JAVA

Java:

* High Level programming Language.

↳ [Communication with machine or System]

Ex: Java, C, C++, python, .Net, C#....

[Human understandable Language]

* Object Oriented programming Language

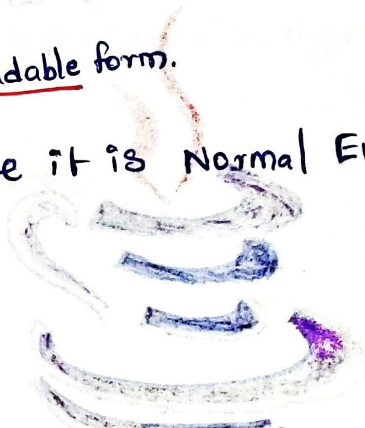
programming Language:

* programming Language is a medium to Interract with a System.

High Level Language:

(or) Human readable form.

* Human understandable Language it is Normal English form is known as High Level Language.



History:

* James gosling is the father of Java and Introduced in 1991.

① Green Talk

* The Software was named as green Talk and the team which develop is called green team.

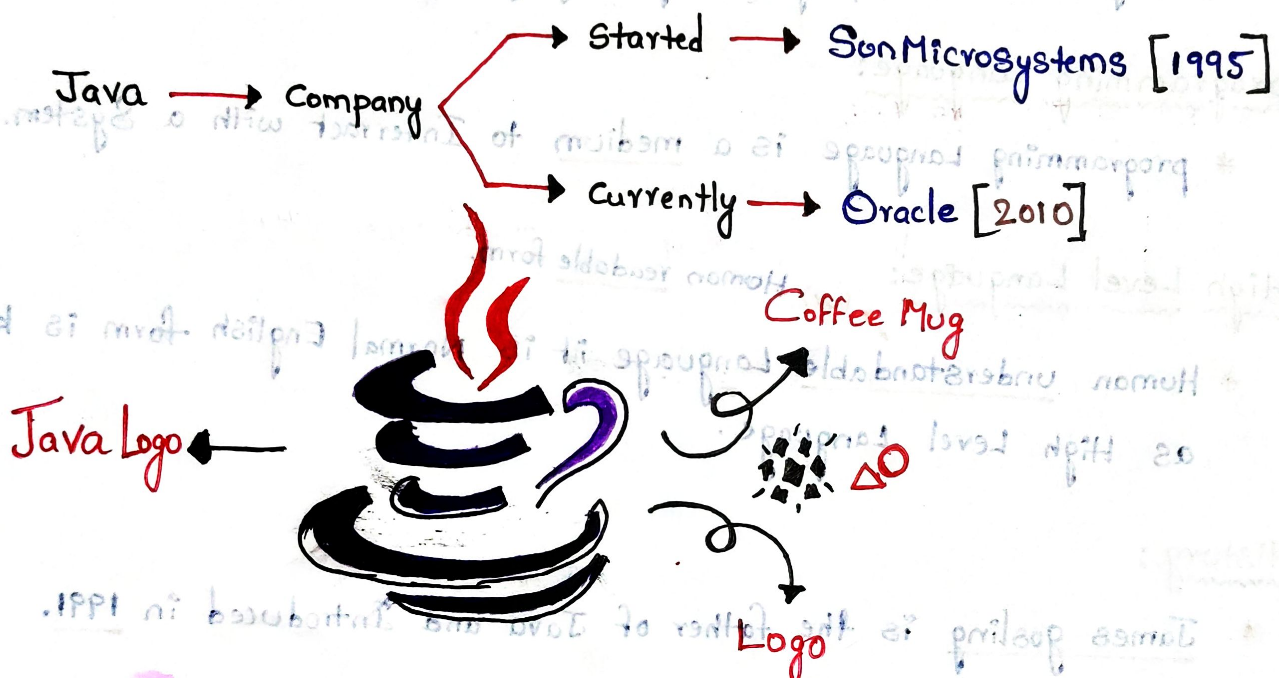
Abdulbasid Shaik

② Oak

* Oak is a Symbol of Strength and it is a national tree for Germany.
* There was already existing Company Called Oak technology which had become a legal issues due to this legal issues they changed the Software name. Oak to java.

Oak ⟶ ③ Java

* James gosling and his team went for a coffee to an island and the coffee shop named was Java. hence they kept the software name as Java,
* since they went for a coffee they kept the coffee Mug as a logo for the java software.



* Java is a high-level programming Language.
* Originally developed by Sun Microsystems and released in 1995.
* Java runs on a variety of platforms, such as windows, Mac OS, and the various versions of UNIX.

***

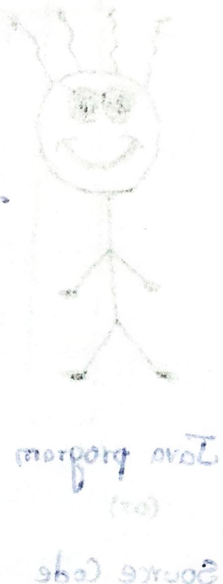
Features of Java:

1. Highly Secured Language → It is Secored programming Language.
2. platform Independent Language → Write Once Run Anywhere [WORA]
3. Robust → Strong → Easy to debug
4. Object Oriented programming → Objects, → class
5. Easiest Language → Syntax easy
6. Automatic Garbage Collection → Cache → Heap area

Structure of Java program:

* Java instructions are always written inside the class.

→ Class Name First letter Should be upperCase.



```
class Class_Name
{                                         // class blocks
    public static void main(String[] args)
    {                                     // main blocks
        //statements.
    }
}
```

Filename: class_Name.java

* NOTE: Every class in java must have a name, it is known as class name.
Every class has a block, it is known as a class block.

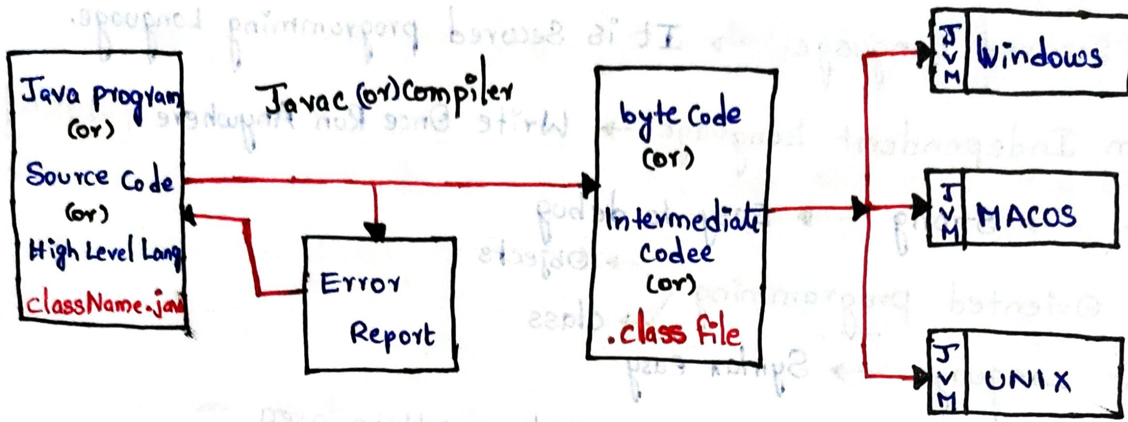
Example:

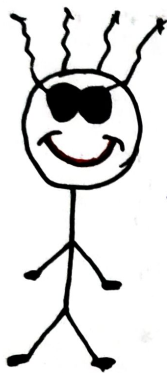
```
class Abdulbasid
{
    public Static void main(String[] args)
    {
        System.out.println("Hello world!");
    }
}
```

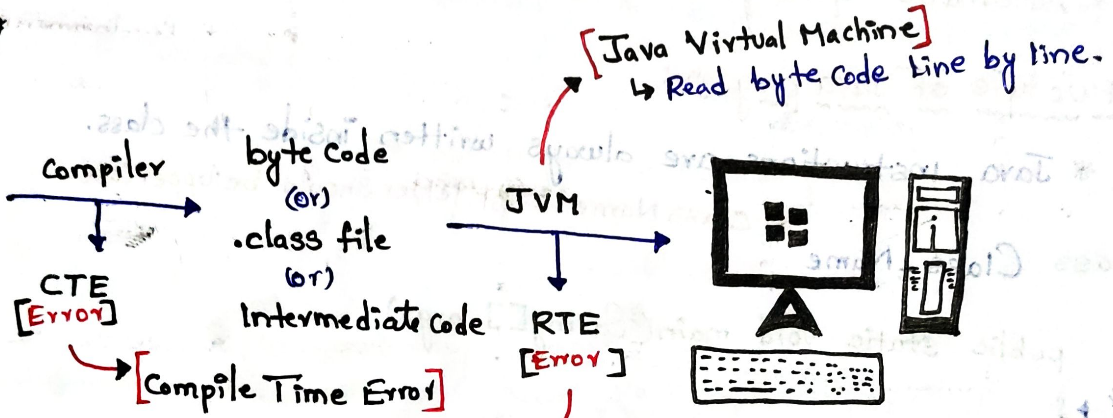
→ What is System.out.println?

* System is a class
* Out is a global static reference variables.
* println is non-static method of print Stream.

Platform Independent:







Java program (or) Source Code (or) HLL or HUL] - className.java

↳ [Human understandable Language]

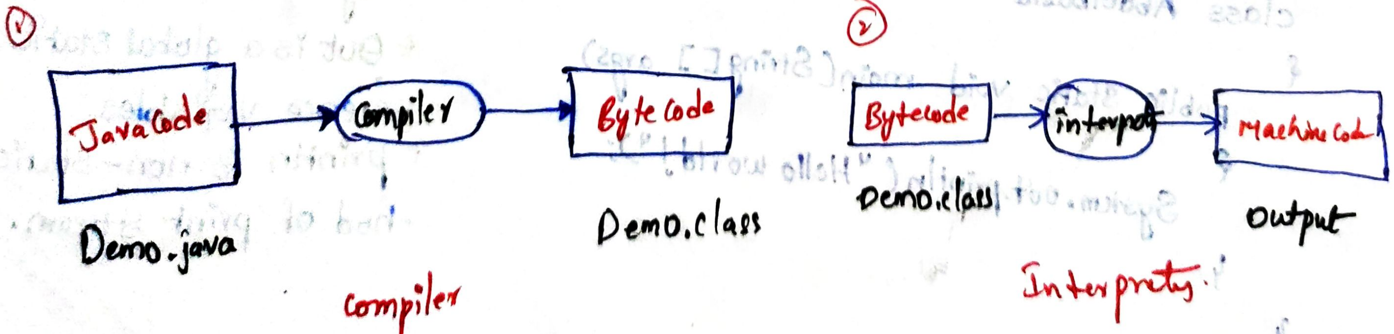
↓ [High Level Language]

✓0's and 1's
or
✓Binary code
or
✓MLL or MUL

↳ [Machine understandable Language]

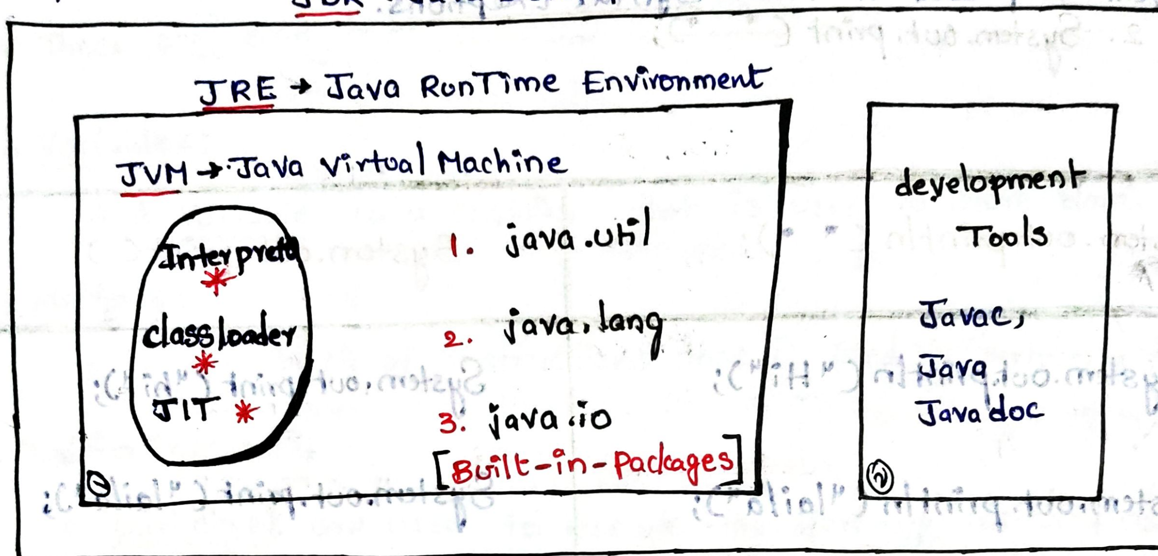
↓ [Machin Level Language]

↳ Java architecture



JDK Architecture:

* Abbreviated as Java Developement kit is a package which consists of Java development tools like java compiler and JRE for execution.



JRE [Java RunTime Environment]

* It is a Environment which consists of JVM and built in classes which is required for the execution of Java program.

JVM [Java Virtual Machine]

* It converts byte code instruction line by line into current system machine level language with help of an interpreter.

It is called a virtual machine because it doesn't physically exist.

Printing Statements: The prints method simply print text on the console and does not add any new line. while println adds new line after print text on console.

1. System.out.println(" ");

* print method works only with input parameter passed otherwise in case no argument is passed it throws Syntax Exceptions.

2. System.out.print(" ");

System.out.println(" ");	System.out.print( );
System.out.println("Hi");	System.out.print("hi");
System.out.println("laila");	System.out.print("laila");
System.out.println( );	System.out.print( ); //CTE → Compile Time Error
System.out.println("Coffee?");	System.out.print("Chai");
System.out.println("NO");	System.out.print("yes");

Output:

Hi
laila
-........
Coffee?
NO

Output:

hi laila chai yes

Class Members:

* In class block we can create.

1. Variables.
2. Method
3. Initializers.

* These are said to be members of a class.

1. Variables:

* A Variable is a container that is used to store data.

2. Methods:

* It is a block of instructions that is used to perform a task.

3. Initializers:

* Initializes are used to execute the start-up instructions.

Note:

* A class in java can be executed only if the main method is created as follows.

Syntax: Syntax to creating the main method.

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

Note:

* We can create a class without main method. It is compiled time Successfully and the class file is generated but we can't execute that class

Tokens:

* The Smallest unit of programming language which is used to compose instructions are known as tokens.

Types of Tokens:

1. Keywords
2. Identifiers
3. Literals

Keywords:

* A predefined word which the java compiler can understand is known as a keyword.
* Every keyword in java is associated with a specific task.
* A programmer can't change the meaning of a keyword (can't modify the associated task)

Example:

we have 50+ keywords in java....

1. Abstract	11. Continue	21. for	31. null	41. switch
2. Assert	12. default	22. goto	32. package	42. synchronized
3. Boolean	13. do	23. implements	33. private	43. this
4. Break	14. double	24. import	34. protected	44. throw
5. Byte	15. else	25. instanceof	35. public	45. throws
6. Case	16. enum	26. int	36. return	46. transient
7. Catch	17. extends	27. interface	37. short	47. try
8. Char	18. final	28. long	38. static	48. void
9. class	19. finally	29. native	39. strictfp	49. volatile
10. const	20. float	30. new	40. super	50. while.

Rule: keywords are always written in lower case.

Identifiers:

* The name given to the components of java by the programmer is known as Identifiers.
* Identifier means any name which is given by programmers.
* Identifier in java are symbolic names used for identification.

List of Components:

* class
* method
* Variables
* interface
* enum. etc.,

Note:

* A programmer should follow the roles and conventions for an identifier.

Rules of an Identifiers:

1. Identifier should not starts with digits.

Example:

→ num1 ✓
→ 1Demo ✗
→ m1 ✓
→ m2 ✓

2. while declaring identifiers we should not use any special characters expect dollar ($) and underscore (_)

Example:

→ num@1 ✗
→ num 1 ✗
→ num-1 ✗
→ num$1 ✓
→ num_1 ✓
→ _num ✓
→ $num ✓

3. character space is not allowed in identifiers.

Example:

→ Abdu l ✗

→ Ab dul ✗

→ Abdul ✓

→ Abdul1 ✓

4. keywords should not be used for identifier.

Example:

→ static ✗

→ void ✗

→ else ✗

→ class ✗

→ Static ✓

→ String ✗

CONVENTIONS:

* The coding or Industrial standard to be followed by the programmer are known as conventions.

Note:

* Compiler doesn't validate the convention, therefore if conventions are not followed then we won't get a compile-time error.
* It is highly recommended to follow the convention.

