Constructor();
```

[or]

```
Classname  refVar = new ClassName();
```

Examples:

(*) Employee e = new Employee();
(e is reference of type Employee).

(*) Animal a = new Animal();
(a is reference of type Animal).

(*) Product p = new Product();
(p is reference of type Product).

[(*) Reference variables are used to manipulate objects in java. It stores the memory address of an object rather than the object itself. (i.e.,) when any object is created, a space is reserved in heap memory for that object. By default, if no object is passed to a ref. variable, then it will store a null value. Ref. variable can also store null value.]

(*) Reference variable can store only one address at a time. Object is an inbuilt-class present in java library.

Generalisation: The process of representing multiple different objects by a common category or type is called generalization (basically).

⇒ object refVar = new Car();
(Car() → any obj.)

(*) public and default are the access modifiers. [Public is a java keyword which declares a member's access as public. Public members are visible to other classes. It means that any other class can access a public field or method.]

(*) If any java member is public, then it can be accessed within the package and also outside the package.

(*) If any java member is default(Non-public), then it cannot be accessed outside the package. [i.e., it can be accessed only within the same package in which it is declared. If we have not assigned any access modifier to the variables (or) methods (or) constructors (or) classes, then by default, it is considered as "default access modifier".

Block:

(*) It is a piece of code (or) some set of instructions which are written inside the curly braces.

(*) In java, Blocks cannot have any name. Hence, they are automatically executed by JVM.

Types of Blocks:

1) Non-static / Instance Block.
2) Static block.
3) Try-catch-Finally Block.
4) Synchronized Block.

Non-Static/ Instance Block:

(*) Instance Block is also called as object Block, which is automatically executed by JVM, when an objects gets executed.

(*) The number of times an instance block gets executed depends on the number of objects.

E.g:-

```
class mango
{
    {
     system.out.println ("I'm mango");
    }
    public static void main (string[] args)
    {
      System.out.println ("main");
      mango m1 = new mango();
      mango m2 = new mango();
      mango m3 = new mango();
    }

}
```

→ Instance block can be either before (or) after the main method.

→ If this object is not created then the Instance block will not be executed.

Result:-

```
main
I'm mango
I'm mango
I'm mango.
```

(*) We can have multiple blocks in a class and in this case, the order of execution depends on the sequence of blocks. [one object can call any number of Blocks].

Eg:-

```
class PaniPuri
{
    {
      System.out.println ("G");
    }
    public static void main (string[] args)
    {
      System.out.println ("main");
      new PaniPuri();
```

```
            new PaniPuri();
            new PaniPuri();
        }
        {
         System.out.println ("B");
        }
    }
```

Result:

```
main
G
B
G
B
G
B
```

15/07/2024.

Program Execution - step by step - in Java.

(*) The program execution starts from the line where the "MAIN function" is declared. The execution process will begin at the line after the open parentheses of main method declaration. It terminates at the line where the main method's parentheses is closed.

```
Class Name
{
    public static void main (String[] args)   → main method declaration
    {                                          ← open [starts]
        lines of code.
    }                                          ← closes [terminates]
}
```

Example - program:

```
1   class Mango
2   {
3       public static void main (String[] args)
4       {
5           System.out.println ("main start");
6           new Mango();
7           System.out.println ("main end");
8       }
9       .
10      {
11          System.out.println ("MANGO BLOCK");
12      }
13  }
14
15
```

Steps of execution:.

starts at 3
Goes to 4; 5 → Print → output [main start]
Goes to 6 → object created.
↓
goes to Block, [i.e., 10
11 → prints [MANGO Block]
12 → Block ends
Goes to 7 → prints [main end]
Goes to 8 → closing brace of main method.

→ Program Terminates ←

output:

main start
MANGO BLOCK
main end

Example - 2 - program :

```
1   class    PaniPuri
2   {
3       {
4         System.out.println ("G1");
5       }
6       public static void main (String [] args)
7       {
8         System.out.println ("MAIN START");
9         PaniPuri  p = new  PaniPuri ();
10        System.out.println (p);
11        new PaniPuri ();
12        System.out.println ("MAIN END");
13      }
14      {
15        System.out.println ("B");
16      }
17  }
18
```

Output:

```
MAIN START
G1
B
PaniPuri@515f550a
G1
B
MAIN END
```

Steps of execution :

Starts By → Main method
- Line no: → 6 Goes to → 7 (open brace)
- Line no: → 8 prints → MAIN START
- Line no: → 9 object created. [Right side]
- Goes to → 3 (Block-1) Then → 4 print → prints "G1". Then → 5 [Block ends] -1
- Goes to → 14 Then → 15 prints "B" Then → 16 [Block-2 ends]
- Goes to → 9 Reference variable generated. [left side]
- Goes to → 10 prints the hexa-decimal address.
- Goes to → 11 [Object is created again.] → Goes to → Block 1 Line → 3, → 4 (prints 'G1') → 5 (Block ends)
- Goes to → Block 2 Line → 12 → 13 → prints 'B' → 14 (Block ends)
- Goes to → 12 print → MAIN END
- Goes to → 13 Main method - ends.

— program terminates —

Data - Members:-

(*) Data-member is a container (or) a data holder which is used to store 'data'.

(*) [A data member refers to fields (or) instance variables that belong to a class.] It represents the memory in which data is stored. The two kinds of data members are :
- ⇒ VARIABLES
- ⇒ CONSTANTS.

(*) They are used to store the state of an object, created from the class.

(*) In another way, The variables which are declared in any class by using any fundamental data-type (like int, char, float, etc...) or derived data-types (like class, structure, pointer...etc) are known as <u>Data Members</u>.

15/07/24.

DATATYPES.

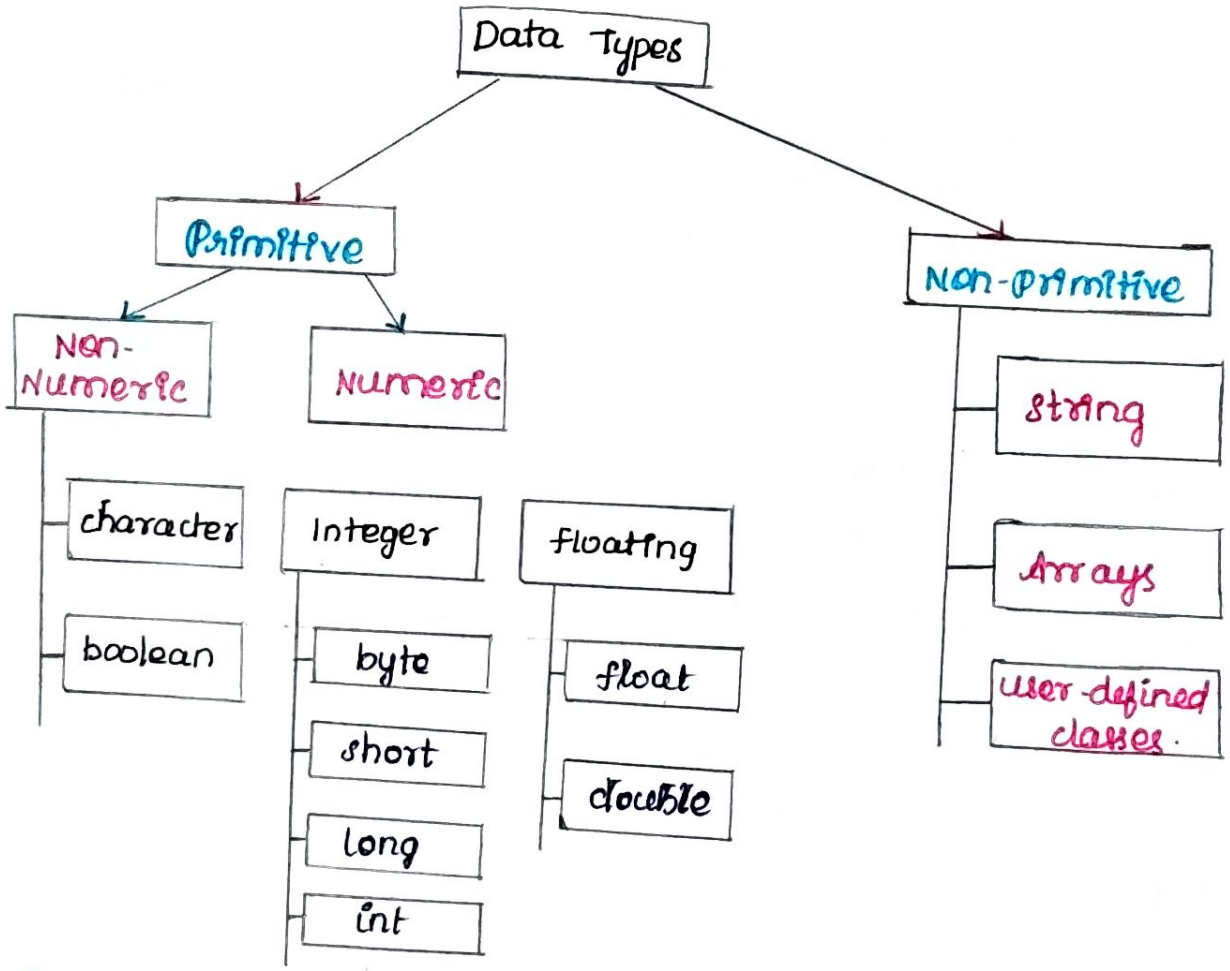
Datatype:

Datatype is the one which is used to describe the type of the data. Datatypes determine the kind of values that can be stored in variables.

Types of datatypes:

(*) Primitive Datatypes.

(*) Non-Primitive Datatypes.



Primitive Data-types:

(*) These are the pre-defined datatypes which are already present as the part of Java. They are present in the form of keywords in Java.

(*) They specify the size and type of variable values. They donot have additional methods.

(*) There are 8-predefined keywords present in java.

byte short int long
float double
boolean char

Default values for these (1) 8 datatypes:

Byte, short, long, int => 0
float, double => 0.0
boolean ⟶ false
String char ⟶ null.
char ⟶ \u0000

Memory allotted for data-types:

byte – 1 bytes
short – 2 bytes
int – 4 bytes
long – 8 bytes
float – 4 bytes
double – 8 bytes
boolean – 1 bit
char – 2 bytes.

Reference types:

String
Arrays
Classes
Interfaces.

NON-primitive datatypes:

(*) In java, it is possible to define our own datatype which is technically called as "non-primitive data types".

(*) These data types cannot be a "java keyword".

Example:

~ String
~ Array[]

(*) They are more complex and can also have methods and properties.

Default value:

(*) 'NULL' is the default value for all non-primitive datatypes. 'NULL' is a keyword.

Difference between Primitive and Non-Primitive data types.

Primitive Datatype	Non-Primitive Data type
(*) Pre-defined - They are already defined in java.	(*) User-defined - They are created by users (or) Programmers.
(*) Values only - They always hold a value.	(*) They may be null values & donot necessarily hold value.
(*) No Methods - They cannot define methods.	(*) Methods:- They can define methods and perform operations.
(*) Can't be null - They can't be null values.	(*) Null value - They are Nullable. they can have null values.
Eg: int, char, double, boolean.....	Eg: Strings, Arrays[].

VARIABLES.

Variable:-

(*) 'Variable' is a container (or) a data-holder which is used to store the data, (of any datatype).

(*) Programmatically, a non-final member is considered as variable. It is a storage location that holds a value. Their values can change as the program executes.

(*) Each variable in java has a specific datatype that determines what kind of data it can store.

Declaring a variable: To declare a variable, you need to specify the datatype and variable.

Syntax: DataType VariableName = Data;

Initializing a variable: We can initialize a variable at the time of declaration.

Example:
```
int number = 11;
String name = "Mitra";
long amount = 5786743120971L;
```
=> int num; num = 07; (✓) => String word; word = "NATURE"; (✓)

Variable Naming Rules / Conventions:

(*) variable name must begin within a letter (a-z, A-Z), dollar sign ($), or underscore (_). [Better to avoid $ or _].

(*) Subsequent characters can be letters, digits (0-9), dollar signs or underscores.

(*) these names are case-sensitive. [lowerCamelCase].

(*) these names cannot be a reserved keywords of java (like int, public, class, ... etc). They can be Nouns.

Examples:

```
byte age = 24;
int pinCode = 560029;
long contact = 8115453701L;
float weight = 58.6f;
double weight = 58.6;
char currency = '$';
char grade = 'A';

boolean govEmp = false;
boolean areYouMarried = false;
String mailId = "misbamr711@gmail.com";
String companyName = "Wipro";
String panNumber = "ADK253108";
```

CONSTANTS

constants: (*) It is a datamember (or a container which is used to store the data but the data cannot be changed, once assigned.

(*) constants are typically used to define fixed values that are used throughout the program, such as mathematical constant or configuration setting.

(*) Programmatically, constant is declared using the keyword "final".

Syntax:

```
final datatype CONSTANT_NAME = Data;
```

Example:

```
final double PIE = 3.1427T;
final String DOB = "11 Feb 1922";
```

Naming conventions: constants should be nouns, they are written in UPPERCASE letters. [can be seperated using underscores].

Keypoints about constants:

Final keyword: This makes a variable as a constant, meaning that its value can be assigned only once.

Initialization: constants must be initialized at the time of declaration. If not, they must be initialized in constructor [for instance variables.

Example:

```
final int k = 70; (constant declared & initialized).
        :
k = 50; (X) → can't be changed.
```

[compiler will throw an error, because it is already declared as "constant"].

VARIABLE TYPES: (Based on scope and lifetime).

The two types of variables in java based on their scope are:

(*) Local variables

(*) Global variables

⇒ Instance / Non-static variables.

⇒ Class / static variables.

Local Variables:.

(*) Any variable that is declared in the local scope is called local variable. That is, these are declared within a method, a block or a constructor. And, they can be used only within the method, the block or the constructor.

usage:. (*) These variables must be initialized before use.

Eg:

```
class Test
{
  { boolean b;
    System.out.println(b);   // [Compilation error is thrown. "Local var not initialized"]
  }
  Test()
  { int i;
    int j;
    System.out.println(i+j);
  }
```

(*) Local variable cannot be accessed outside the local scope.

Eg:

```
class Quiz
{
  {
    string name = "sid";
  }

  {
    system.out.println(name);
  }
}
```

→ compilation error is thrown.

(*) Two local variables within same local scope cannot have same variable name. If same name is used, it is considered as variable-duplication. Hence, compilation error is thrown.

Eg:

```
class Quiz
{
  {
    boolean sal = true;
    double sal = 75,800.96;
  }
}
```

↓ compiler throws error [variable duplication].

(*) Two local variables present in different or separate scopes can have same variable name.

Eg:

```
class Quiz
{ Quiz()
  {
    double sal = 95,700.87;
  }
  {
    boolean sal = true;
  }
}
```

(✓) valid to use.

(*) Local variable can be declared as "public".

(*) The stack memory is used to allocate space for local variables. When a method finishes execution of variable, the method will be removed from the stack.

Global Variables:

(*) In Java, Global variables concept doesn't exist. However, same functionality can be achieved using class variable. Any variable which is directly declared inside the class [global scope] outside local scope is called as Class Variable (Global Variable).

usage: (*) If we dont initialize a Global variable at the time of declaration, then it is initialized to default value. This default value depends on the datatype.

Eg:

```
class Test
{
  int i;
  string name;
  double s;
}
public static void main (string[] args)
{ Test t = new test()
  System.out.println (i +" " + name + " "+s);
}
}
```

Output:

0 NULL 0.0.

(0 ← default (int), (string) NULL, (double) 0.0 value)

(*) If global variable is public, then it can be accessed by other programs of other package. They are declared with 'static' keyword and associated with the class itself rather than in any specific (local) scope. (i.e.,) It can be accessed among all instances of class and accessed globally.

(*) Two global variables within the same scope cannot have same name.

Eg:

```
class Tree
{
  double height = 5.6;
  double height = 79.8;
}
```

compilation error is thrown. [variable duplication].

Instance Variables:

(*) An instance variable in java is a variable that is declared inside a class but outside any method, block or constructor. Each instance of the class (i.e., an object) has its own copy of variables. They are known as variables or field

(*) Instance variable represents the state of an object. It is created in the memory only when an object is created.

(*) Instance variable is stored inside an object as the part of heap memory.

(*) the number of copies of each instance variable depends on the number of objects

Eg:.

If we create 10 objects using class, each instance variable will have 10 copies

(*) If a instance variable is not initialized at the time of declaration, then it automatically initializes to the default value at the time of object creation by JVM. the default value depends on the datatype of variable.

Eg:. Instance variables

class is considered as a user-defined data type, bcz it is used to create reference variable, which depends on the class name declared by user.

```
class student
{
    int id;
    String name;
    double perc;
    public static void main (String args[])
    {
        student s = new student ();
        System.out.println (s.id + " " + s.name + " " +
                                                s.perc);
    }
}
```

→ also considered as non-primitive variable (user-defined)

Output : 0 NULL 0.0

↑ int default value

↑ String default value

→ double default value.

Different ways of initializing instance variable:

1. At the time of declaration. (Eg: int m = 79;)
2. Using an object reference / reference variable. (Eg: Roll r = new Roll(); r.id);
3. Using Instance block.
4. using constructor.
5. using method.

3. Initialize Instance Variable using Instance block.

It is possible to initialize an instance variable using an instance block.

Eg:

```
Class student
{
  int id;
  {
   id = 20;
  }
  public static void main (String Args[])
  {
   student s = new student ();
   system.out.println (s.id);
  }
}
```

CONSTRUCTOR:

(*) It is one of the members of a class which is used to initialize instance variable. It is called when an instance of a class is created.

(*) constructor's name should be as same as the class Name.

(*) constructors cannot have return type, [not even void.]

Keypoints:

Name: Must be as same as class.

Return type: NO - return type.

Invocation: Automatically called when an object is created.

Types of constructors:

(*) default / Implicit constructor.

(*) custom / userdefined constructor → zeroparam constructor
→ parameterized constructor.

Default Constructor:

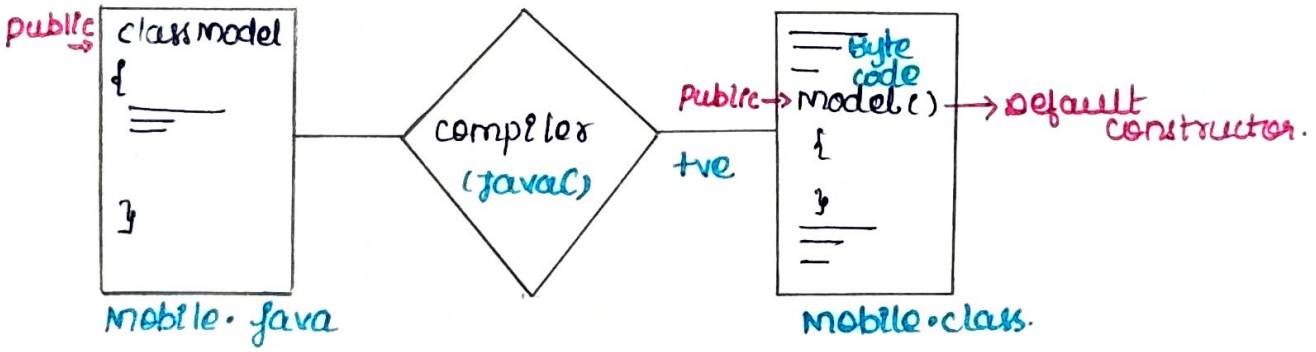
(*) It is a constructor which is automatically generated by the compiler.

(*) After successful compilation of every java program, the compiler generates a default constructor.

(*) In java, this is a no-argument constructor that the java compiler inserts automatically if no other constructors are defined in the class.

(*) If user has defined any constructor, then the compiler will not create a default constructor.

(*) The Access modifier of the default constructor is as same as the access modifier of class. [i.e., If the class is 'public', then the default constructor is also 'public'. If class is 'not-public', then the default constructor is also 'not-public'.



(*) Default constructor is used to initialize instance variables to default values. (These default values depends on the data Type.

Eg:
```
class Module
{
  String brand;
  Int price;
  public static void main (String args[])
  {
    Module m = new Module();
    System.out.println(m.brand);
    System.out.println(m.price);
  }
}
```

O/p:
NULL - String's def. value
0 - Integer's def. value

Customer Constructor:

(*) As a Java developer, we can define our own constructor is a class which is technically referred to as custom constructor or User-defined constructor.

(*) It is used to initialize instance variables to custom values.

(*) If we define a custom constructor in class, then compiler will not generate a default constructor. Hence, in a class, there can be either a default constructor or a custom constructor, But not both.

(*) Access modifier of custom constructor need not be as same as class and it can have its own access modifier.

1- Eg:

```
class Mobile
{
    Mobile()
    {
        System.out.println("custom/user-defined
                                constructor");
    }
    public static void main(String args[])
    {
        Mobile m1 = new Mobile();
        System.out.println(m1);
    }
}
```

O/P: custom/user-defined constructor
Mobile@51fdes*1

2-Eg:

```
1   class Mobile
2   {
3     String brand, model;
4     int price;
5     Mobile()
6     {
7       brand = "samsung";
8       model = "S24-ultra";
9       price = 2,50,000;
10    }
11    public static void main(String args[])
12    {
```