Convention For className, variable, MethodName.

className	Variable, Method Name.
1. Single word: The First word should be in upper case remaining in lower case.	1. Single word: All words should be inlowercase:'
Ex: Addition, ✓ Calculator ✓ Abdulbasid ✓ Sum ✓ etc... ✓	Ex: addition ✓ calculator ✓ abdulbasid ✓ sum ✓ etc...
2. Multiple-word: The First character of every word should be in uppercase remaining in lower case.	2. Multiple-words: the First word all characters is in ~~up~~ lowercase and from Second word should be in uppercase remaining in lowercase.
Ex: SquareRoot ✓ PowerOfDigit ✓ FactorialOfDigits ✓ AbddulbasidShaik ✓	Ex: squareRoot ✓ sumOfDigits ✓ factorialOfDigits ✓ abdulbasidShaik ✓ powerOfDigit ✓

* Pascal case
  → PascalCaseExample

* snake_case
  my_variable

* Camel case
  squareCase

* Kebab Case
  my-variable

Literals:

* The values or data used in a java program is known as literals.
* The data is generally categorized into two types.

    1. Primitive Values/Literal
    2. Non-primitive Values./Literal

1. Primitive Literals:

* Single values data are called primitive values. or primitive literals.

Types of primitive literals:

1. Number Literals:

* Integer number literals.

    Example: 1, 4, 67, 24, 35, etc....

* Floating number literals are decimal literals/values.

    Example: 2.3, 1.0, 35.5,634, 23.53, etc...

2. Character Literals:

* Anything which is enclosed within a single quote ('') is considered as a character literal.
* The length of the character literals should be one.

Example: 'a', 'G', '1', '$', etc..

3. Boolean Literals:

* Boolean literals are used to write logical values. we have two Boolean literals.

    → true

    → false

2. NON-Primitive Literals:

* The multi valued data is known as Non-primitive value (group of data).

1. String Literals:

* Anything enclosed within a double quote (" ") is known as String literals.
* The length of the string literals can be anything.
* They are case-sensitive.

Example: "hello", "true", "a", "123", "hello@", "1.1", etc...

2. className Literals:

* Every className is a Non-primitive Literals.

Example:

```
class Abdulbasit
{

}
class A
{

}
class B
{

}
```

Abdulbasit, A, B → className Non-primitive Literals.

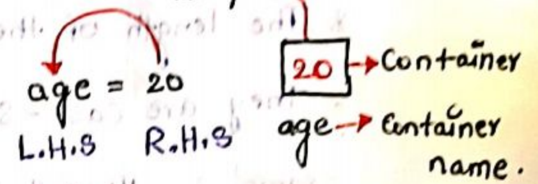
Variables:

* A Variable is a container that is used to store a single value.
* We have two types of variables, they are.

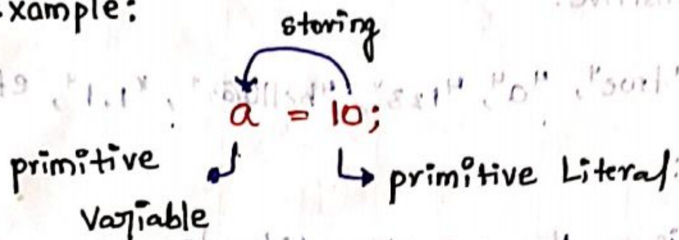
1. primitive variables
2. Non-primitive variables

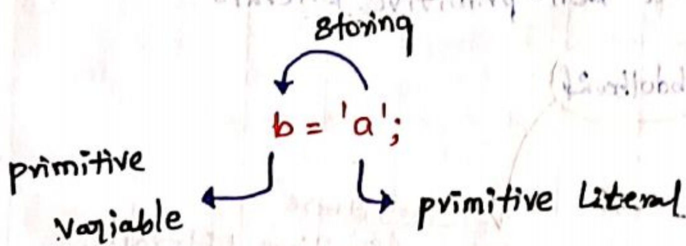
1. primitive Variables:

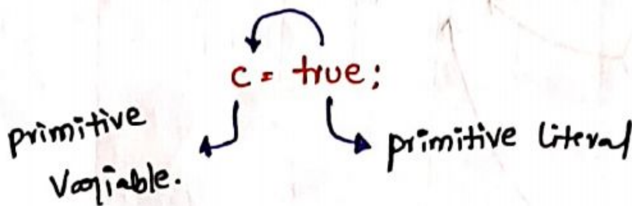
* Primitive Literals / Values are storing in a Variable that is called primitive variables.



Example:



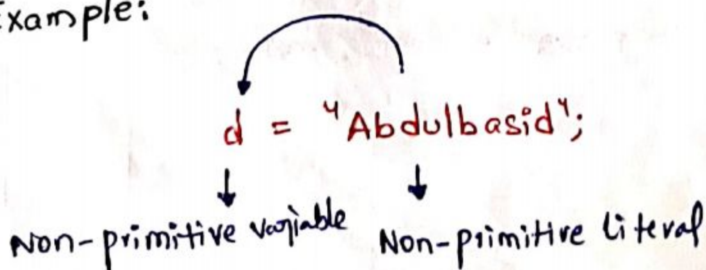




2. Non-primitive Variables:

* Non-primitive Literals or values are storing in a Variable this is called Non-Primitive Variable.

Example:



DataTypes:

* DataType are used to create variables of a specific type.
* In Java, datatypes are classified into two types.

    1. Primitive datatypes
    2. Non-primitive datatypes.

1. Primitive Data type:

* The data type which is used to create a variable to store primitive values such as numbers, characters, boolean, is known as primitive data type.

Note:

* All primitive data types are ~~known~~ keywords in java.
* primitive datatypes are predefine Data types, There are 8 types of primitive Data types.

2. Non-primitive Data types:

* The data type which is used to create a non-primitive variable to store the reference is known as a non-primitive data-type.

Note:

* Every class name in java is non-primitive datatype.
* None primitive data types are user defined data types.

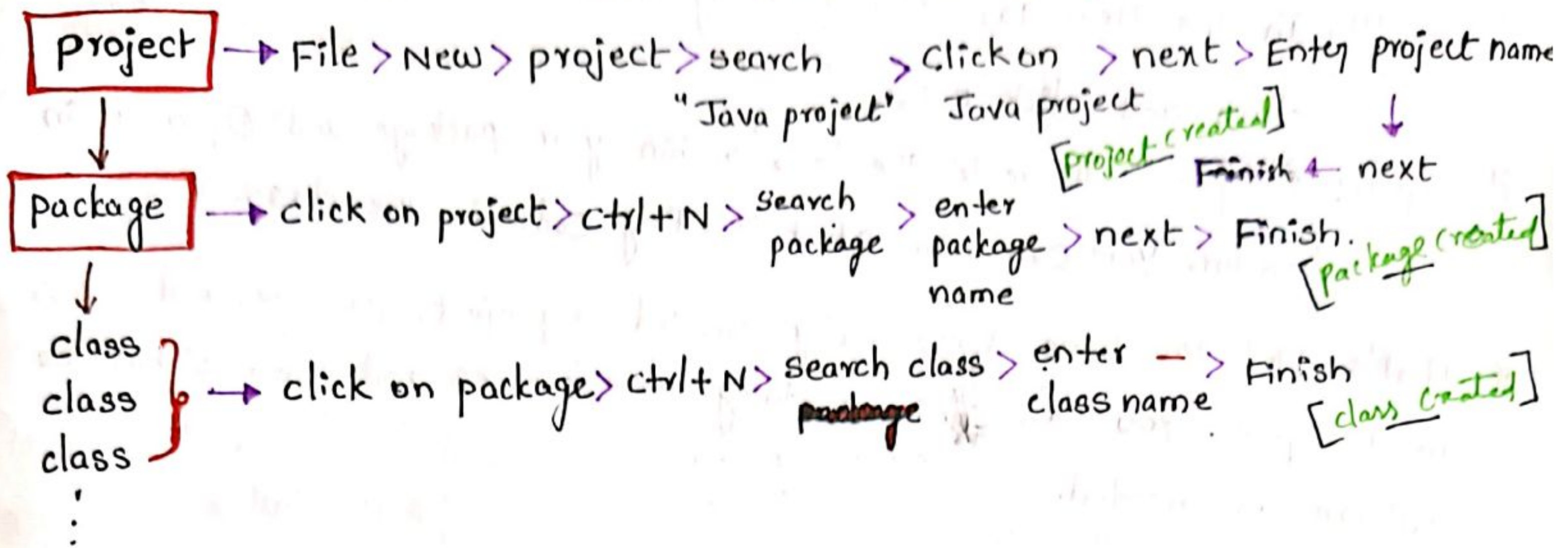
Note:

The number data type in increasing Order of the Capacity.

byte < short < int < long < float < double

Primitive Values		Primitive Data Types	Default Values	Size
Numbers	Integers (whole number) +ve to 0 to -ve	byte	0	1 byte 8 bits
		short	0	2 byte 16 Bits
		int	0	4 byte 32 bits
		long	0 l/L	8 byte 64 bits
	Floating Value	float	0.0 f/F	4 byte 32 bits
		double	0.0 d/D	8 byte 64 bits
Character -		char	/u0000	2 byte 16 bits
Boolean		boolean	False	1 bit

To create a project, package and classes in Eclipse:



Follow these steps: [This steps is written with the help of images]

#Step-01: Open Eclipse IDE.

#Step-02: Create a new Java project by selecting "File" > "New" > "Project" from the top menu.

#Step-03: In the "New Project" dialog box, type "Java" in the Search bar and Select "Java Project." Click "Next"

#Step-04: Give your project a name in the "Project name" field and click "Finish"

#Step-05: Once the project is created, you will see it listed in the "Package Explorer" view on the left side of the Eclipse window.

#Step-06: Right-click on the project name in the "package Explorer" and select "New" > "Package" from the Context menu.

#Step-07: In the New Package dialog box, enter the name of your Package (e.g: abdulbasid (or) com. csehacktech. myproject.). click "Finish"

com → domain, csehacktech → CompanyName, myproject → packageName

#Step-08: The package will now appear under your project in the "Package Explorer" view.

#Step-09: To create a class within the Package, right-click on the package name and select "New" > "class" from the context menu.

#Step-10: In the 'New Java Class' dialog box, enter the name for your class and click 'Finish'

#Step-11: Eclipse will create the class within your package and Open it in editor. You can now start writing code in the class.

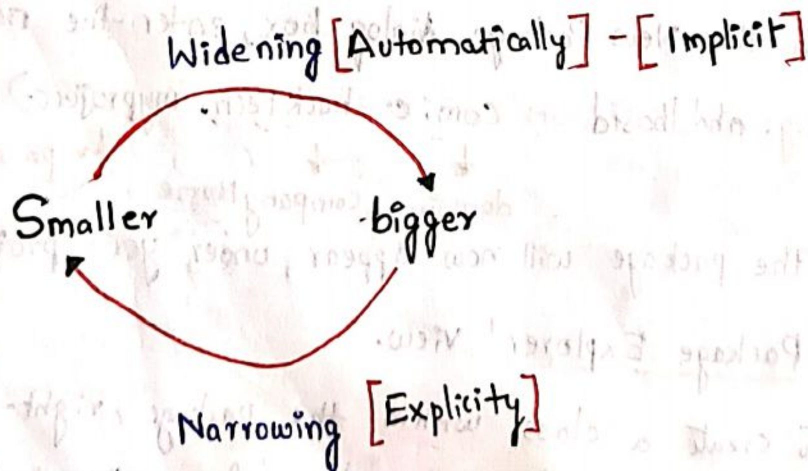
* That's It! You have successfully created a project, Package, and class in Eclipse. You can continue adding more classes and files within the package as needed.

Type Casting:

* The process of converting one type of data into another type is known as Type Casting.
* There are two types of typecasting:

1. Primitive Type casting:
   1. Widening (Implicit)
   2. Narrowing (Explicit)
2. Non-Primitive Type Casting:
   1. Upcasting (Implicit)
   2. Downcasting (Explicit)

} Part II



1. primitive Type Casting:

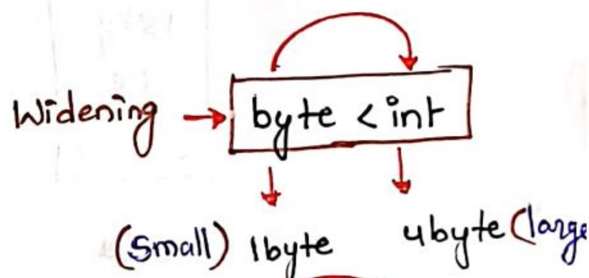
* The process of Converting one primitive value into another primitive value is known as primitive type Casting.

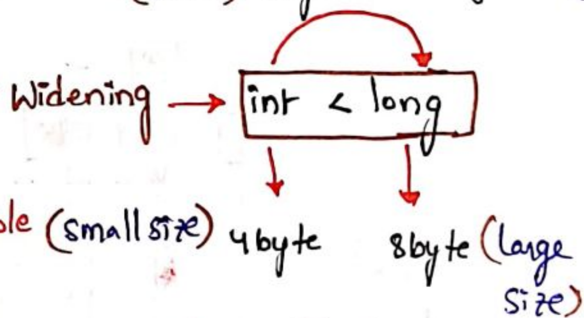
1. Widening:

* The process of converting Smaller range of primitive data type into larger range of primitive data type is called widening.
* In widening process there is no data loss.
* Since there is no data loss, compiler can Implicity do widening hence it is also known as auto widening (Implicit - Automatically)

Example:

```
class Widening
{
    public static void main(String[] args)
    { System.out.println("main Starts");
      int num = 10; // creating int Type variable.
      System.out.println(num); //10
      double num1 = num; // converting into double Type
      System.out.println(num1); //10.0
      System.out.println("main Ends");
    }
}
```





Output:

main Starts
10
10.0
main Ends

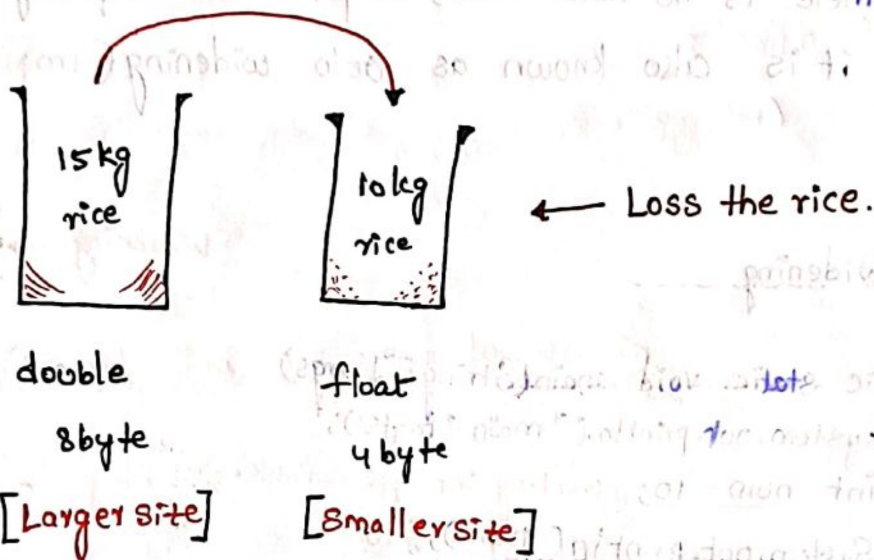
* boolean ≠ TypeCasting → Not Support
* char → int → char Convert only in int.

char → float ✗

char → long ✗

2. Narrowing :

* The process of converting larger of primitive data type into Smaller range of primitive data type is known as Narrowing.
* In narrowing process there is a possibility of data loss.
* Since there is a possibility of data loss, compiler does not do narrowing implicity.
* It can be done explicity by the programmer with the help of type cast Operator.



Example :

```
class Narrowing
{
    public static void main(String[] args)
    {   System.out.println("main Starts");
        double num = 10.98;
        System.out.println(num);
        int num1 = (int) num;
        System.out.println(num1);

        System.out.println("main Ends");
    }
}
```

Annotations:
- `int` → declares the datatype of the variable num1
- `(int)` → Narrowing TypeCast Operator "( )" pair paranthesis
- `num` → is the value that is being converted to an int.

Typecast Operator:

* It is a unary Operator (Only one operand).
* Type cast operator is used to explicity convert one datatype into another data type.

OPERATOR:

* Operator are predifined Symbol which is used to perform specific task on the given data.
* The data given as a input to the operator is known as Operand.
* Based on the number of operand Operator are futher classified into the following.

Types of operators:

1. Unary Operator
2. Binary Operator
3. Ternary Operator

1. Unary Operator:

* The Operator which can accept only one Operand is known as Unary Operator.

2. Binary Operator:

* The Operator which can accept two Operand is known as Binary Operator

3. Ternary Operator:

* The Operator which can accept three Operand is known as Ternary Operator.

Classification of Operator:

* The Operators can also be classified based on the task.

1. Arithmetic Operator ✓✓
2. Assignment Operator ✓✓
3. Relational Operator ✓✓
4. Logical Operator ✓✓
5. Increment / Decrement Operator ✓✓ (unary Operator).
6. Bitwise Operator ✓
7. Conditional Operator ✓✓
8. Miscellaneous (Conditional or Ternary Operator)

8. Shift Operator.

1. Arithmetic Operator:

* The symbols that are used to perform mathematical operations on operands.

Operator's	Name of Operator	Syntax:
+	Addition	x + y (operator: +; operands: x, y)
–	Subtraction	x-y
*	Multiplication	x*y
/	Division	x/y
%	Modulus [Remainder after division]	x%y

2. Assignment Operator:

* Assignment operators are used to assigning the value on its right to the variable on its left.

Operator	Syntax	Equivalent to
=	a=5	Assign the value
+=	a+=b	a=a+b
-=	a-=b	a=a-b
*=	a*=b	a=a*b
/=	a/=b	a=a/b
%=	a%=b	a=a%b

3. Relational Operator:

* Relational operator used to compare the values, Returns Boolean value.

Operator	Syntax	Name of the Operator
==	a==5	Is equal to
!=	a!=b	Not equal to
>	a>5	Greater than

Operator	Syntax	Name of the Operator
<	a<b	Less than
>=	a>=b	Greater than or Equal to
<=	a<=b	Less than or Equal to

4. Logical Operator:

* Logical operator is used to compare the boolean values, Returns boolean value.

operator	operations.
&& [Logical AND]	If all the expression returns true then this operator will return true.
|| [Logical OR]	If any of one of the expression return true then this operator will return true.
! [Logical NOT]	If the result is true then it will return false and vice versa.

Logical AND [&&]		
T	T	T
F	T	F
T	F	F
F	F	F

Logical OR [||]		
T	T	T
F	T	T
T	F	T
F	F	F

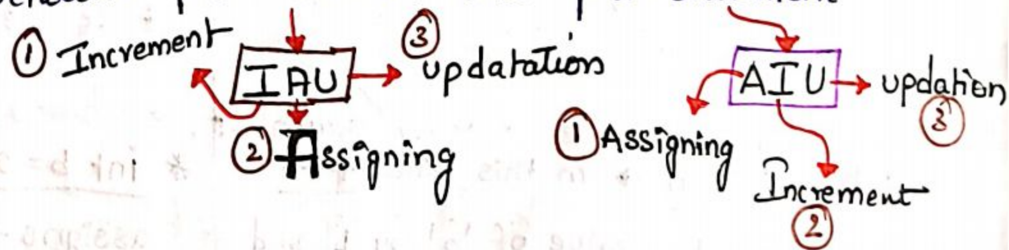
Logical NOT [!]	
T	F
F	T

5. Increment/ Decrement Operator:

* The increment and decrement operators are "unary operators" that increase or decrease the value of their operand by one.
* The increment operator is represented as the double plus (++) symbol, while the decrement operator is represented as the double minus (--) symbol.

operator	Types	Operation.
1. ++ [Increment]	++i [1. pre-increment] IAU	* Increase the value by one and use it * update
	i++ [2. post-Increment] AIU	* use the value * Increase the value by one and update it
2. -- [Decrement]	--i [1. pre-decrement] DAU	* decrease the value by one and use it. * update
	i-- [2. post-decrement] ADU	* use the value. * Decrease the value by one and update it.

Example: Difference between pre-Increment and post-Increment



Pre-Increment	Post-Increment

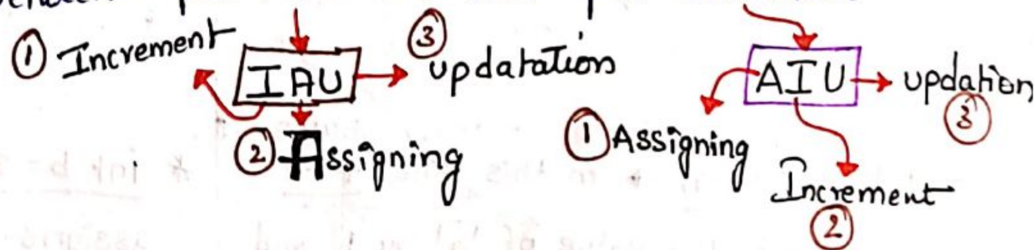
#1 ++a
int a=5;
int b=++a;
Sop(a);//6
sop(b);//6
IAU
① Increment
② Assigning
③ Updation
	#1 a++
	int a=5;
	int b=a++;
	Sop(a);//
	sop(b);//
	AIU
	① Assigning
	② Increment
	③ updation

5. Increment / Decrement Operator:

* The increment and decrement operators are "unary operators" that increase or decrease the value of their operand by one.
* The increment operator is represented as the double plus (++) symbol, while the decrement operator is represented as the double minus (--) symbol.

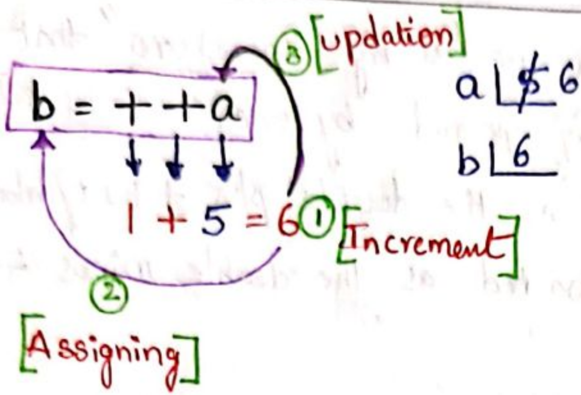
operator	Types	Operation.
1. ++ [Increment]	++i [1. pre-increment] IAU	* Increase the value by one and use it * update
	i++ [2. post-Increment] AIU	* use the value * Increase the value by one and update it
2. -- [Decrement]	--i [1. pre-decrement] DAU	* decrease the value by one and use it. * update
	i-- [2. post-decrement] ADU	* use the value. * Decrease the value by one and update it.

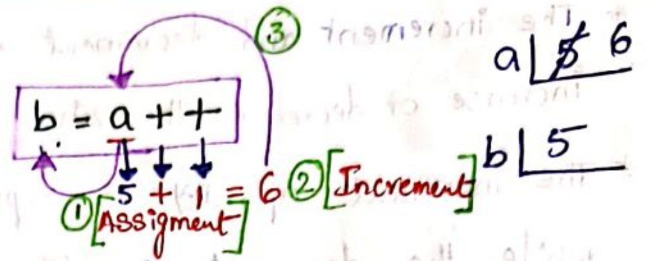
Example: Difference between pre-Increment and post-Increment



Pre-Increment	Post-Increment

#1 ++a
int a=5;
int b= ++a;
Sop(a);//6
sop(b);//6
IAU
① Increment
② Assigning
③ Updation
	#1 a++
	int a=5;
	int b = a++;
	Sop(a);//
	Sop(b);//
	AIU
	① Assigning
	② Increment
	③ updation





* First Increment the Value by one, second Assigning the value and update the value pre-Increment

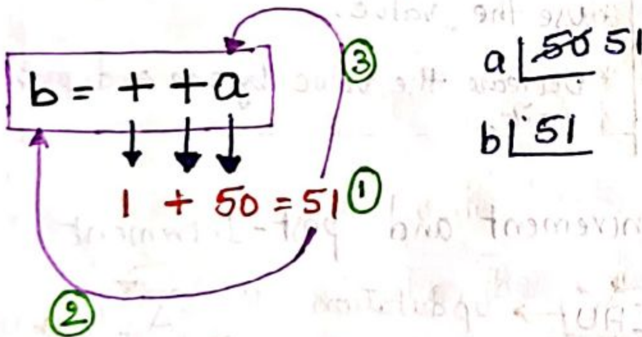
* First Assigning the value then Second Increment the Value by one and update the value. post-Increment

#2

```
int a = 50;
int b = ++a;
sop(a);//51
sop(b);//51
```

IAU

① Increment
② Assignment
③ updation

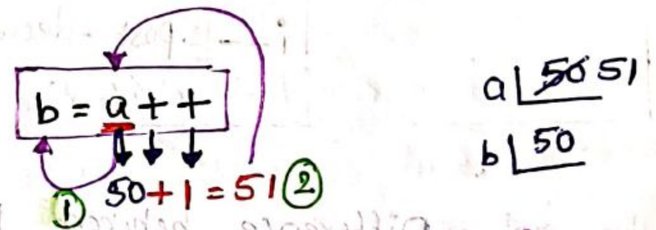


#2

```
int a = 50;
int b = a++;
sop(a);//51
sop(b);//50
```

AIU

① Assigning
② Increment
③ updation



* int b= ++a; → in this line, '++a' increment the value of 'a' by 1 and then assign the updated value to 'b'.

* So, after this line, 'a' become 51, and 'b' is also set to 51.

* int b=a++; → In this line, 'a++' assigns the current value of 'a' to 'b' and then increments the value of 'a' by 1.

* So, after this line, 'a' becomes 51 and 'b' is set to 50.

Example: Difference between pre-Decrement and post-Decrement

Pre-Decrement	post-Decrement

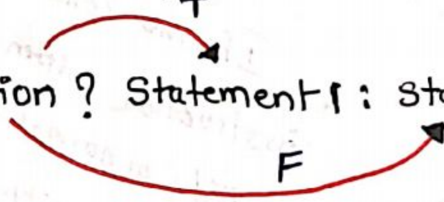
int a=5;
 int b= --a;
 sop(a);
 Sop(b);
 --a
 DAU
 ① Decrement
 ② Assigning
 ③ updation
 b= --a
 1 - 5 = 4 ① ② ③
 a ~~5~~ 4
 b 4
	int a=5
	 int b=a--;
	 sop(a);
	 sop(b);
	 a--
	 ADU
	 ① Assignment
	 ② Decrement
	 ③ updation
	 b=a--
	 ① 5 - 1 = 4 ② ③
	 a ~~5~~ 4
	 b 5

6. Conditional Operator:

* It is a ternary Operator.

Syntax: to create Conditional operator:

Operand1 ? operand2 : operand3;



Operation:

1. The return type of Operand 1 must be Boolean.
2. If the condition is true, Statement1 will get execute else Statement2 will get execute.

Decision Making statement: Conditional Statements

* In java, decision making statements allow you to control the flow of your program based on certain conditions.
* Decision making statements are also known as Conditional statements or Control Statements.

Types:

* Skip the block based on condition.

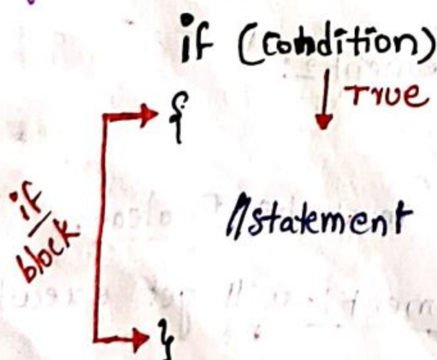
1. ① Simple If ✓
2. ② If-Else ✓
3. ③ If-elseif Ladder ✓
   ④ Nested If
4. ⑤ switch ✓

1. Simple if:

→ keyword

* The simple "if" statement checks a single condition and executes the associated code block only if the condition is true. if the condition is false, the code block is skipped and the program continues to the next statement.

Syntax: //syntax to create if Statement.



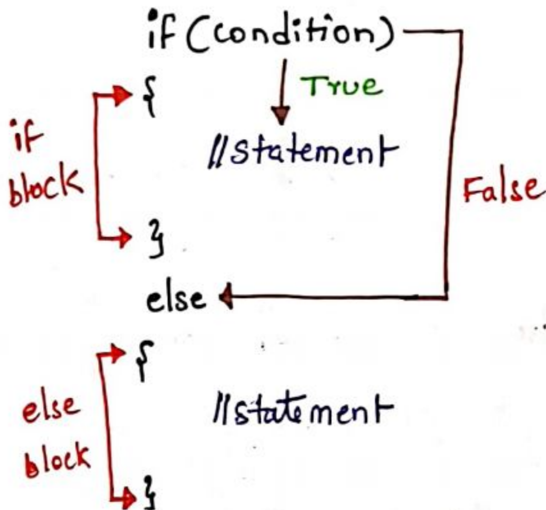
Work flow:

If the cond is satisfied then the instruction written inside the if block gets executed or normal flow of the execution continue (instructions written inside the if block is skipped).

2. if-else :

* The "if-else" statement provides an alternative code block that is executed when the condition is false. So, there are two possible outcomes based on the condition: one for true and another for false.

Syntax: //syntax to create if-else statement.



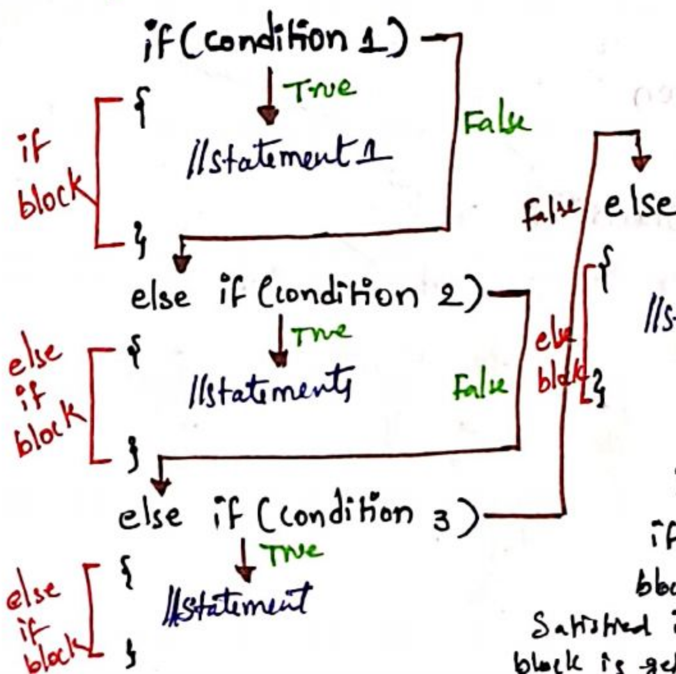
work flow:

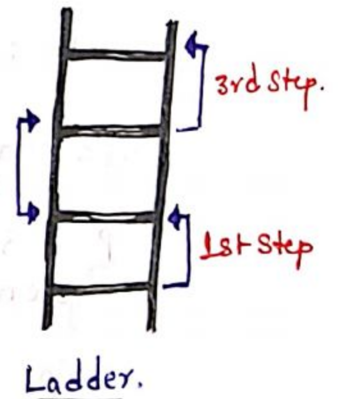
if the cond is satisfied then the instruction written inside the if block gets executed if not satisfied else block gets executed (any one of the blocks will be skipped based on a condition.)

3. if-else ladder:

* The "if-else ladder" involves a chain of "if-else" statements. Each condition is checked one after the other until a true condition is found. Only the code block corresponding to the first true condition is executed.

Syntax:





Ladder.