```
13  Mobile m1 = new Mobile();
14  System.out.println (m1.brand+" "+m2.model+"
                                    +m1.price
15  }
16 }
```

Execution: 11-13R - 3-4-5-6-7-8-9-13L - 14-15-> main end. (Terminate

Note: An instance variable of a class can be accessed within the same class either directly (or) by the use of "this" keyword

Eg:

```
class City
{
    String cityName;
    int pinCode;
    City()
    {
        this.cityName = "Bengaluru".
        this.pinCode = 560029;
        // cityName = "Bengaluru";
        // pinCode = 560029;
    }
    public static void main (String args[])
    {
        City c = new City();
        System.out.println (c.cityName +" "+
                            c.pinCode);
    }
}
```

Parameter: It is nothing but a kind of local variables which is declared within the parentheses of a constructor (or) a method. A constructor or method can accept any number of parameters, by which it allows us to initialize an object with specific values at the time of its creation.

Arguments:

(*) Argument is an actual data which is passed to a parameter, to constructor or method, by allowing them to perform specific tasks or create objects with specific characteristics.

Method arguments: values passed to a method.

constructor arguments: values passed to a constructor.

(*) Using a parameter, a constructor or a method can take any number of dynamic data.

(*) Arguments are the values passed to the methods when they are invoked. The two types of arguments in java are:

Formal arguments and Actual arguments.

Actual arguments: These are values or expressions that are passed to a method when it is called.

Eg:- `int result = sum(10,3); // 10,3 are actual args.`

Formal arguments: these are the parameters defined by the method that will receive the actual arguments.

Eg:-
```
public int add(int a, int b) // a,b are formal args.
    return (a+b);
```

Rules of arguments:

(*) Number of parameters and number of arguments should match.

(*) DataType of parameters and the DataType of arguments must be the same.

(*) The sequence of order of parameters and the sequence of order of arguments must match.

Zero-Parameter custom constructor:

(*) A user-defined constructor which does not take any parameters is called zero-param custom constructor.

⇨ [In java, the default constructor generated by the compiler is always a constructor with no arguments. Hence, it is also a zero-param custom constructor.

Parameterized Constructor:

(*) A custom constructor which takes one (or) more parameters is called parameterized constructor.

(*) It is used to initialize instance variables to dynamic values.

Eg:

```
class Pen
{
    String color;
    Pen (String c)
    {                    ↳ parameter
      this.color = c;
    }
    public static void main (String args[])
    {
     Pen P1 = new Pen("pink");
     System.out.println (P1.color);
     Pen P2 = new Pen ("Black");
     System.out.println (P2.color);
    }
}
```

P1.C = ⊗ [Because 'c' is in local scope, and it cannot be used out of local scope).

(✓), [parameter is of 'String' type, so the argument passed is of 'String' type]

(*) The arguments and parameters should be seperated with commas.

Eg: 1. (define constructor and assign parameters to arguments).

```
class Student
{
  int id;
  String name;
  double perc;
  Student(int i, String n, double p)
  {
    this.id = i;
    this.name = n;
    this.perc = p;
  }
}
```

Student.java

2. Define main method in a class and define a object for "student" java

```
class Test
{
   public static void main (String args[])
   {
      Student S1 = new Student(11, "Ria", 89.8);

      System.out.println ("ID:" + S1.id);
      System.out.println ("Name:" + S1.name);
      System.out.println ("Percentage:" + S1.perc);
   }
}
```

Test.java

Example-2:

```
class Person
{
   String name;
   int age;
   Person (String n, int a)
   {
      this.name = n;
      this.age = a;
   }
   public static void main (String args[])
   {
      Person P1 = new Person ("Bharath", 26);
      Person P2 = new Person ("Karan", 27);

      System.out.println ("NAME:" + P1.name + " " +
                          "AGE:" + P1.age);
      System.out.println ("NAME:" + P2.name + " " +
                          "AGE:" + P2.age);
   }
}
```

Person.java

5/7/24

Note:

Among constructor and instance block (non-static), the instance block is given highest preference by the main method. Hence, it is executed before the constructor. [It doesn't matter where block is located].

Example:

```
class Check
{
    int i=33;
    Check ();                                   ← constructor
    {
        System.out.println ("constructor:" +i);
    }
    public static void main (String args[])     ← main method
    {
    }
    {                                           ← Block
      System.out.println ("Block:" +i);
    }
}
```

Check.java

output:

Block: 33

constructor: 33.

Variable Shadowing:

(*) Variable Shadowing in java occurs when a variable declared within certain local scope (Eg: method, block, constructor) has the name as same as a variable declared in the class scope (i.e., Global scope). In this case, in the local scope, the variable "shadows" or hides the variable of Global scope, by making it inaccessible inside the local scope.

(*) A global and a local variable can have the same name and in this case, inside the local scope, the local variable dominates the global variable, which is referred to as variable shadowing.

Example:-

1.
```
class Test
{
  int i = 33;
  {                                  → Block started.
   System.out.println(i);            → prints 33, because 'i' is already initialized in global scope & the block scope 'i' is not initialized before it.
   String i = "Dinga";
   System.out.println(i);            → prints "Dinga".
   System.out.println(this.i);       → [prints "33", bcz "this" is used which calls the global scope var]
  }
  public static void main(String args[])
  {
    Test t = new Test();
  }
}
```

[Before local variable's value is initialized with same name, the global variable can be accessed with (or) without "this"]

output:
```
33
Dinga
33
```

2.
```
class Quiz
{
   String s = "Dance";
   {
     System.out.println(s);          → Dance
     System.out.println(this.s);     → Dance
     String s = "Sing";  // variable shadowing
     System.out.println(s);          → Sing [local var. is initialized]
     System.out.println(this.s);     → Dance
   }
   public static void main(String args[])
   {
     new Quiz();
   }
}
```

output:
```
Dance
Dance
Sing
Dance.
```

Note:
```
class car {
   car(String s = "colour")  → (X) parameter can't be initialized
   {
   int miles = 1124;         → (✓) variable can be initialized.
   }
}
```

Difference Between Local variables and Parameter:

Local variable	Parameter
(*) A local variable can be initialized at the time of declaration	(*) A Parameter cannot be initialised at the time of declaration.
(*) These local variables are initialized locally.	(*) It is meant to accept external values in the form of arguments
(*) They are created in the local scope (method, constructor or block) and destroyed once it is exited out of local scope.	(*) They are created when the method is called and destroyed when method's execution is completed.

(*) "Variable Shadowing" can even happen in case of parameter. [Eg Global variable and a parameter can have same name].

Example:

```
class Car {
    String clr = "Blue";  → Global variable
    Car (String clr) → constructor.
    {
        System.out.println (clr); // prints "Grey"  → object's referred value
        System.out.println (this.clr); // prints "Blue"
    }                          ↳ keyword to call
    public static void main (String args[])
    {
        new Car ("Grey");  Object → passing value to the global variable.
    }
}
```

27/07/24.

Practice [Ex].

(*)
```
class Employee
{
    int salary;
    Employee (int sal)
    {
        System.out.println("sal =" +sal);
        System.out.println ("this.salary =" +this.salary);
        sal = this.salary;
        System.out.println ("sal is: " +sal);
    }
    public static void main (String args[])
    {
        Employee e = new Employee (40,000);
    }
}
```

output:

sal = 40,000
this.salary = 0
sal = 0

(*)
```
1  class Car
2  {
3      String clr;
4      Car (String clr)
5      {
6          System.out.println(clr);
7          System.out.println (this.clr);
8          clr = this.clr;
9      }
10     public static void main (String args[])
11     {
12         Car c = new Car ("red");
13         System.out.println (c.clr);
14     }
15 }
```

12R ⇒ Object created (with argument)
3 → variable
4 → constructor & parameter
5 → opens
6 → prints "red", parameter assigned.
7 → prints null. [bcz G.var has no value]
8 → parameter = global var [null]
9 → closes
12L → Objects reference variable.
13 → prints null.

output:

red
null
null.

28/07/2024

Methods

Method:

(*) Method is one of the members of class which represents any functionality (or) work. In other words, Method is a block of code that performs a specific task and it is defined within a class.

(*) A method represents the behaviour of an object. A class can have any number of methods in any order.

Method Declaration:

Access modifier: public, private, protected (or) package-private (no modifier).

Return type: This denotes the data type of the value which the method returns. Eg: int, void, string.....

Method Name: The name of the method.

Parameters: A list of parameters given. [may be optional].

Calling a Method:

Calling Non-static methods: We need an object of the class to call Non-static methods. (Methods declared without "static" keyword). This can't be executed on its own. We should invoke it.

Calling static method: Static methods can be called using the class name directly. (i.e., when program executes any method with 'static' keyword executes automatically before "main" method. A method can be invoked any number of times. (multiple times).

Example:

```
class Pen
{
                    → method name
return type ← void Write()
{
  System.out.println("Writing");   → Non-static method
  // does not return anything
}
public static void main(String[] args)
{
```

```
        System.out.println("MAIN");
        Pen p = new Pen();  → object created.
        p.write();// method invocation.
        p.write();
        p.write();
    }
}
```

Types of methods:

(*) Concrete method.

(*) Abstract method.

1. Concrete Methods:

A concrete method is a complete method which has both method declaration and method definition / method implementation [method body + method logic].

Example:

```
void doubleClick() → method declaration.
{
    ≡≡≡   } → method definition / implementation.
}             [method body / logic].
```

2. Abstract Methods:

(*) An abstract method in Java is a method that is declared without an implementation. It is an incomplete method which has only method declaration without method definition.

(*) Abstract method must be terminated by (;) semi-colon without the body.

(*) Abstract method must be declared by using the keyword "abstract".

Example:

```
abstract void doubleClick();
```

(*) We cannot invoke an abstract method.

29/07/24.

Naming Conventions for a method.

(*) Must be a Verb.
(*) It should be in LowerCamelCase.
(*) It must be in simple tense.
(*) It is better to avoid $ and _.

Difference between Constructor and Methods.

Constructor.	Method.
(*) Name: Must be as same as class. (starts with Upper case)	(*) Name: can be anything [as per conventions]. (starts with lower, then lowerCamelCase)
(*) Return type: Doesnot have a return type.	(*) Return type: Must have a return type.
(*) Invocation: Invoked when object is created using keyword "new".	(*) Invocation: It is invoked through method calls. [depends on the type of method (static/non-static)].
(*) Inheritance: A constructor can not be inherited.	(*) Inheritance: A method can be inherited.
(*) It is a block of code that initializes newly created object.	(*) A method is a collection of statements which returns a value (except if void) upon its execution.

Example:

```
class Car
{
no return type → Car()
{
}                  // constructor
return type → void Car()
{
}                  // method
}
```

Note: can the method name be as same as class name?
Yes, But its a bad practice.

Main Method:-

(*) It is a spcial method that is used by the java virtual machine (JVM) to start the execution of a java program. It is dedicated for JVM.

(*) Main method is the entry point of any java application. (In java)

Key aspects of main method:

1) Syntax:

public static void main (String[] args).

(String args[]) or (String... args)

2) Modifiers:

public - The method must be public so that JVM can access it.

static - so that it can be called without creating an instance of the class.

void - method doesnot return any value.

String[] args - The method accepts a single argument, which is an array of string objects. [used to pass command-line arguments].

3) Functionality:

The main method serves as the starting point for the program (for execution). It can create objects, call other methods, and perform any actions required for the application.

Method - Calling rules:

① using other method.

A method can be called by other methods of the same class just by the name and here object reference is not required.

Example:.