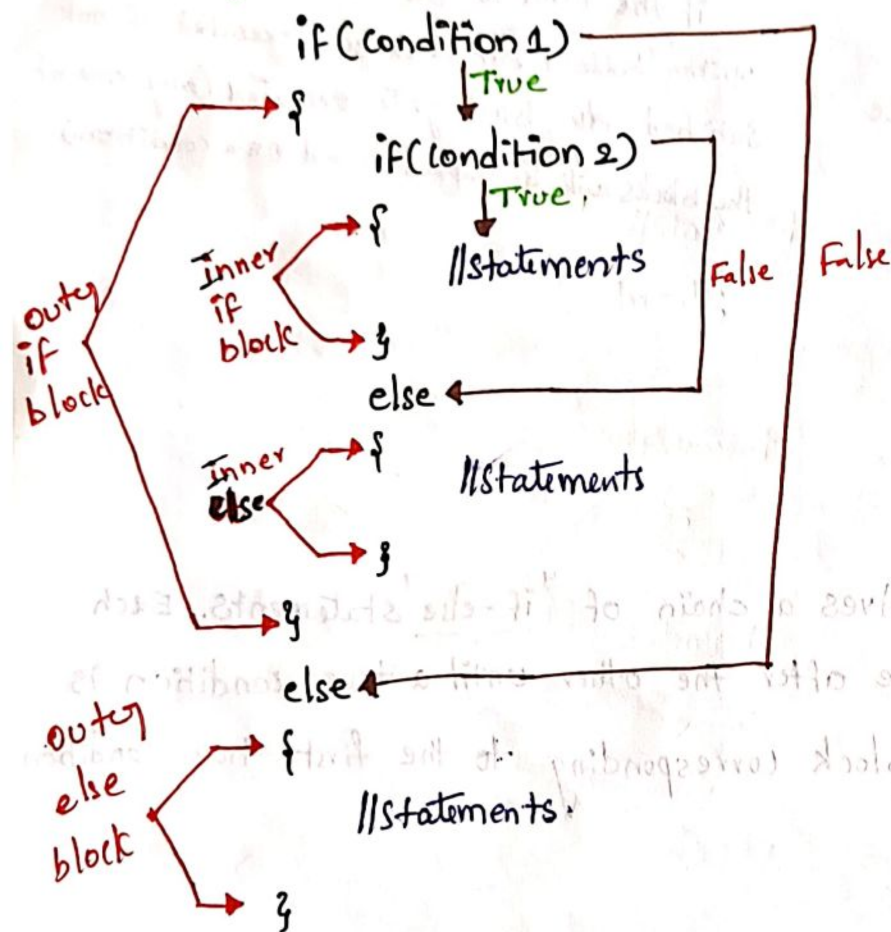
work flows

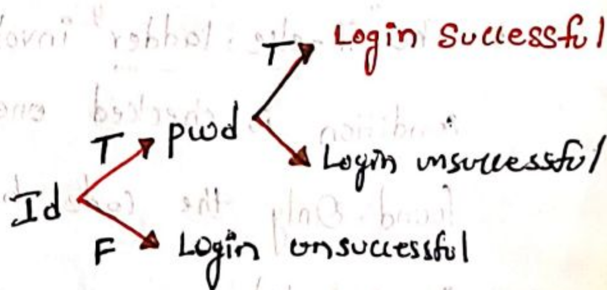
if the condition is satisfied then the instruction written inside the if block gets executed if not satisfied, the condition is checked in the else if block from top to bottom order and if the condition is satisfied in any of the else if block then, only that else if block is gets executed. If not satisfied else block gets executed blocks are skipped

4. Nested If:

* Nested If Statements involve using one "if" statement inside another if statement. This allows for checking multiple conditions and executing different blocks of code based on combination of those conditions.

Syntax:





Examples:

```
public class Gmail_verification
{ p s v m(—){
    System.out.println("Main Starts");
    String id = "admin";
    int pwd = 1234;
    if(id == "Admin")
    {
```

```
System.out.println("Id is verified");
if(pwd == 12345)
{
    System.out.println("Login successful");
}
else
{
    System.out.println("Login unsuccessful");
}
}
else
{
    System.out.println("Id is not verified");
    System.out.println("Login unsuccessful");
}
System.out.println("Main Ends");
}
}
```

5. Switch Statement:

* Instead of writing many if.. else statements, you can use the switch statement.
* The switch statement selects one of many code blocks to be executed.
* switch is a keyword.

Syntax:

```
switch(expression)        // switch -> keyword; expression -> value/variable/expression
{
   case 1: {              // case -> keyword; 1 -> value/expression
          //statement
        } break;
   case 2: {
          //statement
        } break;          // break -> keyword
   case 3: {
          //statement
        } break;
   default: {             // default -> keyword
          //statement
        }
}
```

* expression == case 1
* epression == case 2
* expression == case 3
* If three cases are Not matched with expression default will be executed.

How it works:

* The switch expression is evaluated once.
* The value of the expression is compared with the values of each case.
* If there is a match, the associated block of code is executed.
* The break and default keywords are optional.

Example:

Choice → 2 == 1 ✗ (Case)

2 == 2 ✓

```
int choice = 2;
switch (choice)
{
    Case 1 : {
              Sysdom.out.println("case 1 is executes");
            } break;

    Case 2: {
              System.out.println ("Case 2 is executes");
            } break;

    case 3: {
              System.out.println("Case 3 is executes");
            } break;

    default: {
              System.out.println( "default executes");
            }.
}
```

Note:

* For a switch we can't pass long, float, double, boolean.

Switch ( )
└→ float, double, long, boolean. ✗

* For a case we can't pass a variable.

Case
└→ Variable ✗

Break:

Break is a keyword, it is a control transfer statement.

* Break is a keyword, it is a control transfer statement.
* Break is used inside the switch and loop block.
* When the break statement is executed control is transfered outside the block.

Grouping switch statement:

```
char ch = 'A';

switch(ch)
{
    case 'a' : {
    case 'e' :      }'
    case 'i':
    case 'o':
    case 'u':
    case 'A':
    case 'E':
    case 'I':
    case 'O':
    case 'U': {
                    System.out.println(" It is a vowel ");
              } break;

    default : {
                    Syout(" It is a consonant ");
              }
}
```

Looping Statement:

* Looping statement helps the programmer to execute the set of instructions repeatedly.
* In java we have different types of loop statements; they are:

1. While loop
2. do while loop
3. For loop
4. For each / advanced for / enhanced for nested loop
   ↓
   part-3 – collections.

1. While loop:

* while is a keyword. * Looping statement

Syntax: syntax to create while loop.

```
//initialization.
while (condition)
      └→ keyword.
{
    //statements to be repeated.
    //increment / decrement
}
```

```
while (condition)
{
    //statements
}
```

Work Flow:

Case-1: when the condition is true.

* The loop continues.
* Control executes the statement while belongs to the loop.
* After execution once the loop block ends, control goes back to the condition and the entire process will be repeated till the condition becomes False.

Example:

```
while(true)
{
    //statement
    //end of the loop block
}
```

Case-2 : when the condition is false.

* The loop is stopped i.e. repetition is stopped.
* The loop block will not get executed.
* The control comes outsides the loop to the next Statement.

Example

```
while(false)
{
    //statement
}
```

2. Do-while loop:

* do is a keyword. * Looping statement

Syntax: Syntax to create do-while loop.

```
do
{
    //statement
}
while(Condition);
```

Work Flow:

Case-1: When the condition is true.

* Control goes to the loop block directly, ~~exectly~~ execute the instructions.
* The control goes to the conditions, if the condition is true the control goes back to the do block.

Case-2: When the Condition is false

* Control goes to the loop block directly, execute the instructions. Then control goes to the Condition, if the Condition is false the loop is stops and control goes to the next statment.

Difference between while and Do-while loop:

While	Do-While
* First the condition is checked, if the condition is true. then the loop block gets executed.	* In do-while, first the loop block gets executed and then the condition is checked.
* The minimum iteration can be zero.	* The minimum iteration is one.

Example:
int a=5, b=10, count=0;
while (a>b)
{
count++;
sopln("value of b is "+b);
}
sopln("Ietation "+ count);
	Example:
	int a=5, b=10, count=0;
	~~while(co~~
	do{
	count ++
	Sopln("value of b is "+b);
	}
	while(a>b);
	Sop("Iteration ."+Count);

Output: Iteration 0	Output: value of b is 10 Iteration 1

3. For loop:

* for is a keyword
* for is a looping statiment or repetative statement.
* traverse and repeat
* divide in three segments by semicolon (;).

Syntax: Syntax to create for loop.

```
for(initialization; condition; update)
{
      //statiment to be repeated.
}
```

(① initialization → ② condition → statement → ③ update → ② condition)

workflow:

Step-1: control go to the initialization part.

Step-2: Then it will go to the condition part.

Step-3: If the condition is true then it will enter inside the loop block.

Step-4: once the execution of Instruction inside the loop block is completed control will go to the update segment.

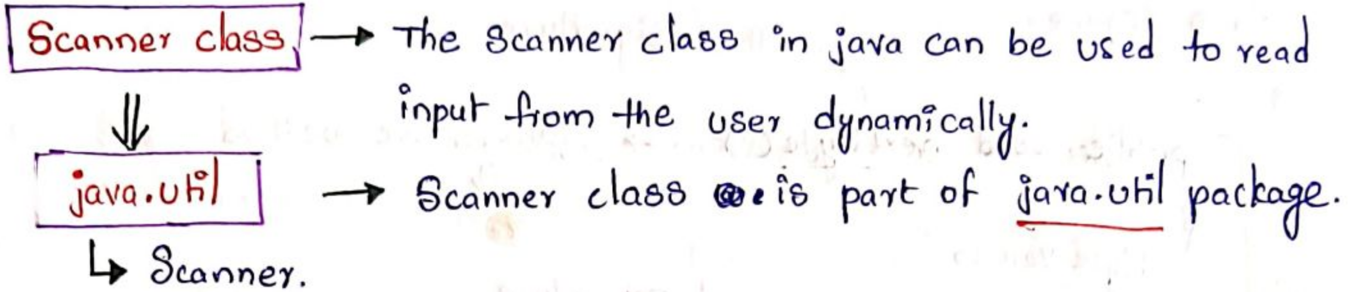
Step-5: Then it will go back to the condition. Step 1,2,3,4 will continue until the condition become false.

Note:

* All the three segments are optional (Initialization, condition, update)
* If the condition is not provided, by default it is considered as true.

Dynamic Read:

* The process of reading a data from the user during execution of a program is known as dynamic read.
* Taking values/inputs from the user.
* In java we can treat input data from user by using Scanner class.



Steps to achieve Dynamic Read:

Step 1: Import Scanner class from java.util. package.

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

↳ write before the class.

Step 2: create an Object for the child class.

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

- Scanner → Scanner class Type
- SC → Reference variable / input
- new → Keyword
- System.in → input Streams. (Standand input)
- new Scanner(System.in) → Constructor creates a new Scanner Object

Output: System.out
- System → built-in class
- out → Variable out in System class

input: System.in
- System → built-in class
- in → Variable in in System class

Step 3: Call the method of Scanner class to read the data from the user

Sc.method();
↓
input.

```
class Scanner
{
    public void nextByte()   → Non-static method.
    {
      Byte values
    }
    public void nextShort()  → Non-static Method
    {
      Short Values
    }
    public void nextInt()    → Non-static Method
    {
      Int values
    }
    public void nextLong()   → Non-static Method.
    {
      Long values.
    }
    P    V. nextFloat()
    {
       .
    }  .
    P    V : nextDouble()
    {
    }
}
```

(nextByte, nextShort, nextInt, nextLong → method Signature)

Non-static ⇓ Object

Methods in a Scanner class:

Type of Data	Method Signature.
byte	nextByte()
Short	nextShort()
int	nextInt()
long	nextLong()
float	nextFloat()
double	nextDouble()
boolean	nextBoolean()
char	next().charAt(0)
String (Single word)	next()
String (Multi word)	nextLine()

Scanner class:

* Scanner class is part of java.util package.
* Scanner class we didn't created, java developers only Created.
* In Scanner class all the methods are non-static methods.

Example:

```
import java.util.Scanner; //1. creating Scanner class

public class DynamicReading
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in); //2. creating Object
        System.out.println("Main Starts");

        System.out.println("Enter the integer value: ");
        int num1 = sc.nextInt(); //3. call the methods
        System.out.println(num1+num1);

        System.out.println("Main ends");
    }
}
```

Output:

```
Main starts
Enter the integer value:
10
20
Main ends.
```

Scope of the Variable:

* Scope of a variable is the part of the program where the Variable is accessible.

Types: Three types of variables in java are:

1. Local Variable
2. Static Variable
3. Non-static variable — #oops - 2nd part

1. Local Variable:

* A Variable which is declared or Created inside the method those Variables is called "Local Variable".

* They are used to store temporary values that are only accessible within the block in which they are declared.

*

Example:

```
class LocalVariable
{
    public static void main(String[] args)
    {
        int a = 10; //Local variable
        System.out.println(a);
    }
}
```

2. Static variable:

* Static variables are declared using the static keyword and belong to the class rather than to any specific instance (object) of the class.

*

Characteristics:

1. Static variable reinitilization inside main() is possible.
2. Static variable can be declared with default value.
3. Local variable can't be declared without Initilization.

   ↳ Compile Time Error.

4. If Local variable and static variable are having same names (always Local variables will executes)

5. If we want static variable.

   (className.staticVariableName;)

Example-01:

```
public class Variable
{
    static int a = 10;
    public static void main(String[] args)
    {
        Sop("main starts");
        sop("Before ReInitialization : "+a);
        a=20;
        Sop(" After reInitialization : "+a);
        sop("main ends");
    }
}
```

Example-02:

```
public class Variable
{
    Static int a;
    public static void main(String[] args)
    {
        sop("main starts");
        Sop(a);
        sop("main ends");
    }
}
```

Example-03:

```
public class Variable
{
    public static void main(String[] args)
    {
        Sop("main starts");
        int a;
        Sop(a); // Compile Time Error
        Sop("main ends");
    }
}
```

Example-04:

```
public class Variable {
    static int a=10;
    public static void main(String[] args) {
        sop("main starts");
        int a=20;
        sop(a);
        Sop("mainends");
    }
}
```

Example-05:

```
public class Variable
{
    static int a = 10;
    public static void main(String[] args)
    {
        System.out.println("main starts");
        int a = 20;
        sop("Local variable : "+a);
        sop(" static variable : "+Variable.a);
        sop("main ends");
    }
}
```

Example : Local variable and static variable.

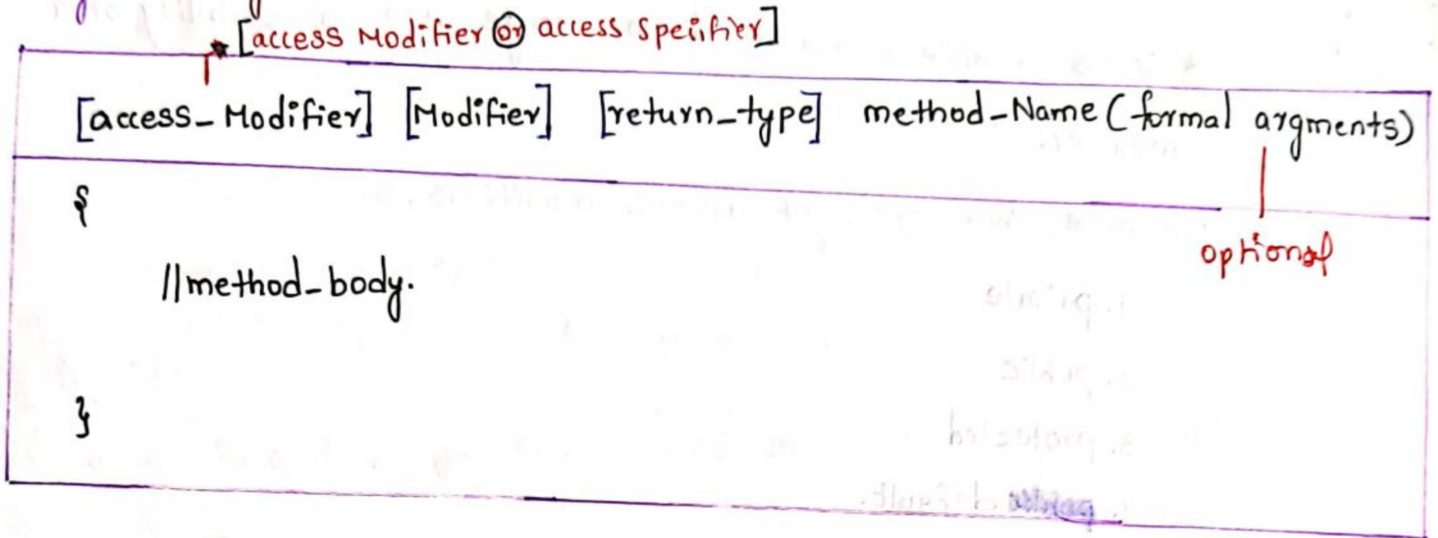
```
public class Variable
{
    static int b = 10; //static variable
    public static void main(String[] args)
    {
        sop("main starts");
        int a = 20; // Local variable.
        sop(a);
        sop(b);
        sop("main ends");
    }
}
```

Methods:

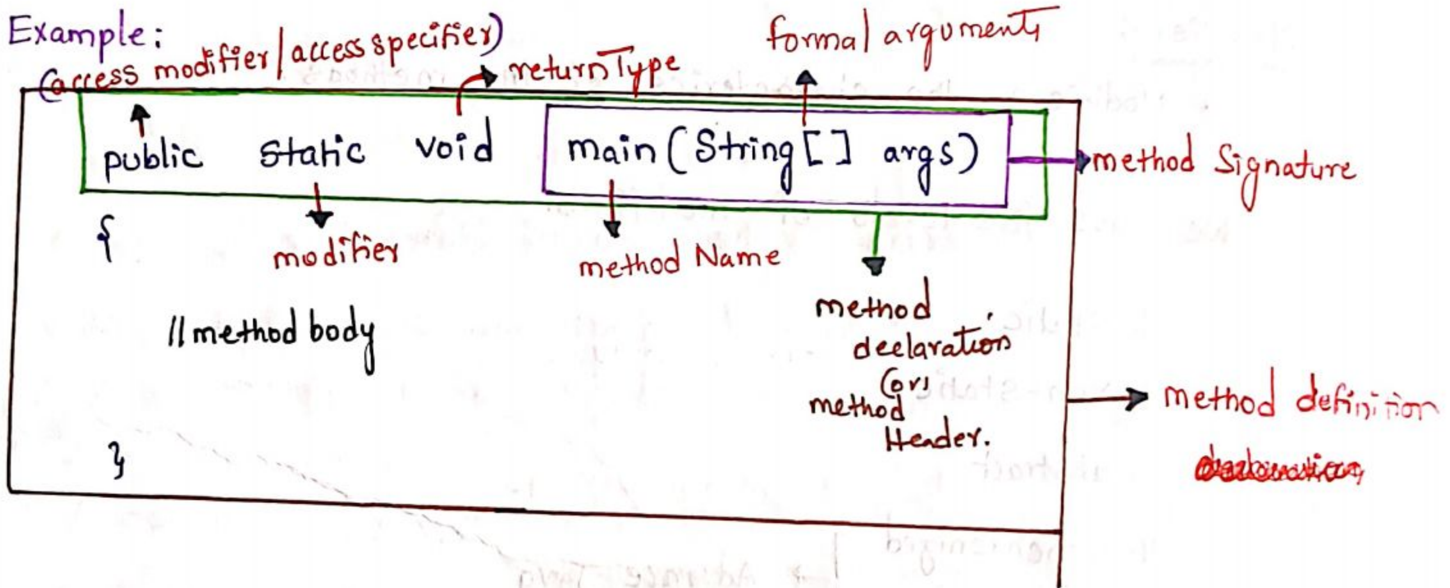
*Part-2

* It is a block of Instructions that is used to perform a specific task.