```
class chef
{
  // calling method
  void prepareFood ()
  {
     bringVeg();
     washVeg();      } Method Call / Method Invocation.
     chopVeg();
     System.out.println("Done Cooking");
  }
  void bringVeg()
  {
     System.out.println("Bring");
  }
  void washVeg()
  {
     System.out.println("wash");
  }
  void chopVeg()
  {
     System.out.println("chop");
  }
  public static void main (String[] args)
  {
  }
}
```

Using object reference:

② We can call methods by creating objects. Any static member cannot directly invoke a non-static member, rather object is required. (reference)

Example:

```
class Car
{
    void start()
    {
        System.out.println("start car");
    }

    public static void main(String args[])
    {
        // start(); ERROR
        // drive(); ERROR.
        Car c = new Car();
        c.start();
        c.drive();
    }
    void drive()
    {
        System.out.println("Drive car");
    }
}
```

} using object reference. (method invocation).

③ Method of one class invokes method of other class.

A method of a class can invoke the other method of other class by using object reference (or) object.

Example:

Car.java

```
class Car
{
  void start()
  {
    sop("start");
  }
}
```

Driver.java

```
class Driver
{
  void drive()
  {
    sop("drive");
    Car c = new Car();
    c.start();
  }
}
```

Runner.java

```
class Runner
{
  public static void main
          (String args[])
  {
    Driver d = new Driver();
    d.drive();
  }
}
```

"Void."

(*) "void" is a keyword. It can be used only with a method. It indicates that the method does not return any data.

(*) When a method is declared with void, it means that it performs an operation but does not produce a result that can be used.

"Return-statement": It is used inside a method to return the data.

"Return-type": It is the dataType of the data returned by the method.

Syntax:

```
access_modifier  return_type  methodName (parameter1, param2, .....)
{                                                  optional.
   ===
}
```

Example:

```
class Account
{
   double fetchBalance()
   {
      ===
      double bal = 36,800.50;   <- data from dataBase
      return bal;               -> return statement
   }
}
```

return type: double

method: fetchBalance() { ... }

Method Rules:-

(*) "void" method can't return any data.

Eg:

```
void method()
{
   ===
   return 25;   -> compilation error.
}
```

(*) A method must be declared with "void" or with some returnType.

Eg:
```
doSomeWork()
{
  ≡≡≡
}
```
(X) - Compilation Error.

[No dataType / return type is given].

10/7/24

(*) A method can have only one dataType.

Eg:
```
int, double, boolean, add()
{
  // some code
  ≡≡≡
  return 33;
}
```
(X) compilation error.

(*) A method cannot return multiple data. To return multiple data from a method, we need to use either arrays (or) collection.

```
double math()
{
  return 25,36,10.7,9.1; X
}
```
Completion error.

Eg:
```
void doStuff()
{
  // some code
  return; ✓
}
```
↳ "return" statement without return value can be present in "void" meth

(*) A method cannot logically have multiple return statements.

```
int add()
{
  return 25;
  return 50;
  // some code
  return 20;
  // some code
}
```
(X) Multiple return statements.

(*) If a method has "return" statement, then it must be the last executable code inside the method. We cannot have any executable code after the return statement.

```
boolean math()
{
   boolean result = true;
   return result;
   System.out.println("hello!");
}
```

(X) There should be no code after "return" statement.

(*) We can initialize instance variables even by using method's (parameters).

Example:

1)

```
class Flower
{
   String type, colour;
   void init(String type, String colour)
   {
      this.type=type;
      this.colour=colour;
   }
}
```

```
class FlowerMain
{
   public static void main(String args[])
   {
      Flower f1=new Flower();
      f1.init("Rose","white");
      Flower f2=new flower();
      f2.init("lotus","pink");
   }
}
```

2)

```
class Account
{
   String name;
   double bal;
   long accNo;
   void init(String name, double bal, long accNo)
   {
      this.name=name;
      this.bal=bal;
      this.accNo=accNo;
   }
}
```

```
class AccountTest
{
   public static void main(String args[])
   {
      Account a1=new Account();
      sop(a1.name+a1.bal+a1.accNO);
      a1.init("Ria",987549.28,913485214089764);
      System.out.println("Name:"+a1.name+" "+"Balance:"+a1.bal+" "+"Account Number:"+a1.accNo);
   }
}
```

Output:

0.0 NULL 0 [Bcz it is not initialized]

Name:Ria Balance:98 75349.28 Account: 9134

01/08/24.

Anonymous Objects.

(*) An object without the reference is called Anonymous object.

(*) We can use anonymous objects in below situations:-

⇨ If we want to use one instance variable only once.

⇨ If we want to use method only once. [often used for one-time use].

Eg:

```
new Fan();

new Fan().increaseSpeed();
```

[one object to call one method only once].

Note: If we want to use multiple instance variables of an object, or multiple methods of an object, then we must use object reference.

Eg:

```
Fan f1 = new Fan();
f1.switchOn();
f1.increaseSpeed();
f1.decreaseSpeed();
f1.switchOff();
```

"this" - Keyword:

(*) 'this' is a keyword which refers to the current (or) present invoking object.

(*) 'this' keyword works like a reference variable which is used to access instance variables and instance methods.

(*) We cannot use 'this' keyword inside static method (or) static block.

(*) The most important usage of 'this' keywords is that it differentiates between the instance variables & parameters when they have same names. [the name used with 'this' is non-static (or) instance variable.]

Eg:-

```
class car
{
    string brand, color; →variables
    void drive()
    {
        System.out.println(this);
        System.out.println("driving" + this.brand + " " +
                                        this.color );
    }
    car(string brand, string color) →parameters
    {
        this.brand = brand;
        this.color = color;
        System.out.println("driving" + this.brand + " " + this.color);
    }
    public static void main(string args[])
    {
        car c1 = new car("kia", "Black");
        System.out.println(c1);
        c1.drive(); →//calling method.

        car c2 = new car("fortuner", "white");
        System.out.println(c2);
        c2.drive();
    }
}
```

method

construction

reference variable

object creation ←

Arguments.

Output:

```
car@ba01f5506 →→ c1 object reference address
car@ba01f5506 →'this' keyword from 'drive'-method invoked by c1.
driving kia Black // print statement.
car@7a1645706 → c2 object reference address
car@7a1645706 →'this' keyword from 'drive' method invoked by c2
driving fortuner white // print statement.
```

Eg:-

```
class Employee
{
    void work()
    {
        System.out.println ( this + "works");
    }
    public static void main (String[] args)
    {
        Employee e1 = new Employee ();
        System.out.println ("E1 = " + e1);
        Employee e2 = new Employee ();
        System.out.println ("E2 =" + e2);
        Employee e3 = new Employee ();
        System.out.println ("E3 =" + e3);
        e2.work();
    }
}
```

Output:

```
E1 = Employee @ 3d012ddd
E2 = Employee @ 7b1d7Hf
E3 = Employee @ 29990a69C
Employee @ 7b1d7Hf works.
```

same

Because we have invoked method 'work()' using 'e2' object only

'this' key word refers to the same.

2/08/24

Signature.

Signature:-

(*) 'Signature' is the characteristics which are used to describe the parameters.

(*) It is uniquely used to identify a method within a class. The signature doesnot include methods return type.

(*) There are three characteristic which are used to describe a signature. They are:
- (*) Number of Parameters
- (*) DataType of parameters
- (*) Sequence / order of parameter

(*) Signature is applicable for methods as well as constructors [i.e., only methods and constructors can have signatures].

Eg:

```
class Calculator
{
    void add (int fn , double sn, int tn, long ln)
    {
      System.out.println ( fn + sn + tn + ln);
    }
    public static void main (String args[])
    {
      Calculator c = new Calculator ();
      c.add (100, 11.365, 200, 1986531627896L);
    }
}
```

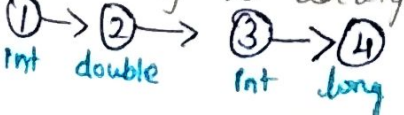
order / sequence of parameters should always match the order / sequence of arguments declared in method.

Parameter-executions:

① Number of Parameters ⇒ 4

② DataTypes of Parameters ⇒ 2 int
1 double
1 long

③ order / sequence of Parameters
(according to dataType order)



Method Overloading:

(*) Having multiple methods in a class with the same name but change in the signature is called method overloading.

(*) Change in the signature here refers to:

1. Either there has to be a change in the number of parameters.

(or)

2. There has to be a change in DataType of parameters.

(or)

3. There has to be a change in the sequence of parameters.

Eg:

```
public class Airtel
{
    void login (long cardNum, String expDate, int cvv)
    {
      // using Card
    }
    void login (String upiId, int uPin)
    {
      // using upI
    }
    void login ( String userName, String password)
    {
      // using netBanking
    }
    public static void main (String args[])
    {
    }
}
```

login => method name [same]

signatures => [different]

(*) Method Overloading is a way to achieve Compile Time Polymorphism (or) Early Binding.

(*) In case of method overloading, we only consider the method name and the signature but the method return type is not considered. [i.e., returnTypes of methods can be different for methods with same name] Example → → next page

Eg:

```
public class Website
{
    public void login(String userName, String password)
    {
    }
    public void login(long mobNum, int otp)
    {
    }
}
```

(same name) (same signature [no. of parameters is same])

(*) In a class if method names are same & signatures are same, then it is considered as method duplication [even if returnType is different]. Hence, we get compilation Error.

Eg:

```
class Quiz
{
    void m1()
    {
      //
    }
    int m1()
    {
      //
      return 24;
    }
    public static void main(String args[])
    {
      Quiz q = new Quiz();
      q.m1();
    }
}
```

→ throws "ERROR"

[Because, object doesnot know which method to be invoked as there occurs method duplication.]

Question:

Can we overload main method?

Ans:

Yes, Technically it is possible but it is un-necessary. Among, multiple main methods, the main method with (String args[]) gets executed first.

```
public void login (String userName, String password)
{
}
public void login( long mobNum, int otp)
{
}
```

(same name) (same signature [no. of parameter is same])

(*) In a class if method names are same & signatures are same, then it is considered as method duplication [even if returnType is different]. Hence, we get compilation Error.

Eg:

```
class Quiz
{
    void m1()
    {
    }
    int m1()
    {
        return 24;
    }
    public static void main(String args[])
    {
        Quiz q = new Quiz();
        q.m1();
    }
}
```

→ throws "ERROR"

[Because, object doesnot know which method to be invoked as it occurs method duplication]

Question:

<u>Can we overload main method?</u>

Ans:

Yes, Technically it is possible but it is un-necessary. Among, multiple main methods, the main method with (String args[]) gets executed first.

Example:

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

executes 1st.

output:

String.

Constructor Overloading.

constructor overloading:

(*) constructor overloading is a technique in java where a class can have multiple constructors in a class with change in the signature. [i.e., difference in the datatype (or) sequence (or) number of parameters)

(*) this allows an object to be initialized in different ways depending on the type of arguments passed during object creation.

Eg:

```
class Employee
{
    int id;
    double sal;
    String name, dept;
    Employee()  ---> zero param
    {
     // empty - zero param
    }
    Employee (int id) ---> one param
    {
        this.id = id;
    }
    Employee (int id, double sal, String name, String dept) -> 4 param
    {
        this.dept = dept;
        this.id = id;
        this.name = name;
        this.sal = sal;
    }
    public static void main (String[] args)
    {
        Employee e1 = new Employee (23); -> calls 2nd constructor
        Employee e2 = new Employee(); -> calls 1st constructor
        Employee e3 = new Employee (24, 9867.72, "Dudi", "IT")
                                        -> calls 3rd constructor
    }
}
```

3-same names constructor

[calls according to the type of arguments passed in each object.]

Methods to fetch data from table in a database:

(*) To fetch data from table in database through Java program, we use JDBC [Java Database Connectivity]. JDBC is an API (Application Programming Interface) that allows Java applications to connect and interact with database.

(*) This enables the user to execute SQL queries, update records and retrieve results from databases by providing a standard interface for database access.

JDBC - It contains a lot of pre-defined classes which together are called as JDBC-Libraries. → called as "Result set".

Eg:

E-Id	E-name	Dept	Salary
113	Sid	Development	1,60,000
number	varchar	varchar	Number → DataTypes as per the data stored in database.
int	String	String	int → These are Java datatypes for reference.

To retrieve the specific row's data, [consider '113']. i.e., EId = 113.

Syntax: getInt (String ColumnName).

- getInt (method (To get Integer value))
- (String ColumnName) ('Name' should be given. (from this column, we)
- (Integer number) ↓ column name (String)

Eg: ⇒ getInt ("EId");

From getInt (EId), the Integer value present in E.Id is retrieved.

column number →	1	2	3	4
	E-Id	E-name	Department	Salary
	113	Sid	Development	1,60,000
Data types →	int	String	String	int

↑ result

⇒ getInt ("Salary");

- To get Integer value
- ColumnName

Another way to declare methods, using column Number, Syntax: getInt(int columnNum);

Eg: ⇒ getInt (4);

- To get Integer value 'method'
- column Number.
- ⇒ 1,60,000 - (result) ↓ retrieved.

Syntax:

getString (String column_Name);

↓ method ⇒ To get string value.

↓ through column Name. ("String" - format)

Eg:

getString ("Dept");

⇒ Using getString method, the string value present in 'dept' is retrieved

Syntax: **getString (int column_Number);**

↓ method ⇒ To get string value

↓ through columnNumber ("int" - format).

Eg: getString (3);

⇒ using getInt method, the string value present in '3' is retrieved

05/08/24

1) Write a java program by following below scenarios:

- create a class called as 'student'.
- declare 2 datamembers called as 'age' & 'name'.
- Initialize the datamembers using a constructor. (variables)
- under main method create two instances (objects) of student & print their age, name

```
class Student
{
  int age;
  String name;
  Student (String name, int age)
  {
    this.name = name;
    this.age = age;
  }
  public static void main (String[] args)
  {
    Student s1 = new Student ("Bharat", 22);
    Student s2 = new Student ("Riya", 24);
    System.out.println ("Name:" + s1.name + " " + "Age:" + s1.age);
    System.out.println ("Name:" + s2.name + " " + "Age:" + s2.age);
  }
}
```

Output:

Name: Bharat Age: 22
Name: Riya Age: 24.

2) Write a java program to store 2 employee details. Print their name & sal in the format: "salary of Tom is 6.7 LPA".

```
class Employee
{
  String ename;
  double sal;
  Employee (String ename, double sal)
  {
    this.ename = ename;
    this.sal = sal;
  }
  public static void main (String[] args)
  {
    Employee E1 = new Employee ("Nisha", 4.5);
    Employee E2 = new Employee ("Ram", 8.9);
    System.out.println ("Salary of " + E1.ename + "is" + E1.sal + "LPA");
    System.out.println ("Salary of " + E2.ename + "is" + E2.sal + "LPA");
  }
}
```

output:

Salary of Nisha is 4.5 LPA.
Salary of Ram is 8.9 LPA.

Accessing Non-static members using same class.

```
class student
{
  // non-static or instance variable
  int age = 22;
  // non-static or instance method
  void study()
  {
    system.out.println("studying
                        java");
  }
  public static void main(string
  {                          args[])
    Student s = new student(); ⟶ In same class, instance is created.
    System.out.print("Age:" + s.age);
    s.study();
  }
}
```

[Student.java]

Note: // comments are the extra information which doesn't execute.

Accessing non-static members using different class.

```
class Car
{
  string Brand = "ferrari";
  void ride()
  {
    System.out.println("Riding car");
  }
}
```

[Car.java]

```
class Carmain
{
  public static void main(String[] args)
  { Car c = new Car(); ⟶ In different class, instance is created.
    System.out.println("car Brand:" + c.brand);
    c.ride();
  }
}
```

[CarMain.java]

06/08/24

'Static'

Static:

(*) ['Static' keyword is used to indicate that a particular member (variable (or) method) belongs to the class itself rather than to instances of the class.]

(*) "Static" is a keyword which can be used with variable, method, blocks and classes.

(*) All static members will have only one copy per class.

(*) All static members should be accessed using ClassNames.

(*) All the static members should be declared using "static" keyword.

Syntax:

```
ClassName.VariableName;
        (or)
ClassName.methodName;
```

Note:

- Static members can be accessed in same class using ClassName (or) even directly (without using ClassName).
- Static members can't be accessed directly in different class. It can be accessed only using ClassName in different class.

Static variable: A static variable is a variable that belongs to the class rather than instances of the class. There is always only one copy of the static variable.

Eg:
```
class Copy
{
    static int i = 20;  // static variable.
    // -----
}
```
[This can be re-initialized further in main method (or) block to change its value (✓)]

Static method: A static method belongs to the class rather than to any specific instance of the class.

```
class Example
{
    static void print() // static method
    {
        System.out.println("Printing");
    }
}
```

```
Example.print();
```

Static Block: A static block is used to initialize static variables. It is also known as static initializer. This block of code is executed when the class is loaded into memory.

Eg:
```
class Example
{
    static int f;
    static {
        f = 11;
    }
}
```
→ static variable initialization.
→ static Block

Example Programs

Accessing static members in SAME class.

```
Class Bike
{
  static String brand = "pulsar";
  static void ride()
  {
    System.out.println("riding Bike");
  }
  public static void main(String args[])
  {
    System.out.println("BIKE NAME:" + Bike.brand);
    Bike.ride();
    System.out.println("~~~~");
    System.out.println(brand);
    ride();
  }
}
```

Output:
```
pulsar
riding Bike
~~~~
pulsar
riding Bike
```

Accessing static members in different class.

```
Class Bike
{
static String brand = "jawa";
static void ride()
  {
    System.out.println("Riding");
  }
}
```

```
class MainBike
{
  public static void main()
  {
    System.out.println("Long Drive");
    Bike.ride();
    System.out.println("I love " + Bike.brand);
  }
}
```

Output:
```
Long Drive.
Riding.
I love jawa.
```

Another eg. for accessing static members in different class.

```
class Employee
{
    static int id = 124;
    static void work()
    {
        System.out.println("The
            employee is working!");
    }
}
---------
class MainEmp
{
    public static void main
                (String args[])
    {
        System.out.println("ID:"
                + Employee.id);
        Employee.work();
    }
}
```

Output:

ID : 124
The employee is working.

Example for both static and non-static members in same class.

```
class Pen
{
    static int x = 11; // static variable
                declared & initialized.
    int y = 29; // non-static/instance
    variable declared & initialized
    public static void main
                (String args[])
    {
        System.out.println("x:"+x);
            // printing static variable
        in same class directly. It
        can also be done using className as
                (Pen.x).
        Pen p1 = new Pen();
        Pen p2 = new Pen();
        System.out.println("y:"+p1.y);
            // printing non-static variable using
            reference variable
        System.out.println("y:"+p2.y);
```

Output:

X : 11
Y : 29
Y : 29

Difference between static and non-static members.

Aspects	Accessing in same class	Accessing in different class	Memory Location	Number of copies.
Static.	can be directly accessed (or) using class-Name.	can be accessed only using different class.	Static Pool (or) class Area.	only one copy of it is created.
Non-static.	By creating object.	By creating object.	Heap Area.	Number of copies depends on number of objects.

08/08/2024.

Arrays.

Array:

An array is a data structure (or) a container that allows you to store multiple values of the same datatype in a single variable. In other words, array is container to store a group of data.

Keypoints:

(*) **Fixed size:** The size of an array is fixed when it is created and this cannot be changed.

(*) **Zero-based indexing:** Array is index-based and the indexing starts from 0. followed by 1, 2, ...

(*) **Homogeneous Nature:** Array has homogeneous nature wherein we can store only one type of data [i.e., data of same datatype] like Int, String, double, ...

Syntax:

1. **DECLARATION:**

Syntax: dataType [] arrayName; ⇒ (most preferred type of declaration)

[or]

(valid)
- dataType arrayName[]; ⇒ Int a[];
- [or]
- dataType [] arrayName[]; ⇒ Int []a;

Example: Int[] a;

Example:

⇒ Int[] a ; [read as Integer array a]

⇒ double [] x; [double array x]

2. **CREATION:**

Syntax:

arrayName = new dataType [size];

- arrayName ↓ Name of the array
- new ↓ key-word
- dataType ↓ which type of array.
- [size] ↳ size of array to be declared.

Example:

① a = new int[4]; (Integer array with 4 values).

a → array name, new → key word, int → data type, 4 → size.

process of creation: (Internally). new array } a [ ][ ][ ][ ]

index value & default value } a [0|0|0|0] → def. value of int.
0 1 2 3
index start from 0

② x = new double[3]; (double array with 3 values).

process: new array } x [ ][ ][ ]

index value & default value } x [0.0|0.0|0.0] → default value of double.
0 1 2

3. DECLARATION & CREATION: (simultaneously).

Syntax:

dataType[] arrayName = new dataType[size].

declaration. creation.

Example:

① Integer array

int[] a = new int[3];

process: a (int) → array name. [0|0|0] → default values. 0 1 2 → index values. int → datatype

② String array

String[] s = new String[5];

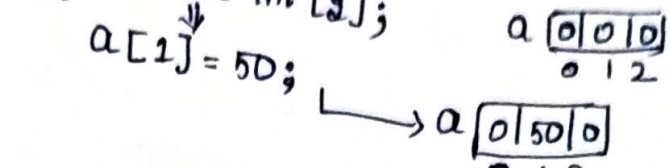
process: s (String) → array Name [null|null|null|null|null]; 0 1 2 3 4 → index values. String → data type. null → default values.

4. ARRAY INITIALIZATION:

Syntax:

```
arrayName [index/position] = value;
```

Example: `int[] a = new int [2];`



`a[1] = 50;` → a [0 | 50 | 0] (0 1 2)

(updated (or) initialized)

`a[2] = 28;` → a [0 | 50 | 28] (0 1 2)

initialized.

⇒ `String[] b = new String[2];` → & [null | null],

`b[1] = "pikaachu"; // Initialization.`

b [null | pikaachu] (0 1)

initialized.

5. DECLARATION, CREATION & INITIALISATION:

Syntax:

```
datatype [] arrayName = { V1, V2, V3, ..... };
```

declaration creation initialisation.

Example:

```
int[] a = {40, 70, 90};
```

a (int) [40 | 70 | 90] (0 1 2)

initialized values

index

6. LENGTH OF AN ARRAY:

Syntax: `arrayName.length;`

Example: `System.out.println(a.length);`

(for the above array)

OUTPUT: 3

09/08/24.

Storing User-defined objects inside an array.

```
class Employee
{
    int id;
    String name;
    Employee (int id, String name) // Parameterized constructor.
    {
      this.id = id;
      this.name = name;
    }
}
```

Shortcut keys to generate constructor.
→ [alt + shift + s]
↓ generate Constructor
→ select source
↓ generate Constructor

```
class EmpMain
{
  public static void main (String[] args)
  {
    Employee e1 = new Employee (22, "Ram");
    Employee e2 = new Employee(24, "Janani");
    Employee[] emp = {e2, e2};
```

Array declared, created and initialised in same step.

Employee[] → class name as datatype; emp → array name; {e2, e2} → array values; ref. variables passed as values in array.

(or)

```
    Employee[] emp = new Employee[2]; // declared & created
    Emp[0] = e1;
    Emp[1] = e2;
```

} Initialized manually.

```
    for (int i=0; i<emp.length; i++)
    {
      System.out.println("Employee Id:" +emp[i].name +" "+
                         "Is" +emp[i].id);
    }
  }
}
```

use (arrayname.length) for a good practice

Id of → arrayName[index]

Output:

Employee Id of Ram is 22
Employee Id of Janani is 24.

12/08/24

Inheritance.

Inheritance:-

(*) The process of one class acquiring the properties of another class is called as inheritance. This is a fundamental concept in object-oriented programming which allows one class to inherit properties & behaviours of one class.

(*) This concept is crucial for code reuse, modularity and establishing relationships between classes.

Key concepts:-

Super class - A class sharing the properties of itself is called Super class. It is otherwise called as parent class [or] Base class.

Sub-class - A class acquiring the properties of another class is called sub class. It is otherwise called as Child class [or] Derived class.

'Is-A-Relationship' - Inheritance is also referred to as "IS-A-RELATIONSHIP" in Java.

"extends" - In Java, Inheritance can be achieved using "extends" keyword.

NOTE:-

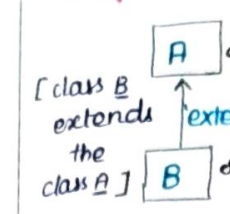
In Java, We can inherit only variables and methods. Blocks and constructors cannot be inherited.

Types of Inheritance:-

There are 5 types of inheritance. They are:

- Single level inheritance
- Multi level inheritance
- Multiple inheritance
- Hierarchical inheritance
- Hybrid inheritance.

Single - level Inheritance:

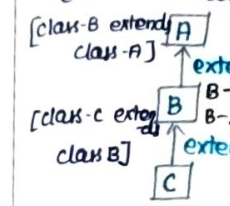


A single class (sub-class) inheriting from another single class (superclass) is called single-level inheritance. [In other words, a combination of one super class and one sub-class is called single-level inheritance]

Note: Java follows "Bottom-up" approach.

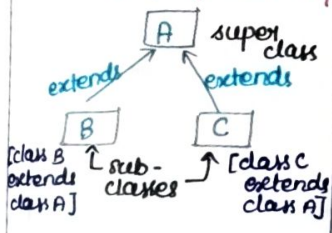
Multi-level Inheritance:

A class inherits from another super-class and one more class inherits from that sub class, is called multi-level inheritance. [A combination of multiple sub-classes and multiple super-classes in a sequential order is called multi-level inheritance].



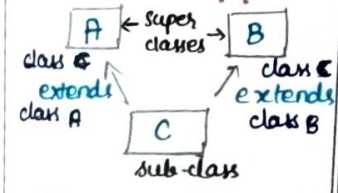
⟹⟹ Here, class B extends class A and again class C extends class B. So, class B - acts as a sub-class to class A and as a superclass to class C in a sequential order.

Hierarchical Inheritance:



A class sharing properties to more than one class, i.e., two or more classes inheriting from one same superclass is called Hierarchical inheritance. [A combination of one super class having many sub-classes is called hierarchical inheritance].

Multiple Inheritance:



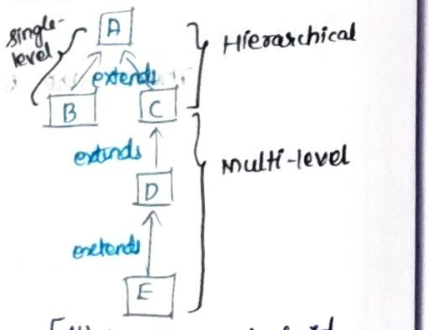
A combination of multiple superclasses with one sub-class inheriting them is called multiple inheritance.

Note:- Java doesnot support multiple inheritance.

Hybrid Inheritance:

A combination of two or more kinds of inheritance together is called hybrid inheritance.

Note: Java supports single-level, multi-level, Hierarchical and Hybrid inheritance (except when multiple inheritance is involved).



[All] together - hybrid inheritance

Example Programs.

Single-level Inheritance:

```
class Father
{
  string name = "Dolly";
}
```

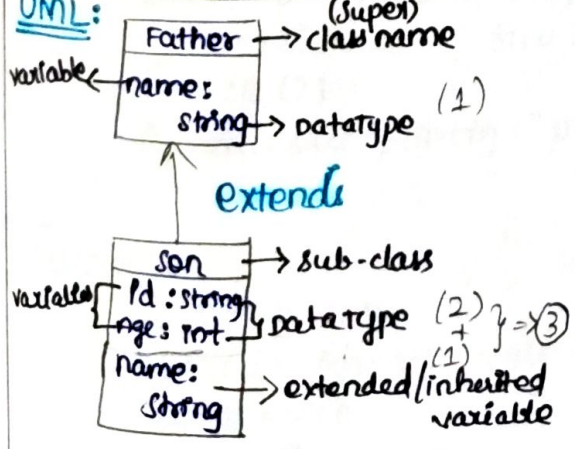
```
class Son extends Father
{
  string Id = "2946S";
  Int age = 24;
}
```

```
class Single main
{
  public static void main (String args[])
  {
    Son s = new Son();
// object for Son-class (sub-class) is created.
  System.out.println("NAME:" +s.name);
  System.out.println("ID:" +s.Id
      +" "+"AGE:"+s.age);
  }
}
```

Output:

NAME: Dolly

ID: 2946S AGE: 24



Multi-level Inheritance:

```
class Person
{
  void write()
  {
    System.out.println("Writing");
  }
}
```

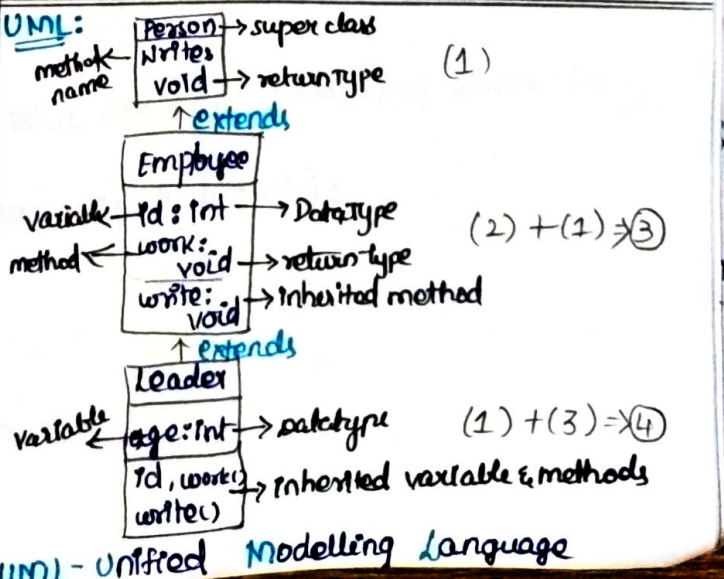
```
class Employee extends Person
{
  Int id = 208;
  void work()
  {
    System.out.println("Working");
  }
}
```

```
class Leader extends Employee
{
  Int age = 20;
}
```

```
class EmpMain
{
  public static void main (String args[])
  {
    Leader l = new Leader();

    System.out.println("ID:"+l.id+" "
                  "AGE:"+l.age);
    l.write();
    l.work();
  }
}
```

Output: ID: 208 AGE: 20
Writing
Working



UML - Unified Modelling Language

Example for Hierarchical Inheritance:

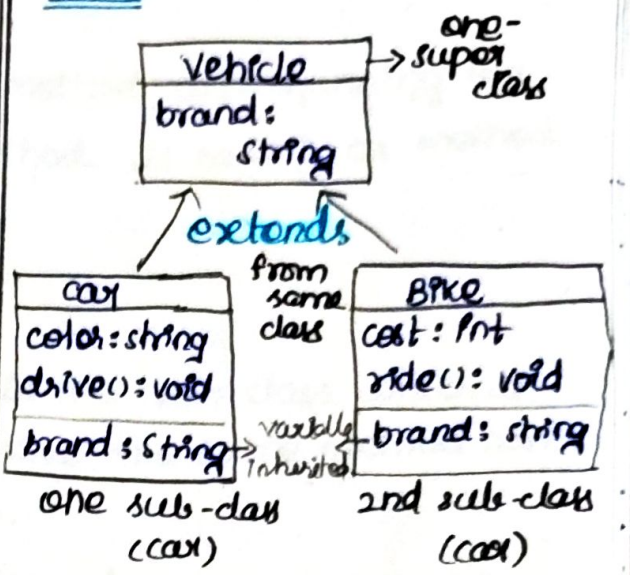
```
class vehicle
{
 string brand = "Ferrari";
}
class car extends vehicle
{
  string color = "Black";
  void drive()
  {
    System.out.println("Driving car");
  }
}
-----
class Bike extends vehicle
{
  int cost = 89978;
  void ride()
  {
    system.out.println("Long drive");
  }
}
-----
class Mainvehicle
{
  public static void main (String[] args)
  {
      car c = new car();
      // one instance of car (sub-class) extending Vehicle.
      c.drive();
      System.out.println("Brand:" +c.brand +" "+" color:"+ c.color);

      Bike b = new Bike();
      // one instance of Bike (another sub-class) extending same super-class
      b.ride();
      System.out.println("price: Rs." + b.cost);
  }
}
```

Output:

```
Driving car
Brand: Ferrari  color: Black
Long drive
price :Rs 89978
```

UML:



[Hierarchical Inheritance]

13/08/24

Method Overriding.

Method Over-riding:-

The process of inheriting a method and changing the implementation of that inherited method is called as method overriding.

Key-points:-

Same method signature: The method in the sub-class which is inherited from the super class must have the same method name, return type, and parameter's signature.

Annotation: The @Override annotation is optional, but it is recommended to use for a good practice. @Override indicates that we are overriding a method.

Access modifiers: The access modifiers of the method in the sub-class must be as same as the access malifier of method in super class.

Instance (Non-static), Static & final methods: Only Non-static methods can be overridden. 'static' methods and the methods declared as 'final' cannot be overridden.

Constructors: Constructors cannot be overridden.

Example:

```
class Student
{
  void study()
  {
    System.out.println("Study well");
  }
}
```

```
class Bubbly extends Student
{
  @Override
  void study()
  {
    System.out.println("Java...!");
  }
}
```

```
class MainOver
{
  public static void main(String[] args)
  {
    Bubbly b = new Bubbly();
    //Sub-class object
    b.study();
  }
}
```

Output:-

Java...!

'Super' - Keyword:

The "super" keyword in Java is used to access the members of super class. The members may include methods, constructors and variables.

Accessing super class methods and variables:

Syntax:

```
super.methodName();
        (or)
super.variableName();
```

(*) The 'super' keyword can be used to call a method from the superclass that has been overridden in the sub-class. This allows the sub-class to extend or modify the behaviour of the superclass method while using its own original functionality. The 'super' keyword is also used to access the variables of the superclass.

Example:

```
class RBI
{
  String bank = "RBI";
  void rateOfInterest()
  {
    System.out.println("Rate of Interest of RBI is 6%.");
  }
}
---------
class Hdfc extends
{
  String bank = "HDFC";
  @Override
  void rateOfInterest()
  {  super.rateOfInterest(); // calling superclass method
     System.out.println("Rate of Interest of HDFC is 8%.");
     system.out.println("Bank main:" + super.bank + " " +
                        "Bank sub:" + this.bank );
  }
}
---------
class MainBank
{
  public static void main(String args[])
  {  Hdfc h = new Hdfc();   h.rateOfInterest();  }  }
```

(sub class object) (sub-class method call)

output:

```
Rate of Interest of RBI is 6%
Rate of Interest of Hdfc is 8%
Bank main: RBI  Bank sub: HDFC
```

14/08/24

Constructor Chaining

constructor chaining:

The process of one constructor calling another constructor within the same class (or) between a super-class and sub-class is called constructor chaining in java.

Types of constructor chaining:

Within same-class: In java, within same class, constructor chaining can be achieved using this() statement [read as this-calling statement] where one constructor calls another constructor in the same class.

Among different-class: In java, constructor chaining can be achieved among different classes (i.e., super and sub class) using super() statement. [read as super-calling statement]. For this, the class must be inherited using IS-A-Relationship. This statement also ensures that the constructor of super-class is called before the sub-class constructor is called.

Example program for constructor chaining in same class. ①

```
class Car
{
  car(int a) //parameterized constructor
  {
    system.out.println(11);
  }
  Car() // zero-param constructor
  {
    this(99); // cons-calling statement with argument to the constructor
    System.out.println(13);
  }
  public static void main(String[] args)
  {
    System.out.println("start");
    Car c1 = new Car(); object to invoke zero-param constructor
    System.out.println("End");
  }
}
```

(Here 2nd constructor is invoking 1st constructor).

②

```
class Car
{
  car(int a)
  {
    this(); //const. invoking statement with arguments to call zero param constructor
    System.out.println(11);
  }
  Car()
  {
    System.out.println(13);
  }
  public static void main(String[] args)
  {
    System.out.println("start");
    Car c1 = new Car(50); //object to invoke constructor with one int-parameter
    System.out.println("End");
  }
}
```

(Here 1st constructor is invoking 2nd constructor).

understanding

Example program for Recursive chaining [which is NOT POSSIBLE in Java]

```
class Car
{
   Car(int s)
   {
     this();
   }
   Car()
   {
     this(25);
   }
   public static void main(String args[])
   {
      Car c = new Car();
   }
}
```

Invokes (Car(int s) → this();) — Invokes (Car() → this(25);) — Invokes (Car(int s) → this();) ... ∞

Infinite loop created & never stops which will throw an error.

Car c = new Car(); → Invokes zero-param constructor.

"Super"()-statement in constructor chaining:-

(*) Super() calling statement in Java is used to achieve constructor chaining in different classes with the help of IS-A-Relationship.

(*) Super() calling statement can be used in 2 ways.

~ Implicit super() statement.

~ Explicit super() statement.

Implicit super() Statement:-

(*) When we create an object of a class and if that class has a super class and if that super class has a zero-param constructor, then the sub-class will call the super-class implicitly.

(*) In other words, if super class constructor is non-parameterized, sub-class constructor will call implicitly.

Example:

```
Package com;
class Father // super-class
{
   Father() // zero-param constructor
   {
     System.out.println(1);
   }
}
```

```
Package com;
class Son extends Father
{
   Son()
   {
     //super(); [implicitly called]
     System.out.println(2);
   }
   public static void main(String[] args)
   {
     Son s = new Son();
   }
}
```

Explicit super() calling statement:

(*) When we create an object of a class and if that class has a super class and if that super class has a parameterized constructor, then the sub-class constructor must call the super-class constructor explicitly. otherwise, we get compile time error.

(*) Sub-class constructor should call super-class constructor using super(arg-1, arg-2, ...) by passing arguments of same type.

(*) In other words, if super class constructor is parameterized then the sub-class constructor should call it explicitly.

Example:-

```
package com;

class Vehicle // super-class
{
  Vehicle(String brand) // parameterized constructor
  {
    System.out.println("Hello");
  }
}
```

Output:

Hello
Bye

```
package com;

class Bike extends Vehicle
{
  Bike()
  {
    super("Audi"); // super() calling statement explicitly
    System.out.println("Bye");
  }
  public static void main(String args[])
  {
    Bike b = new Bike();
  }
}
```

Note: ~ this() and super() statement should be the first executable statement always within a constructor.

~ Java doesnot support recursive chaining. (i.e. if we have 2 constructors, we cannot have 2 super() or this() statements).

~ Therefore, if there are 'n' number of constructors, then we should have (n-1) number of calling statements.

15/08/24.

Advantages of Constructor chaining * Inheritance:-

(*) Code redundancy * Code repetition is avoided.

(*) Code reusability is increased.

Write a program by following the below scenarios:

1) Create class Person
2) Declare 3-datamembers - age, name, height
3) Declare 3- overloaded constructors wherein 1st will initialize only age, 2nd will initialize age & name and 3rd will initialize age, name, height. [ACHIEVE CONSTRUCTOR CHAINING]
4) Under main method create Person object and print their age, name and height. [Eg. Program for constructor chaining with object * WITH reference variable]

```
class Person
{
  int age;
  String Name;
  double height;
  Person (int age)
  {
    this.age = age;
  }
  Person (int age, String name)
  {
    this(age); // calling 1st constructor.
    this.name = name;
  }
  Person (int age, String name, double height)
  {
    this (age, name); // calling 2nd constructor [CONSTRUCTOR CHAINING]
    this.height = height;
  }
```

continued

```
  public static void main (String[] args)
  {
    Person p = new Person (24, "Rasim", 5.10);
// object created * passing argument to 3rd constructor * invoking 3rd constructor.
    System.out.println ("NAME:" + p.name);
    System.out.println ("AGE:" + p.age);
    System.out.println ("HEIGHT:" + p.height);
  }
}
```

Output:

NAME: Rasim
AGE: 24
HEIGHT: 5.10

Example program for constructor chaining with object but WITHOUT reference variable:

```
class A
{
  A(int x)
  { this(11.2); //calling
    System.out.println(1);
  }
  A(double x)
  {
    system.out.println(2);
  }

class B extends A
{
  B(String x)
  { this(10); //calling
    System.out.println(3);
  }
  B(int x)
  {
    super(10); //calling
    system.out.println(4);
  }
}
```

```
public static void main(String args[])
{
  System.out.println("START");
  new B("Java"); //object without reference variable.
  System.out.println("END");
}
}
```

Output:

START
2
1
4
3
END.

```
1 class Car
2 {
3   String brand;
4   int cost;
5   Car(String brand)
6   {
7
8     this.brand = brand;
9   }
10  Car(String brand, int cost)
11  {
12      this("Ferrari");
13      this.cost = cost;
14  }
15  Public static void main(String args[])
16  {
17    Car c = new Car("Jaguar", 999999);
18    System.out.println(c.brand);
19    System.out.println(c.cost);
20  }
```

Execution:

14→ main function / main method →15(

16→ Object with 2 parameters created. so automatically invokes respective constructor in line number 9→10(

11→ this(" ") statement invokes the same class constructor in line 5→6(

7→ parameter "Ferrari" is passed to brand.
8)

12→ cost value is assigned by argument.

13)

17→ prints brand value } using ref. variable.
18→ prints cost value

19) 20) class terminates

main() terminates.

Output:

Jaguar
999999

FINAL - Keyword.

"Final":

"final" is a keyword which can be used with variables, methods and classes.

Final-Variable: We cannot re-initialize the final variable which is otherwise called as CONSTANT. That is, its value cannot be changed once it is initialized.

Eg:-
```
final int MAX = 1000;
-----
-----
-----
MAX = 1050; // compilation error.
```

Final-method: ~ If a method is declared as final, it cannot be overridden by sub-classes. (This is useful when we want to ensure that the implementation of method remains unchanged).

~ We can inherit a final method.

Eg:-
```
class Parent
{
  final void display() // final method
  {
    System.out.println("This method is final, so can't be overridden");
  }
}
-----
class child extends Parent
{
  void display() // compilation error (attempt to override "final" method).
  {
    System.out.println("ERROR");
  }
}
```

final-class: ~ If a class is declared as "final" it cannot be inherited.

Eg:-
```
final class Erase // final class
{
  ≡≡≡
}
-----
class part extends Erase // Compilation error.
{                        Attempting to inherit final class.
  ≡≡≡
}
```

16/08/24

Access Specifiers
[OR]
Access Modifiers.

~ Access specifiers are used to specify the accessibility (or) visibility of classes and methods and variables, constructors.

~ The four different access modifiers in Java are:-

* public
* private
* Default
* protected.

1) "Public" :- The "public" access modifier allows the class, method (or) variables to be accessible from any other class, whether they are in the same package or in a different package.

Example:

```
Public class MyClass
{ public int num;
    public void display()
    {
    }  System.out.println ("public class-method-variable");
}
```

2) "Private":- The "private" access modifier allows the class, method (or) variable to be accessible only within the same class. It can't be accessed from any class even within same package.

Example:

```
class Number
{
    private int num;
    private void show()
    {
        System.out.println ("private class-var-method");
    }
}
```

3) "Default":- When no access modifier is specified, it is known to be declared "default". This modifier allows the class, method or variable to be accessible from any class within same (or) own package. It can't be accessed outside the package. It is also known as "package-private".

Example:

```
class Book                    'default access'
{
  int price;
  void study()
  {
    System.out.println("default class-method-variable");
  }
}
```

4) "Protected":- The "protected" access modifier allows the method or variable to be accessible within its own package (like default access) and also by sub-classes (even if they are in different packages only if inherited).

Example:

```
class Parent
{
  protected int n;
  protected void disp()
  {
    System.out.println("Protected var-method");
  }
}
-------
class child extends Parent
{
  void show()
  {
    System.out.println(n); // accessible after inheriting 'parent'.
  }
}
```

Summary:

Scope	Outer class	Inner class	Variable	Method	Method	Constructor
Public	✓	✓	✓	✓	✓	✓
Private	X	✓	✓	✓	✓	✓
default	✓	✓	✓	✓	✓	✓
Protected	X	✓	✓	✓	✓	✓

Scope	Accessibility in SAME CLASS	Accessibility in DIFFERENT CLASS	Accessibility in DIFFERENT PACKAGE
public	Yes	Yes	Yes - Globally
private	Yes	NO	NO → class level
default	Yes	Yes	NO → package level.
protected	Yes	Yes	Yes. 1) IS-A-Relationship

Accessing Public members in different packages.

① package com;
public class Person
{
public String name = "Tom";
public static int age = 25;
}

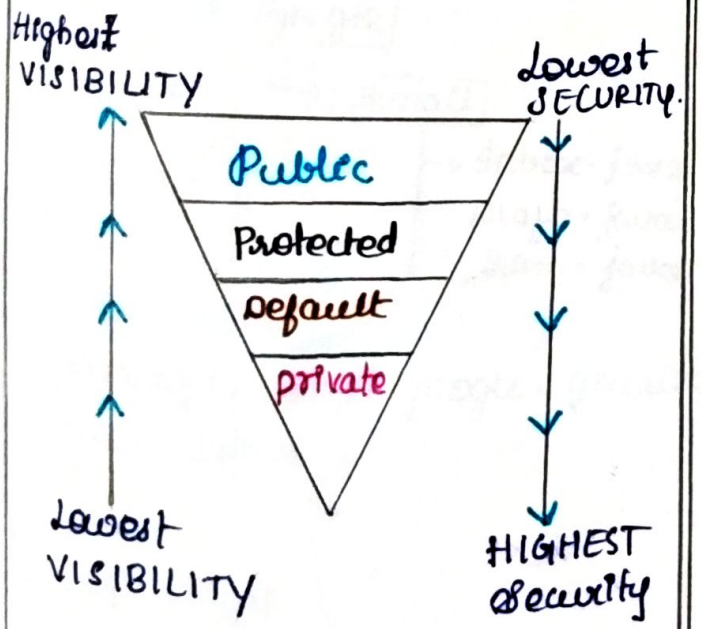
② package org;
import com.Person;
public class PersonMainclass
{
public static void main(String[] args)
{
Person p = new Person();
//Object for person-class created.
System.out.println("Name:" +p.name);
System.out.println("Age:" +p.age);
}
}

Accessing Protected members in different packages.

① package com;
public class Amazon
{ protected int cost = 2000;
protected void buy() // Protected method.
{
System.out.println("shopping");
}
}

② package org;
import com.Amazon; // Inherit package & class
public class Shop extends Amazon
{
public static void main(String[] args)
{
Shop s = new Shop();
s.buy(); // method invocation
System.out.println(s.cost);
}
}

O/P: shopping
2000



Note:

(*) A class can only be public and default. But, a inner class can be public, private, default & protected.

(*) We can use (or) access the protected members in a different packages using a is-A-relationship and import statement but Super class should be public.

19/08/24

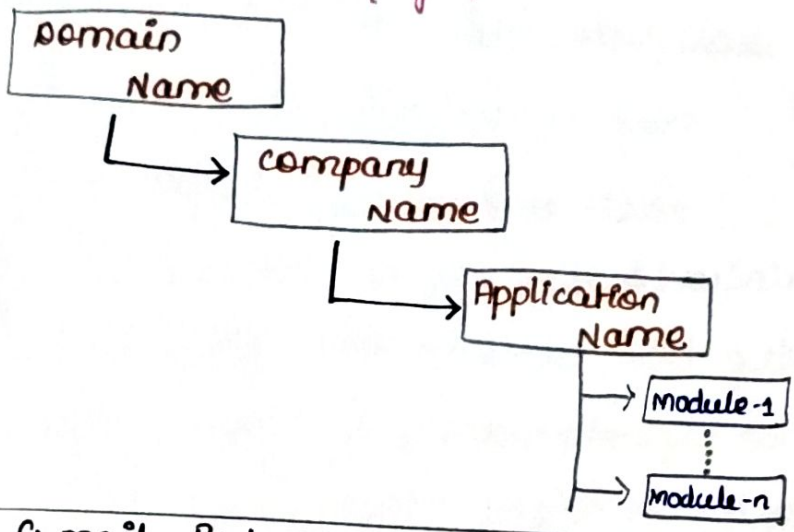
Package.

(*) package is a folder which is used to share a set of java programs.

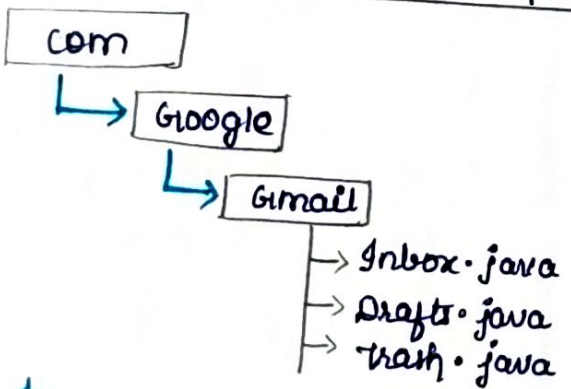
(*) searching of programs becomes every way easier with the help of packages.

(*) There can be a good maintenance with the help of packages. (Basically packages are folders.)

Structure of a Package:.



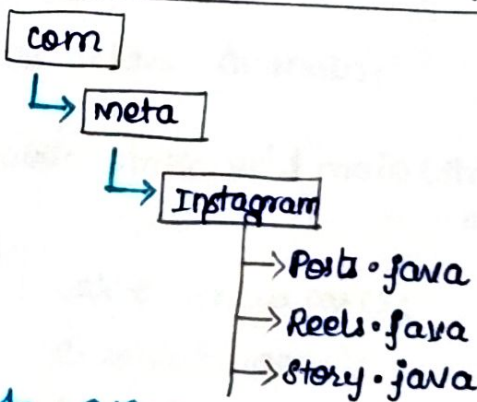
G-mail - Package - Example



In Eclipse,

```
package.com.google.gmail;
class Inbox
{
}
class draft
{
}
class trash
{
}
```

Instagram - Package - Example



In Eclipse,

```
package com.meta.Instagram;
class Posts
{
}
class Reels
{
}
class Story
{
}
```

Encapsulation.

Encapsulation:-

(*) The process of grouping (or) Wrapping of data members (variables) and member functions (methods) inside a single entity as class is known as Encapsulation.

(*) In simple words, Binding of data members and member functions inside a class is called as Encapsulation.

(*) By default, Java supports encapsulation. But, the example as best is Java Bean Class.

Rules to develop Java Bean Class:-

1. Public non-abstract class.
2. Private data members (variables).
3. Public setter method and getter method.
4. Public Non-Parameterized constructor.
5. Class should implement Serializable interface.

Example for Java Bean class:

```
public class Car
{
    private int cost;
    public void setCost
    {
        this.cost = cost;
    }
    public int getCost()
    {
        return cost;
    }
}
```

```
public class CarMain
{
    public static void main (String []
                                args)
    {
        Car c = new Car();
        c.setCost(10000);
        int price = c.getCost(); //store in a
                    variable and retrieve cost.
        System.out.println(price);
        System.out.println(c.getCost());
                                // print directly.
    }
}
```

Setter - method:

(*) It is used to set / initialize the data.

(*) The access specifier of setter method should be public and the return type of setter method should be void.

Getter - method:

(*) It is used to get / return the data.

(*) The access modifier of getter method is public and the return type will change according to the data we return.

(*) Whenever a method is returning some data, we have to store it and print it.

Naming conventions (or) coding statements by oracle:

1) class - Car, AudiCar, SartCar ByCost. [Upper Camel Case]
2) variable - age, studentAge, noOfStudents. [Lower Camel Case]
3) method - display(), main(), getId(), checkEvenOrOdd(). [Lower CamelCase]
4) packages - lower case ⇒ com, org.
5) constants - UPPER CASE ⇒ PI, MAX.

Java - Information Note:

1) Company which started Java - Sun Microsystems.
2) Company which owns Java - Oracle.
3) Person who started Java - James Gosling.

Difference between final method & private method:

(*) Final methods can be inherited but cannot be overridden.

(*) private methods cannot be inherited.

Advantages of Encapsulation or Java Bean class:

(*) Readability of the program is improved.

(*) We can avoid illogical initialization by performing validation.

(*) We can make the data as:
- Read only by having only getter method.
- Write only by having only setter method.
- Read and write by having both getter & setter method.

Note: Performing Validations.

```
public void setAge (int age)
{
    if (age>0)
    {
        this.age = age;
        sop ("valid age");
    }
    else
    {
        sop("Invalid age");
    }
}
```