Syntax: syntax to define a method.



Example:



1. **Method Signature:**
    1. method name
    2. formal arguments:
2. **method declaration:**
    1. Access Modifier/Access Specifier
    2. modifier
    3. return Type
    4. method signature.
3. **method definition:**
    1. method declaration
    2. method body / implementation /block.

Terminology:

Access Modifiers:

* Access modifiers are also known as Access specifier.
* Access modifiers are used to change the accessibility of a member.

We have four levels of access modifiers.

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

Modifiers:

* Modifiers the characterics of the methods.

We have five levels of modifiers.

1. Static
2. Non-static
3. abstract
4. Synchronized] → Advance Java
5. volatile]
6. final.

Return Type:

* the method after execution can return a value back to caller.
* Therefore it is mandatory to specify what type of data is returned by the method in the method declaration. statement, this is done with the help of return type.

Return type definition:

* The return type is a data type that specifies what type of data is returned by the method after execution.

A method can have the following return type.

1. Void ✓
2. primitive data type ✓
3. Non-primitive type. ✓

1. void:

* void is a data type that is used as a return type when the method returns nothing.
* It is a keyword in java.

NOTE:

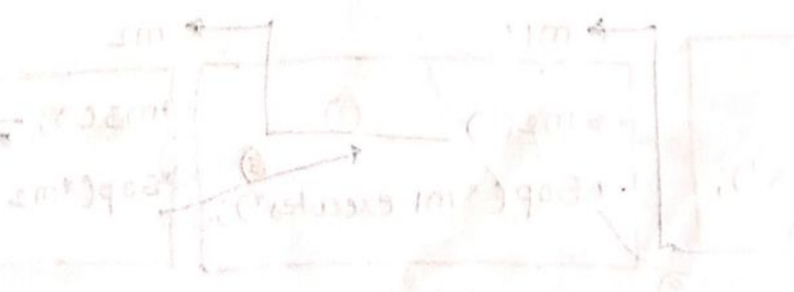
* A method can't create inside another method → Compile Time Error.
* A class can have any number of methods.

Note:

* A method will get executed only when it is called, we can call a method with the help of a method call statement.

Method Name:

identifier



arguments

[Optional]

Method Call:

Method Call Statement:

* The statment which is used to call a method is known as a method call statment.

Syntax:

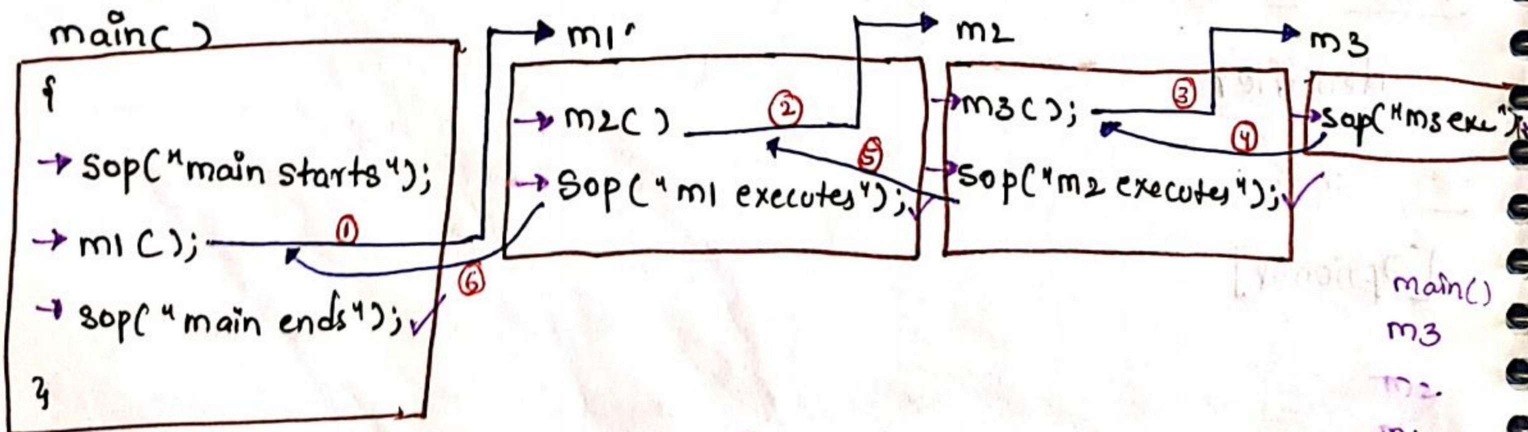
```
methodName([Actual argument]);
```

* we can call a no-argument method without passing an actual argument in the method call statement.

Method Call Flow:

Method Call Statement Flow:

* Execution of calling method is paused.
* Control is transferred to the called method.
* Execution of called method begins.
* Once the Execution of the called method is completed the control is transferred back to the calling method.
* Execution of calling method resumes.



* We can call method inside another method.

Calling Method:

* The method which is trying to call another method is known as the calling method (Caller).

Called Method:

* The method which is being called by the caller is known as a called method.

Example:

(1)

```
add(10); //method call statement          p s v add(int a)
                                          {
                                              Sop(a); //called method or method definition
                                          }
```

- 10 → actual arguments
- int a → Formal arguments
- * Calling method.
- * being call

(2)

```
Sub(50, 30); //calling method             p s v Sub(int a, int b)
                                          {
                                              Sop(a-b); //20
                                          }
```

- Sub(...) → method call statement
- 50, 30 → Actual argument
- int a, int b → Formal argument
- → Called method

main method: main()

* The execution of a java program always starts from the main method defined as follows.

main method

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

}
```

Purpose of the main method:

* Start the execution.
* Control the flow of the execution.
* End of execution.

Note:

* A method can be executed only when it is called, we can call a method any number of times therefore it is said to be code reusability.
* The main method is always called by JVM.

Types of methods:

* Based on number of arguments, methods can be classified into two types:

    1. NO argument method
    2. parameterized method.

1. NO argument method:

* A method without formal arguments is known as NO argument method.

Example:

```
public static void demo()   → No argument
                            ↳ Formal arguments
{
    System.out.println("This is demo() method");
}
```

2. Parameterized method:

* The method which has formal arguments is known as parameterized method.
* Used to accept the data

Example:

```
public static void demo(int n)
{
    System.out.println("This is demo() method");
    System.out.println(n); // prints n value passed in calling method.
}
```

Examples:

No-argument Method.

hash( ) → Method call statement

(actual No arguments)

```
p s v harsh( )     // No argument
{
  sop("Addition Starts");
}
```

* Called method

Parameterized method.

harsh("developer"); → Method call statement

("developer" → actual argument)

```
p s v harsh(String abdul)     // formal argument
{
  sop(a);
}
```

* called method.

Requirements:

1. Get a value from the method call statement
2. add the value.
3. Return the to result to method call statement.
4. print it main().

Example:

```
public class Method4Req
{
    public static void main(String[] args)
    {
        System.out.println("Main starts");
        int res1 = add(10,30);
        System.out.println("Addition : " +res1);

        int res2 = sub(10,20);
        Sop(" Substraction : " +res2);

        int res3 = mul(10,20);
        Sop("multiplication: " + res3);

        int res4 = div(10,20);
        Sop("division : " +res4);

        Sop("Main ends");
    }

    public static int add(int a, int a)
    {
        int c = a+b; // 10+20 = 30
        return c; //30
    }

    public static int sub( int a, int b)
    {
        int c= a-b; //10-20 = -10
        return c; //-10
    }

    public static int mul(int a, int b)
    {
        int c = a*b; //10*20 = 200
        return c; //200
    }
```

```
public static int div(int a, int b)
{
    int c=a/b; //10/20 = 0
    return c; //0
}
}
```

Output:

```
Main starts
Addition: 30
Substraction: -10
multiplication: 200
division: 0
Main ends
```

Formal and Actual Arguments.

* A variable which is declared in a method declaration is known as formal arguments.

Example:

```
public static void demo(int n, int y)
{                              ↓
    Sop(n);                Formal arguments.
    Sop(y);
}
```

Method Overloading:

Actual arguments:

* The values passed in a method call statement is known as actual arguments.

Example:

demo(10,20);
↓
actual arguments.

*— [method overloading is their in after the page]

Rule:

1. Should a change in formal arguments.

2. Should a change in data type of formal arguments.

3. Should a change order/sequence of datatype

Example:

①
```
m1(int a)
{

}
m1(int a, int b)
{

}
m1(int a, int b, int c)
{

}
```

②
```
m1(int a)
{
}
m1(double a)
{
}
m1(String a)
{
}
```

③
```
m1(int a)
{
}
m1(int a, String b)
{
}
m1(String a, int b)
{
}
```

Method overloading:

* multiple methods with same name but different formal arguments.

*— [Rule's formal arguments is their in before page]

Rule to call the parameterized method:

* Number of actual arguments must be same as the number of formal arguments.

* The type of corresponding actual arguments should be same as type of formal arguments, if not compiler tries implicit conversion; if conversion is not possible we get compile time error.

Example:

```
class call
{
    public static void add(int x, int y)
    {
        int sum;
        sum = x+y;
        System.out.println(sum) // 30 as output
    }
    public static void main(String[] args)
    {
        add(10,20); // call by value
        int a =10;
        int b= 20;
        add(a,b); // call by reference
        add(45.54,54.56); // CTE as float to int conversion cannot
                                happen automatically.
    }
}
```

Return type

* In Java, every method is declared with a return type such as int, float, double, String, etc.
* These return types required a return statement at the end of the method. A return keyword is used for returning the resulted value.
* The void return type doesn't require any return statement. If we try to return a value from a void method, the compiler shows an error (instead if return type is void use print statement).

Syntax: return returnValue;

Return keyword:

Following are the important points must remember while returning a value.

* The return type of the method and type of data returned at the end of the method should be of the same type. For example, if a method is declared with the float return type, the value returned should be of float type only.
* The variable that stores the returned value after the method is called should be a similar data type otherwise, the data might get lost.
* If a method is declared with parameters, the sequence of the parameter must be the same while declaration and method call.

Example: return keyword Example.

```
class Sum-Return-eg
{
    public static int sum(int a , int b)
    {
        int sum = a+b;
        return sum;
    }

    public static void main(String[] args)
    {
        int a =10;
        int b =20;
        int sum = Sum(a,b); //store the return value from Sum(a,b)
                              in int type.

        System.out.println(sum);

        /*
        System.out.println(Sum(a,b)); → Can also call the
        returning methods inside a print statement.
        */
    }
}
```

Example: Method with void Return type.

```
class Print_Number
{
    public static void Print_Number(int n) // Parameterized methods
    {
        System.out.println(n);
    }
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        int n = 20; // call by reference
        Print_Number(n);
        Print_Number(50); // call by Value.
    }
}
```

Formal arguments

a → reference Variable

| 20 |

Sum(a,b)

Note: void return type example (for void return type no need to use return keyword)

Difference between Return-type and Return keyword.

Return type	Return keyword.
* Data type which Specifies what type of data is returned after execution of a method.	* return keyword is a Control transfer Statement which transfers the control from the called method to the Calling method.
* The return-type specified in the method header and the ultimate value returned must be same.	* It will ultimately return the value after execution to the calling method.

oops - [Object Oriented Programming]

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

}

1. Static members

   (or)

2. Non-static members.

Static keyword:

* Static is a keyword.
* It is a modifier.
* Any member of the class is prefixed with Static modifier then it is known as static member of a class.

① Static members:

1. Static variable:
2. Static methods:
3. Static Initializers:

1. Static Variable:

* variable declare in a class block and prefexed with static modifier is known as Static variable.

Characteristics:

* It is a member of a class.
* It may be assigned with default value.
* Memory will be allocated inside the class static area.
* It is global in nature. It can be used within the class as well as in different class (using class-name).
* We can use the static variable from different class using class-name

2. Static method:

* A method prefixed with static keyword is known as static method.

Characteristics:

* Static method block is stored in the method area and reference of the static method is stored in the class static area.
* We can use static method with or without creating Object of the class.
* We can use static method with the help of class name.
* A static method of the class can be used in any class with the help of class name.

Example:

```
class Abdul
{
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        demo();
    }
    public static void demo()
    {
        System.out.println("This is demo()");
    }
}
```

3. Static Initializers:

* Static Initializers are used to execute the startup instructions.
* We have two types of static Initializers.

1. Single Line static Initializers
2. Multi Line Static Initializers

1. Single Line static Initializers:

* Static variable is also known as Single Line static Initializers.

Syntax:

```
Static data-type Variable_name = value;
```

Example:

```
Static int a = 10;
```

↪ single Line static initializers.

2. multi Line Static Initializers:

Syntax:

```
Static      ← Keyword.
{
    //statiments;
}
```

Characteristics:

* Static initializers gets executed implicity during the loading process of the class.
* A class can have more than one static initializers, the execute in top to bottom order

purpose of the static initializers:

* Static initializers are used to execute start-up instructions.

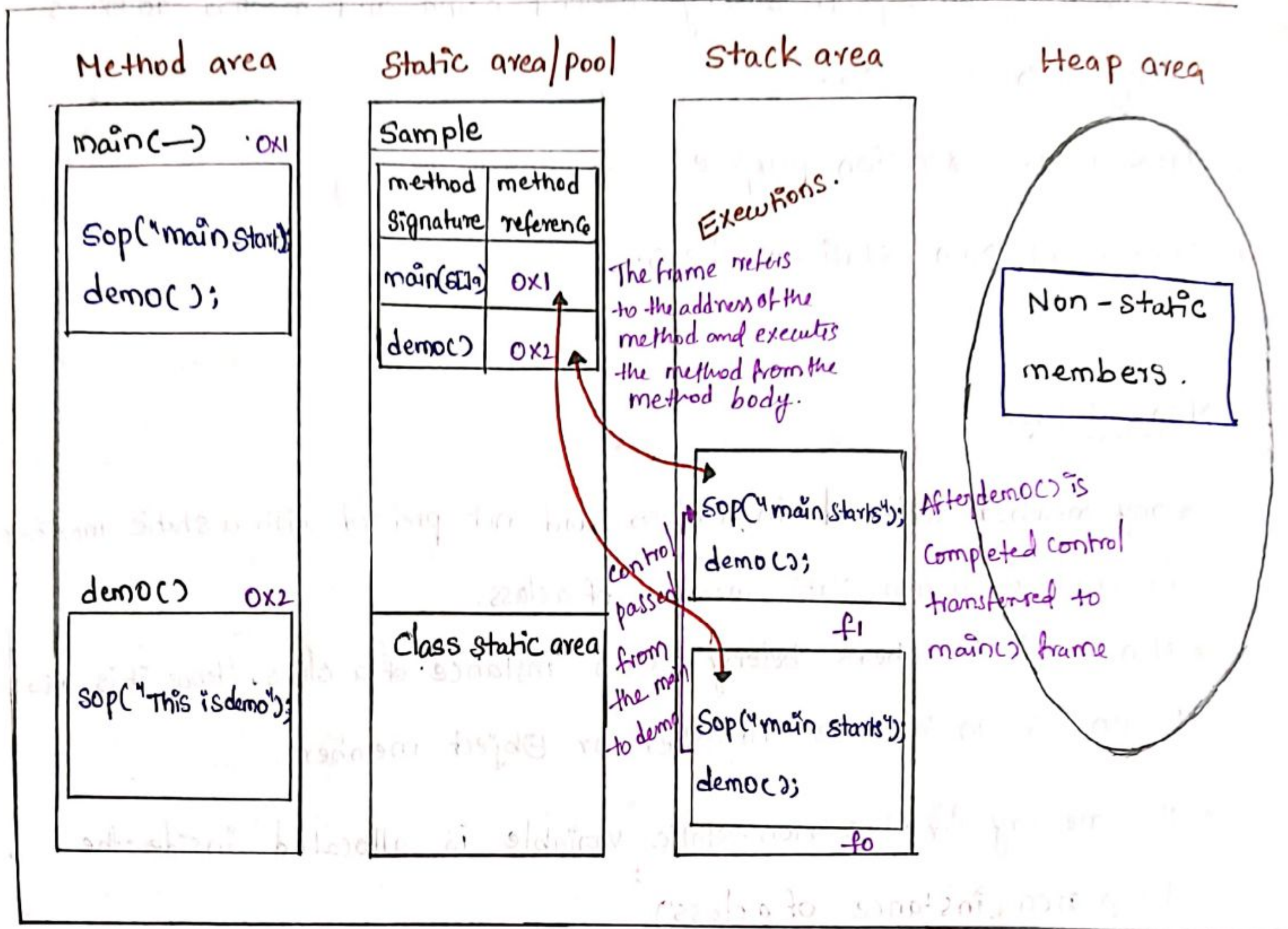
[remember ATM example..]

* static blocks gets executed before the actual execution.

Example:

```
class Static_Example
{
   static //multi line static initializer.
   {
      System.out.println("MultiLine static init 1"); ①
   }
   static //multiline static initializers.
   {
      System.out.println(" multi Line Static init 2"); ②
   }
   Static int b = 1; //single line static initializer
   Static boolean bl; //single Line Static initializer.
   public static void main(String[] args)
   {
      System.out.println(bl); // bl initialised to default false ③
   }
}
```

class Loading process:



* A block is created for a class in the static pool with the class name.
* All the method definition are loaded in the method area and if the method is static then the reference of that block is stored inside the class static area.
* If the class has any static variable they are loaded in the class static area with default values.
* If the class has any static initializers they are executed from top to bottom Order.
* After class load completes, JVM calls the main method of the initial loading class.

* Method area: Storing methods
* Static area/static pool: Storing method name and method address along with variables.
* Stack area: Execution purpose
* Heap area: Non-static members.

Non-static:

* any member declared in a class and not prefixed with a static modifier is known as a non-static member of a class.
* Non-static members belong to an instance of a class. Hence it is also known as an instance member or Object member.
* The memory for the non-static variable is allocated inside the heap area (instance of a class).
* We can create any number of instances for a class.
* Non-static members will be allocated in every instance of a class.

Non-Static Members:

1. Non-static Variable
2. Non-static method
3. Non-static Initializers
4. Constructors.

1. Non-Static Variable:

* A variable declared inside a class block and not prefixed with a static modifier is known as a "non-static variable".

Characteristics:

* we can't use the non-static variable without creating an Object.
* We can only use the non-static variable with the help of Object reference.
* Non-static variables are assigned with default during the Object loading process.
* multiple copies of non-static variables will be created (Once for every Object).

2. Non-static method:

* A method declared in a class block and not prefixed with a static modifier is known as a "non-static method".

characteristics:

* A method block will get stored inside the method area and a reference of the method is stored inside the instance of a class [object].
* We can't call the non-static method of a class without creating an instance of a class [object].
* We can't access the non-static method directly with the help of class names.
* The non-static method can't be accessed directly with their names inside the static context.

Non-static Context:

* The block which belongs to the non-static method and multi-line non-static initializer is known as non-static context.
* Inside a non-static context, we can use static and non-static members of the same class directly by using its name.

Example:

```
public class NonStaticMem
{
    int a=10; // Non-static Variable
    public static void main(String[] args)
    {
        NonStaticMem d= new NonStaticMem(); // Object Created.
        System.out.println("main starts");
        System.out.println(d.a);
        d.m1();
        System.out.println("main ends");
    }
    //Non-static Initializer.
    {
        System.out.println("non static initializer 1");
    }

    public void m1() //Non-static method.
    {
        System.out.println(" Non static method");
    }
}
```

Output:

Non static initializer ✓
main starts ✓
10 ✓
Non static method ✓
main ends. ✓

Non-Static Initializers:

* Non-static initializers will execute during the loading process of an Object.
* Non-static initializers will execute once for every instance of a class created. [object created].

purpose of non-static initializers:

* Non static initializers are used to execute the startup instructions for an Object.

Types of non-static Initializers:

1. Single line non-static initializer
2. Multi line non-static initializer

1. Single Line Non-static Initializer:

* Syntax: to create Single line non static Initializers:

dataType variableName = value;

* Non static variable also known as Single line Non static Initializers.

2. Multi Line Non static initializers:

Syntax: Syntax to create multi line Non static initializers.

```
{
    //statements;
}
```

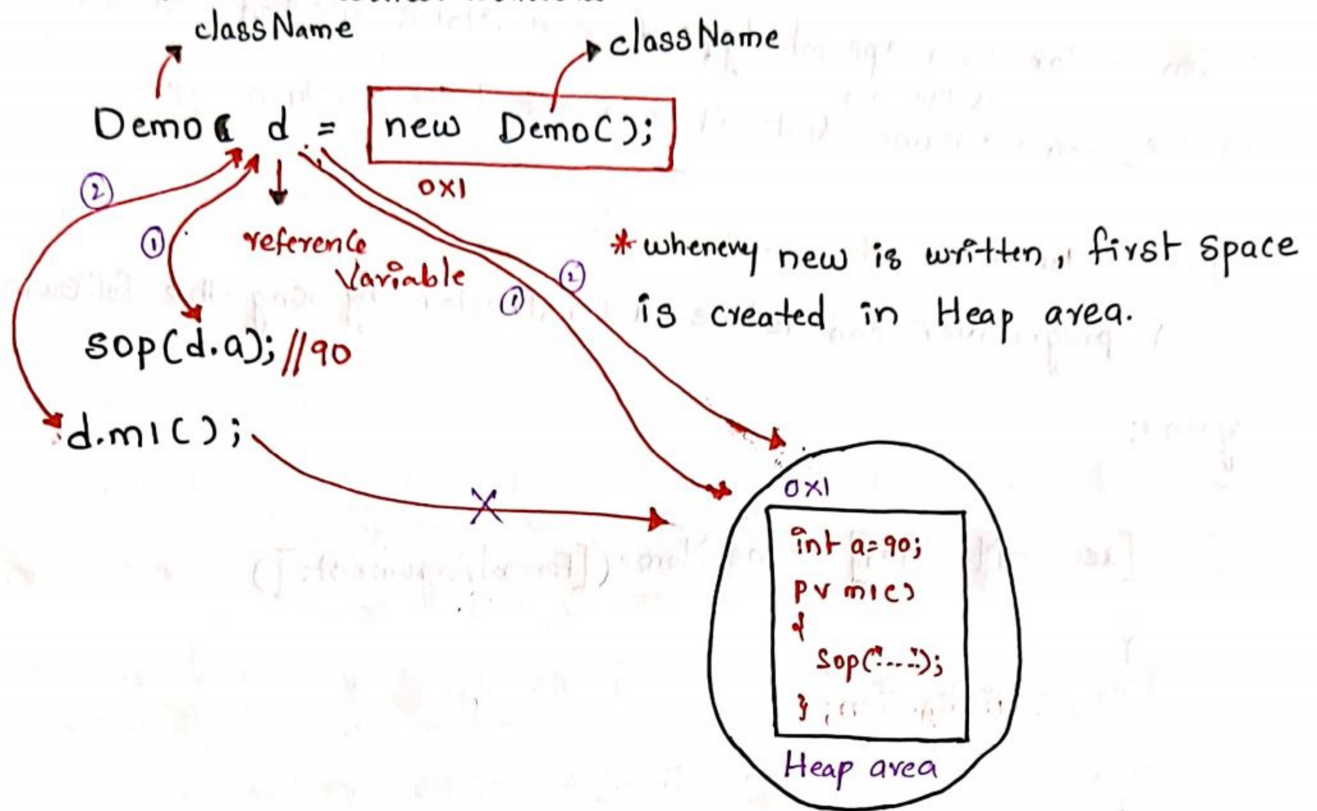
Note: All the non-static initializers will execute from top to bottom order for every Object creation.

difference between static members and non-static member.

Static member	Non-static member

static int a =10;
└→ Static Variable
 	
public static void m1()
{ ↓
//static method
}
 	
static → Static Initializer
{
→ directly
→ Object
}
	int a =10;
	└→ Non-static Variable
	
	public void m1()
	{
	// Non-static method
	}
	
	{ → Non-static Initializers
	→ Objects
	}

Creation of Non-static method:



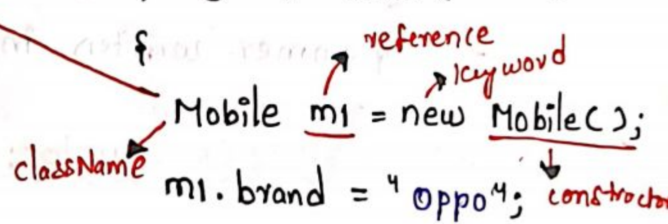
* All the Non-static member of Demo will be loaded.

Example:

```
public class Mobile
{
    String brand;
    int price;

    public void MobileDetails( )
    {
        sop(" Mobile brand : " +brand);
        sop(" Mobile price: " +price);
    }
}
```

```
public class MobileDriver
{
    P S V main (——)
    {
        Mobile m1 = new Mobile();
        m1.brand = "oppo";
        m1.price = 30000;
        m1.mobileDetails();
    }
}
```



Constructor:

* Constructor is a special type of non-static method whose name is the same as the class name but it does not have a return type.

Syntax: to create the constructor.

A programmer can define a constructor by using the following

Syntax:

```
[accessModifier] className([FormalArguments])
{
    //initialization;

}
```

Constructor body:

A constructor body will have the following things:

* Load Instructions added by the compiler during compile time.
* Non static initializers of the class.
* programmer written instructions.

purpose of the constructor:

During the execution of the constructor.

* Non static members of the class will be loaded into the object.
* If there is a non-static initializer in the class they start executing from top to bottom order.
* programmer written instruction of the constructor gets executed.

Note:

* If the programmer fails to create a constructor then the compiler will add a default constructor.

Classification of Constructor: [Types]

* Constructors can be classified into two types based on the formal argument.

1. No argument constructor
2. Parameterized constructor.

1. No Argument Constructor:

* A Constructor which doesn't have a formal argument is known as a no-argument Constructor.
* No formal argument
* default Constructor (Implicity)

↳ Employee e1 = new Employee( );

2. Parameterized Constructor:

* A Constructor which has a formal argument is known as parameterized Constructor.
* Values are passed
* programmer has to create it (Explicity)

↳ Employee e2 = new Employee("Abdulbasid", 111);

this:

* It is a keyword.
* It is a non-static variable it holds the reference of a current executing Object.

uses of this:

* used to access the members of the current Object.
* It is used to give the reference of the current Object.
* Reference of a current Object can be passed from the method using the 'this' keyword.
* Calling a Constructor of the same class is achieved with the help of this call statement.

1. No argument Constructor:

Syntax:

```
accessModifier className( )
{
  //code;
}
```

Note: If the programmer fails to create a Constructor then the compiler implicitly adds a no-argument Constructor only.

Loading process of an Object:

* a new keyword will create a block of memory in a heap area.
* Constructor is called.
* During the execution of the Constructor.
  1. All the non-static members of the class are loaded into the Object.
  2. If there are non-static initializers they are executed from top to bottom Order.
  3. Programmer written instruction of the constructors will be executed.
* The execution of the constructor is completed.
* The Object is created successfully.
* The reference of an Object is returned by the new keyword.
* These steps are reapeated for every Object creation.

Parameterized Constructor:

* The Constructor which has a formal arguement is known as parameterized constructor.

purpose of the parameterized Constructor:

* parameterized constructors are used to initialize the variables (non-static) by accepting the data from the Constructor in the Object creation statement.

Example:

```
public class Person
{       //attributes
    String name;
    int age;

    public Person( ) //NO-argument Constructor -[Implity]
    {

    }

    public Person(String name, int age) //parameterized constructor -
    {                                                    [Explicity]
        this.name = name;
        this.age =age;
    }

    public static void main(String[] args) //main.
    {     //Object creations.
        Person person1 = new Person( );
        Person person2 = new Person("Alice", 25);

        System.out.println("Name: " + person1.name + ", Age = " + person1.age);
        System.out.println("Name: " + person2.name + ", Age = " + person2.age
    }
}
```

Output:

Name: null, Age = 0

Name: Alice, Age = 25

Constructor Overloading:

* If a class is having more than one constructor it is known as constructor Overloading. but diff

Rule:

The Signature of the constructor must be different.

* multiple constructor with same name but different formal arguments.

Example:

```
public class Pen
{
    String brand;
    int price;
    boolean refill;

    pen( )
    {

    }
    pen(String brand)
    {
        this.brand = brand;
    }
    Pen(String brand, int price)
    {
        this.brand = brand;
        this.price = price;
    }
    Pen(String brand, int price, boolean refill)
    {
        this.brand = brand;
        this.price = price;
        this.refill = refill;
    }
```

```
    public void PenDetails()
    {
        System.out.println(brand);
        System.out.println(price);
        System.out.println(refill);
    }
}

public class Pendrive
{   public static void main(String[] args){
    Pen P1 = new Pen();

    P1.PenDetails();
    System.out.println("--------");

    Pen P2 = new Pen("Rorito", 20, true);

  }
}
```

Output:

```
null
0
false
--------
Rorito
20
true
```

Constructor chaining :

* A Constructor calling another Constructor is known as Constructor chaining.
* In java, we can achieve constructor chaining by using two ways

  1. this( ) → (this call statement)
  2. super( ) → (Super call Statement)

1. this( )

* It is used to call the construstor of the same class from another constructor.

Rule:

* this( ) can be used only inside the constructor.
* It should always be the first statement in the Constructor.
* The recursive call to the Constructor is not allowed (Calling by itself)
* If a class has n Constructors we can use this statement in n-1 Constructors only(at least a constructor should be without this( ))

Note:

* If the Constructor has this( ) statement then the compiler doesn't add load instruction & non-static initializers into the constructor body

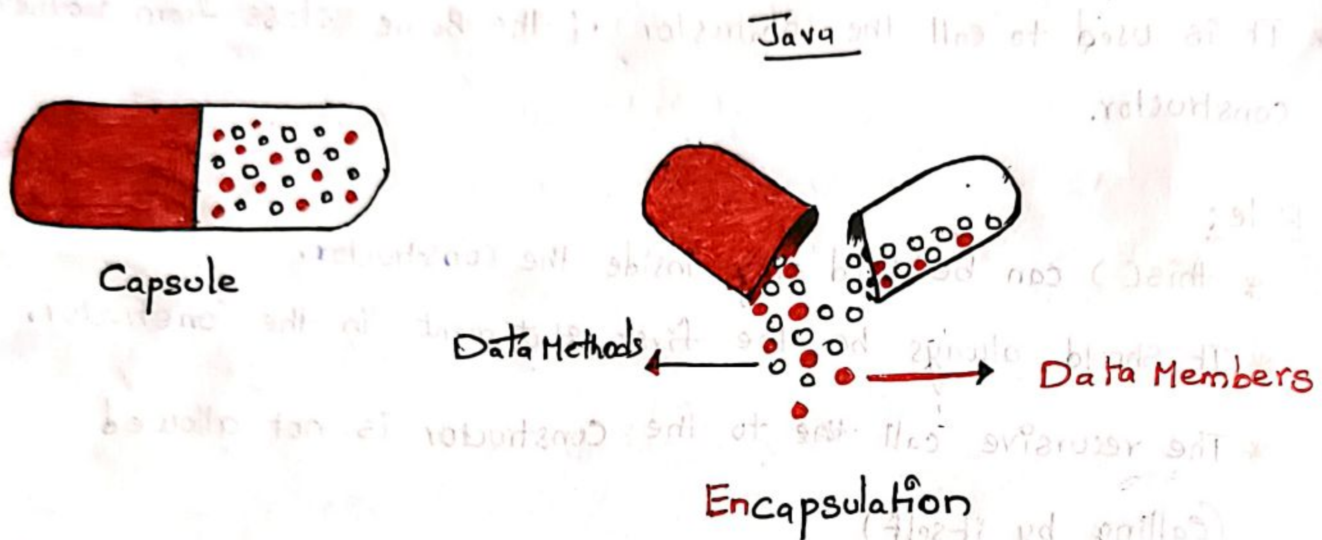
#OOPS Pillar

1. Encapsulation
2. Inheritence
3. Polymorphism
4. Abstraction → Interface.

1. Encapsulation: [Binding Variable and methods together of an Object.]

* Encapsulation in java is a process of wrapping code and data together into a single unit. for Example: a capsule which is mixed of several medicines.

* We Can Create a fully encapsulated class in java by making all the data members of the class private. Now we can use setter and getter



Encapsulation

Advantage of Encapsulation:

(Set the New value) ↗ (To getting a value)

* By providing only a setter or getter method, you can make the class read-only or write-only. In other words, you can skip the getter or setter methods.

* It provides you the Control over the data.

* It is a way to achieve data hiding in Java because other class will not be able to access the data through the private data members.

Example:

```
public class Passenger
{
    private String name;
    private int id;
    private String destination;

   public String getName( )
   {
       return name;
   }

   public int getId( )
   {
       return id;
   }

   public String getDestination( )
   {
       return destination;
   }

  public void setNames(String a)
  {
      this.name = a;
  }

  public void setId(int a)
  {
      this.id = a;
  }

  public void setDestination(String a)
  {
      this.destination = a;
  }
```

```
public class PassengerDriver
{
    public static void main(String[] args)
    {
        Passenger p1 = new Passenger();
        System.out.println(p1.getName());
        System.out.println(p1.getId());
        System.out.println(p1.getDestination());
        Sopln("-----------------");

        Passenger p2 = new Passenger();
        p2.setName("Abdulbarid");
        p2.setId(1120);
        p2.setDestination("bangalore");

        Sopln(p2.getName());
        Sopln(p2.getId());
        Sopln(p2.destination());
        Sopln("-----------------");
    }
}
```

Relationship:

* The Connection or Association between two Objects is known as the relationship.

Types of Relationships:

1. Has-a relationship
2. Is-a relationship

1. Has-a Relationship:

* If one Object is dependent on another it is known as a "has-a relationship."
* Based on the level of dependency, a relationship is classified into two types.
  1. Aggregation
  2. Composition.

1. Aggregation: (Weak)

* The dependency between two Object such that one Object can exist without the Other is known as "aggregation."

Example:

Cab → Ola (has-a)

Train → Online tick booking

Bus → Passenger.

Abdulbasid → girlFriend

Ashok → Car

2. Composition:

* The dependency between two Objects such that one Object can't exist without the other is known as "Composition"

Example:

Car → Engine (has-a)

Human → Oxygen.

Raghul → wife

fish → water

Activity

Note:

* In java we can achieve Has-a relationship.
* By creating the reference Variable of one Object inside another Object.

2. Is-a Relationship:

* The relationship between two objects which is similar to the parent and child relation is known as the "Is-A Relationship"
* In a Is-A relationship, the child Object will acquire all properties of the parent Object, and the child Object will have its own extra properties.
* In an IS-A relationship, we can achieve generalization and specialization.

Note-1:

* Parents are called generalized.
* Children are called Specialized

Note-2:

* Private members, constructors and initializers are not inherited to the child class.

Example:

* With the help of the child class reference type, we can use the members of the parent's class as well as the child.

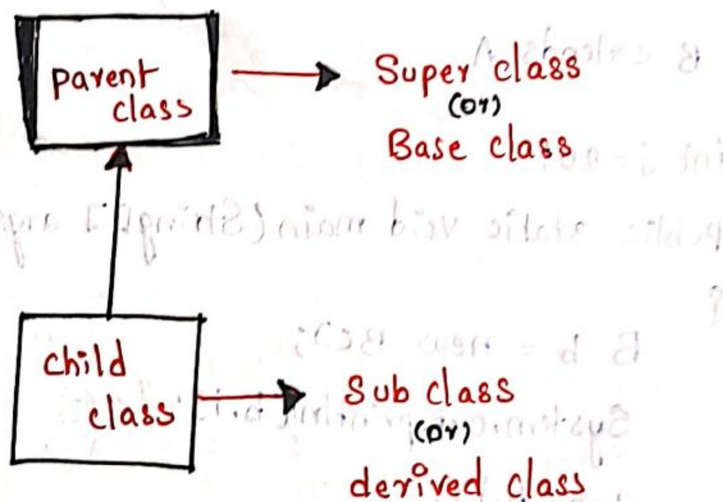
* With the help of parent class reference, we can use only the members of a parent but not the child class.

Parent class:

* The parent class is also known as a Super class or base class.

Child class:

* The child class is also known as a subclass or derived class



Note: In Java Is-A relationship is achieved with the help of Inheritance.

2. Implements keyword:

* Implements keyword is used to achieve inheritance between a class and an ~~inke~~ interface.

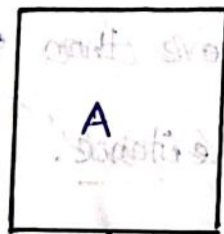
Types of Inheritance:

1. Single Level Inheritance
2. Muliti Level Inheritance
3. Hierarchical Inheritance
4. Mutiple Inheritance
5. Hybrid Inheritance

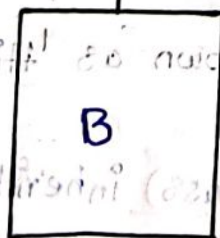
(4, 5) → * don't support in java with classes ✗
* do support interface ✓

1. Single Level Inheritance:

* Inheritance of only one level is known as "single-level inheritance"
* child class having one super class or super class having one child class is known as "Single Level inheritance."
* It is more efficient to create Object of child class.



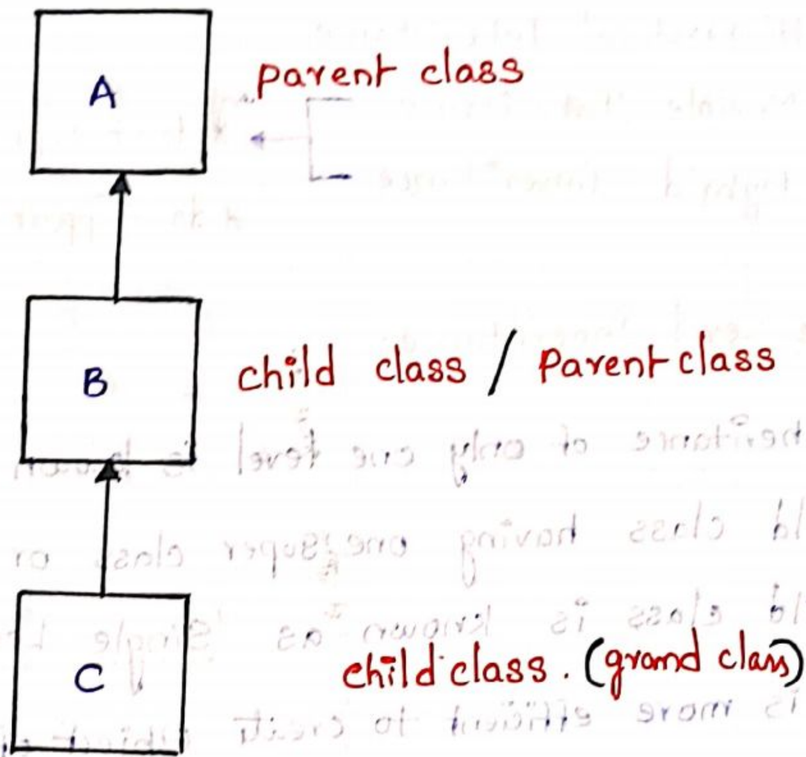
[parent class/ super class/ base class]



[child class/ ~~base~~ Sub class/ derived class]

2. Multi Level Inheritance:

* Inheritance of more than one level is known as 'multi-Level inheritance'.
* A class acting as a super class and also acting as a sub class is known as 'multi-level inheritance'.

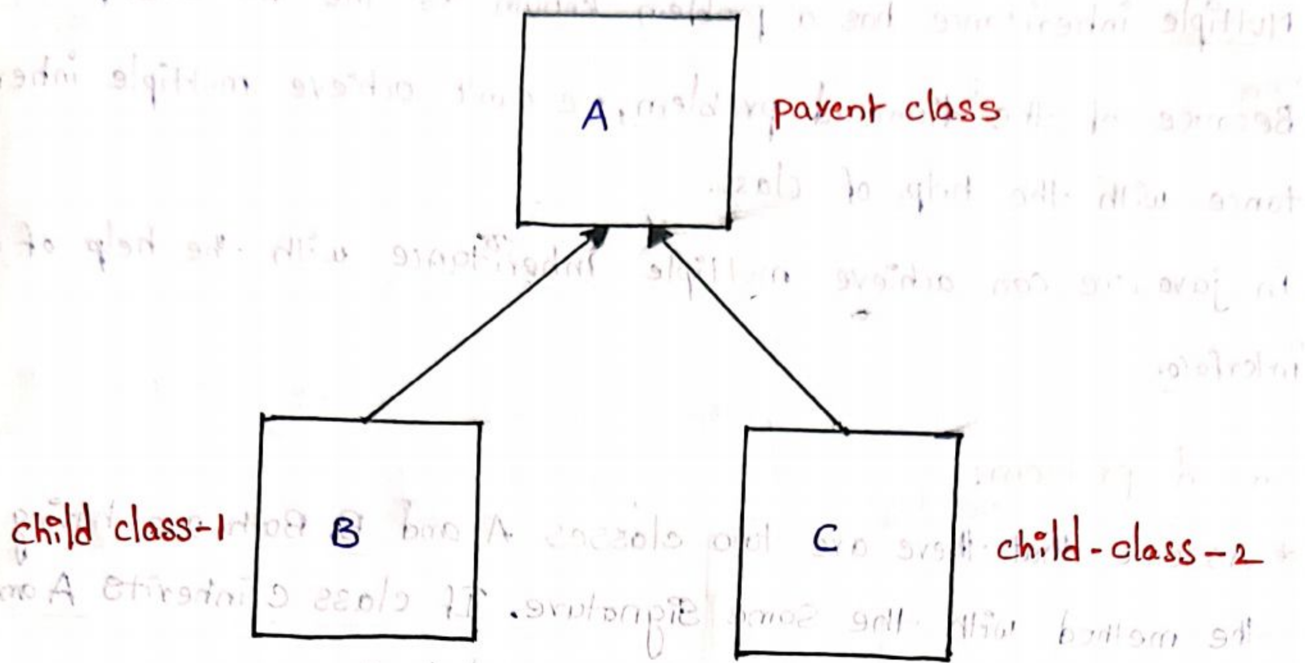


3. Hierarchical Inheritance:

* ~~If a subclass (child) inherits more than one parent (Super class) then it is known as 'multiple inheritance'.~~
* Creation of all the child class Objects is mandatory.
* If a parent (Super class) has more than one child (Sub class) at the same level then it is known as "Hierarchical Inheritance".

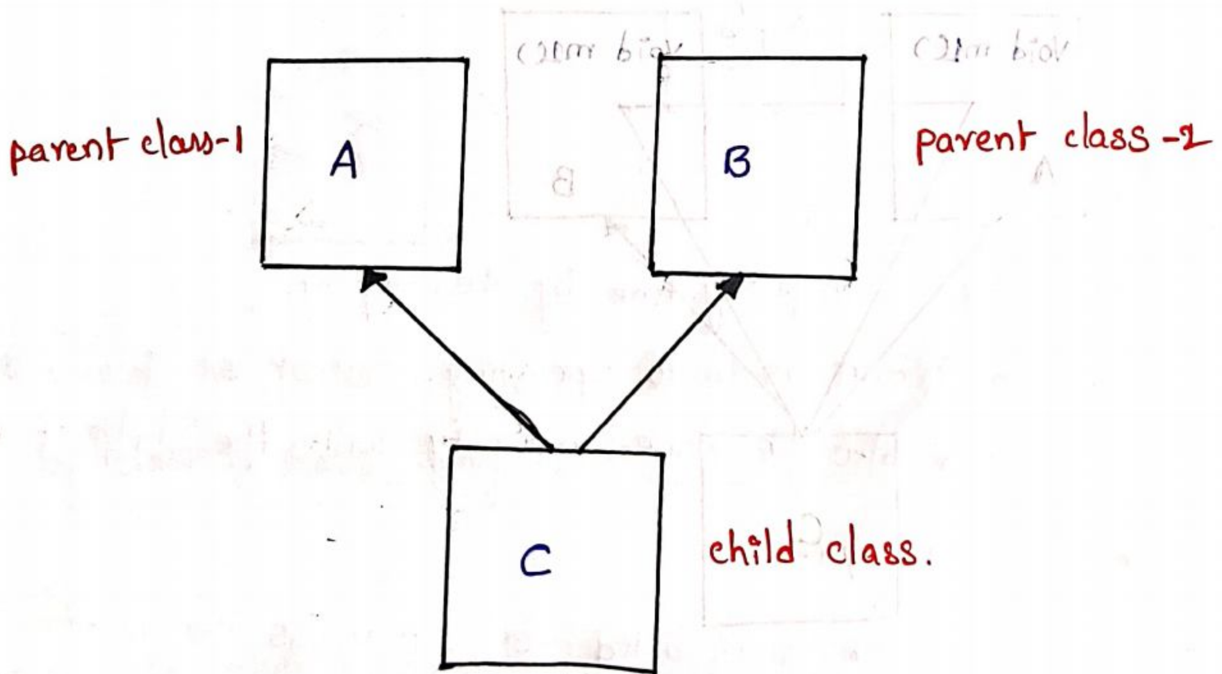
(or)

* more than once child class (Sub class) inheriting from same parent class (Super class) is called "Hierarchical Inheritance"



4. Multiple Inheritance:

* If a subclass child inherits more than one parent super class then it is known as "multiple inheritance."

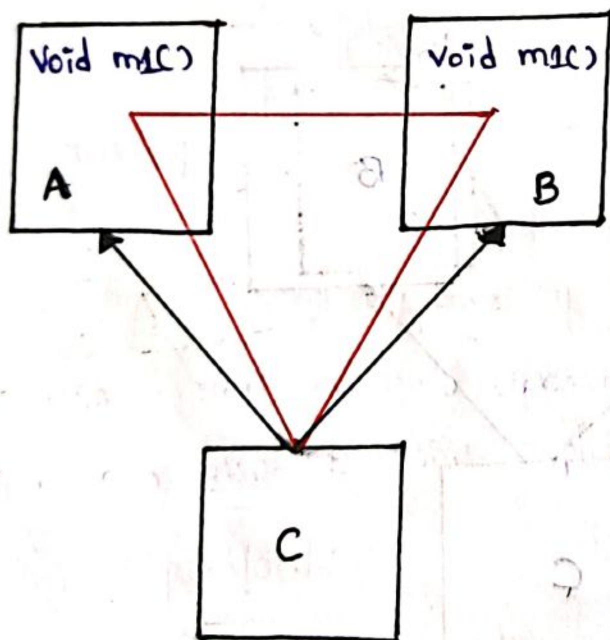


NOTE:

* Multiple inheritance has a problem known as the "diamond. problem".
* Because of the diamond problem, we can't achieve multiple inheritance with the help of class.
* In java, we can achieve multiple inheritance with the help of an interface.

Diamond problem:

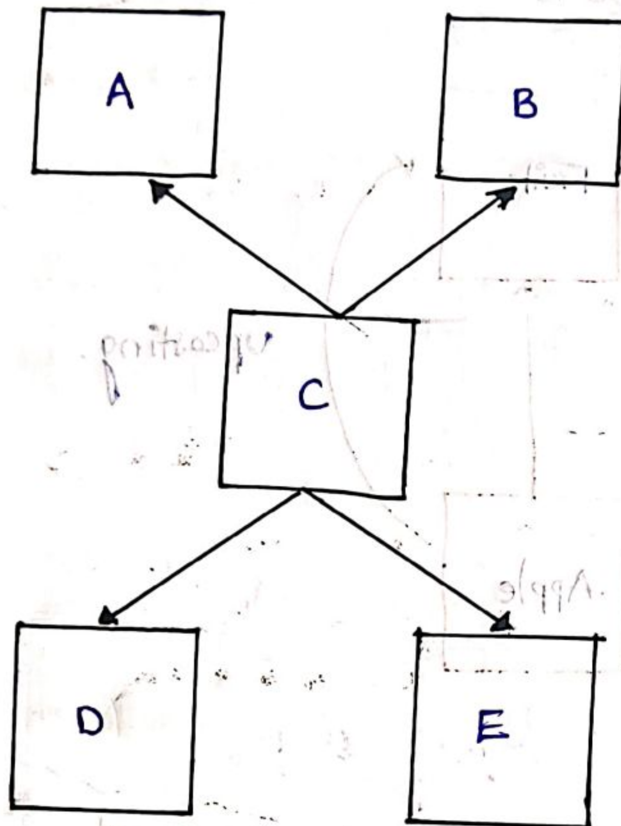
* Assume that there are two classes A and B. Both are having the method with the same signature. If class C inherits A and B then these two methods are inherited to C.
* Now an ambiguity arises when we try to call the superclass method with the help of subclass referenence.
* This problem is known as the diamond problem.



5. Hybrid Inheritance:

* The combination of multiple inheritance and hierarchical inheritance is known as hybrid inheritance.

Multiple Inheritance + Hierarchical Inheritance
= Hybrid Inheritance



* One of the reasons Here multiple Inheritance doesn't support in java by classes. That's why hybrid inheritance also not support.

Note:

* Multiple Inheritance and Hybrid Inheritance Cannot be achieved through classes in java.
* It can be achieving using Interface.

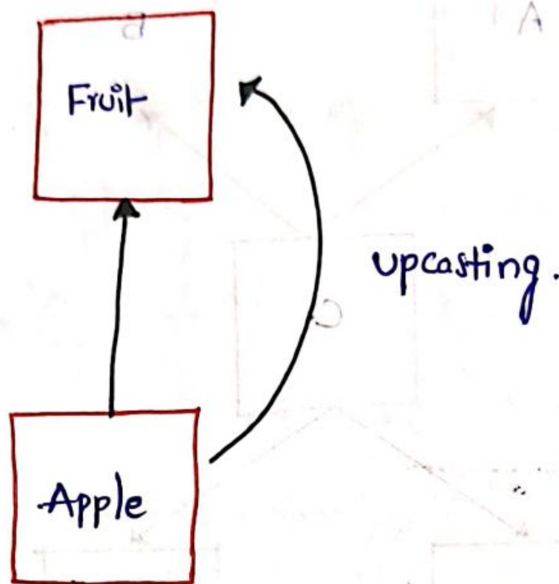
Types of Non-primitive or Derived type casting.

Non-primitive typecasting can be further classified into two types.

1. Upcasting
2. Downcasting.

1. Upcasting: child ⟶ Parent

* The process of converting the child class reference into a parent class reference type is known as upcasting.



Note:

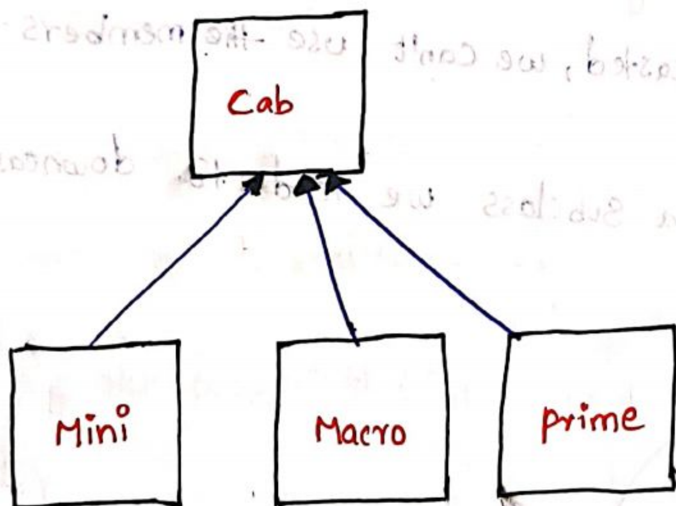
* The upcasting is implicity done by the Compiler.
* It is also known as auto upcasting.
* upcasting can also be done explicity with the help of a type cast Operator.
* Once the reference is upcasted we can't access the member of the child.

Example:

Why do we need upcasting?

* It is used to achieve generalization.
* It helps to create a generalization container so that the
* reference of any type of child Object can be stored.

Example:




```
Cab c;
c = new Mini();
c = new Macro();
c = new Prime();
```

DisAdvantage:

* There is only one disadvantage of upcasting that is, once the reference is upcasted its child members can't be used.

Note:

* In order to overcome this problem, we should go for "downcasting".

Downcasting: Parent ⟶ child

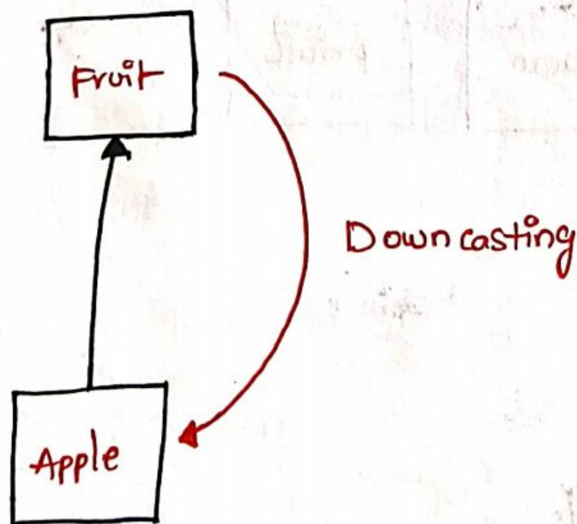
* The process of converting a parent (super class) reference type to a child (sub class) reference type is known as "downcasting".

Note:

* Downcasting is not implicity done by the "Compiler."
* It should be done explicity by the programmer with the help of a "typecast operator".

Why do we need a downcasting?

* If the reference is upcasted, we can't use the members of a Subclass.
* To use the members of a subclass we need to downcast the reference to a subclass.



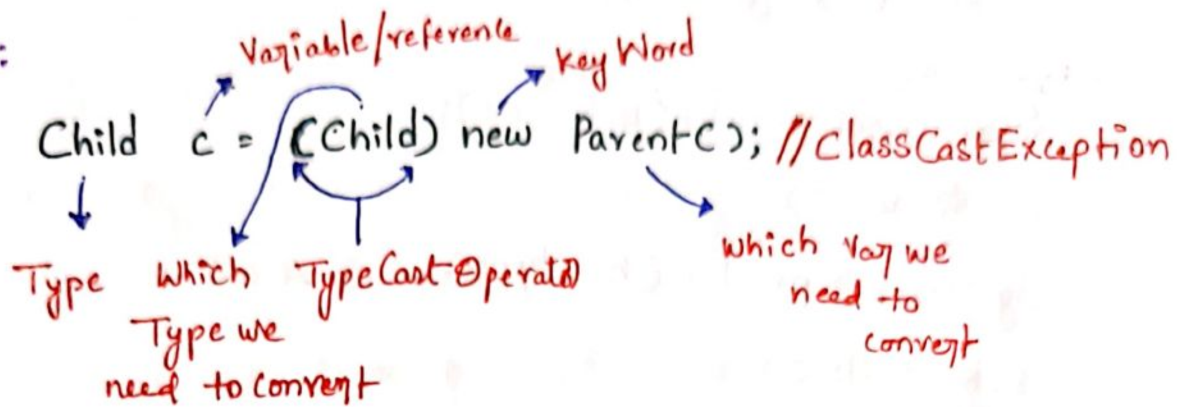
ClassCastException:

* It is a RuntimeException.
* It is a problem that occurs during runtime while downcasting

When and why do we get a ClassCastException?

* when we try to convert a reference to a specific type(class), and the Object doesn't have an instance of that type then we get ClassCastException.

Example:



Example-program:

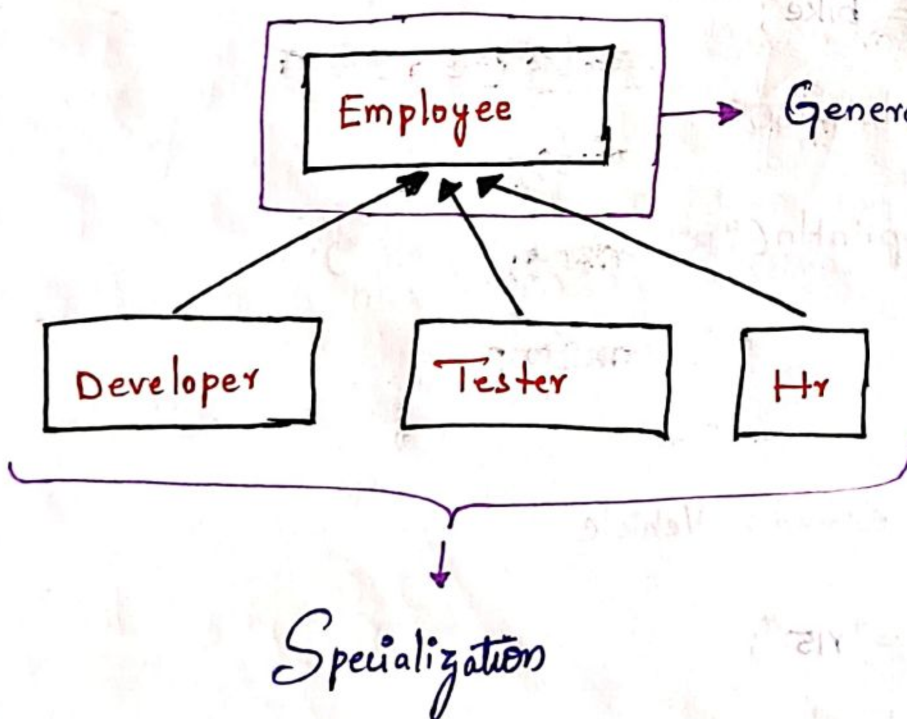
```
public class Vehicle
{
    String vtype = "bike";
    public void vdetails()
    {
        System.out.println("Bike is a Vehicle");
    }
}

public class Bike extends Vehicle
{
    String bname = "Y15";
    public void bdetails()
    {
        System.out.println("Y15 is on ride");
    }
}
```

```
public class Driver
{
    public static void main(String[] args)
    {
        Vehicle v = new Bike();
        System.out.println(v.vtype);

        System.out.println("------------");
        Bike b = (Bike).V;

        System.out.println(b.bname);
        b.bdetails();
        System.out.println(b.vtype);
        b.bdetails();
        System.out.println("------------");
    }
}
```



Developer d = new Developer();
Testy t = new Testy();
Hr hr = new Hr();

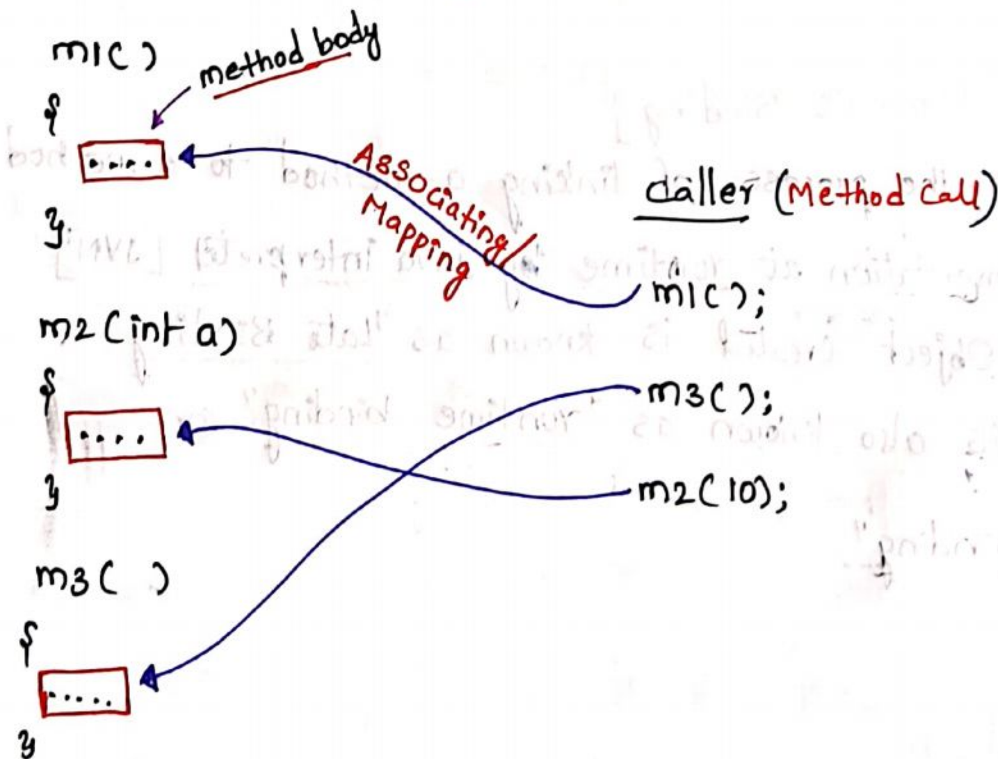
Employee e = new Employee();

Binding:

* Binding refers to the process of linking a method call to its corresponding method implementation.
* Binding decides which method implementation is classed when a method is called.
* If a method call is bound to the method definition it is known as "Binding."
* Binding can happen either at runtime or at compile time.
* Binding are two types, they are...

1. Early Binding
2. Late Binding

Example: //method Binding.



Note: Connecting a method call to the method body is known as "Binding

1. Early Binding: [Static Binding]

* Early binding is the process of linking a method call to the a method definition/implementation at compile time by Java Compiler [javac]

* This means that the decision of which method to call is made by the compiler based on the reference type of the class is known as "Early Binding"

* Early binding is also known as "Compile Time Binding" (or) "Static binding"

Example:

→ Method Overloading

→ Constructor Overloading

2. Late Binding: [Dynamic Binding]

* Late binding is the process of linking a method to a method definition/implementation at runtime by java Interpretor [JVM] based on the Object created is known as "Late Binding"

* Late binding is also known as "runtime binding" (or) "dynamic binding"

Example:

→ Method overriding

3. Polymorphism: [Many Forms]

* polymorphism in java is a Concept by which we can perform a Single action in different ways.
* An Object showing different behaviour at different stages of its lifecycle is called "polymorphism".
* polymorphism is derived from 2 Greek words: poly and morphs. The word

"poly" means many

"morphs" means forms

So polymorphism means "many forms".

Types of Polymorphism:

* There are Two types of polymorphism...
  1. Compile Time Polymorphism
  2. Run Time Polymorphism

* Before starting withe these two types of polymorphism here, Let we Learn Binding here. first.

1. Compile Time Polymorphism:

* The method declaration getting binded to its definition at the Compile Time by the Compiler based on the arguments passed is called "Compile Time Polymorphism"
* It is also known as "Early Binding" or "Static Binding"

Example :

→ Method Overloading

→ Constructor Overloading

→ Operator Overloading (java doesn't support).

Example - program :

```
class WhatsApp
{
    void send(int no)
    {
        System.out.println("Sending no " + no);
    }
    void send(String msg)
    {
        System.out.println("Sending msg " + msg);
    }
    void send(int no, String msg)
    {
        System.out.println("Sending msg and number " +msg+ " "
            +no);
    }
    void send(String msg, int no)
    {
        System.out.println("Sending number and msg "+no+ " "
            +no);
    }
}
```

```
class WhatsAppDriver
{
    public static void main(String[] args)
    {
        WhatsApp w1 = new WhatsApp();
        w1.send(123);
        w1.send("hello");
        w1.Send(126,"hi");
        w1.send("bye",127);
    }
}
```

Output:

```
Sending no 123
Sending msg hello
Sending number and msg 126 hi
Sending msg and number bye 127
```

2. RunTime polymorphism:

* The method declaration gets binded to its definition at the run time by the JVM based on the Object created is called as "RunTime polymorphism".
* It is also known as 'Late Binding' or 'Dynamic Binding'

Example:

Method Overriding

Example program:

```
class MethodOverridingR
{
    void cool( )
    {
        System.out.println("hello java");

    }
}

class Tester extends MethodOverridingR
{
    void cool( )
    {
        System.out.println("Hello java");

    }
}

class MethodDriver
{
    public static void main(String[] args)
    {
        //mo d1 = new Tester( );
        MethodOverridingR m1 = new Tester( );
        m1.cool( );
    }
}
```

Output: Hello java

Method Overriding:

* Parent class and child class having same method names is known as method Overriding.
* method Overriding is defined as same method name with same Signature (arguments).
* Method Overriding is used to provide the specific implementation of a method which is already provided by its superclass.
* Method Overriding is used for runtime polymorphism.

Rule:

1. Is-A relationship, must be.
2. Upcasting
3. The method be non-static
4. method must have same name as in the parent class.
5. method must have same parameter as in the parent class.

Example-program:

```
public class WhatsApp_V1
{
    void status( )
    {
        System.out.println("Status with text");
    }
}
public class WhatsApp_V2 extends WhatsApp_V1
{
    void status( )
    {
        System.out.println("Status with text, images, videos");
    }
}
```

Overriding

If both parent and childclass methods are non-static. It will be Overriding

```
public class MainClass
{
    public static void main(String[] args)
    {
        WhatsApp_V1 w2 = new WhatsApp_V2(); //upcasting

        w2.Status();
    }
}
```

Output: Status with text, images, videos.

Example-program-02:

```
class Kitkat
{
    void camera()
    {
        System.out.println("Back Camera");
    }

class Lollipop extends Kitkat
{
    void camera()
    {
        System.out.println("Front and back camera");
    }

ass MainClass1

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

```
        Lollipop ll = new Lollipop();
        ll.Camera();
    }
}
```

Output: Front and back camera

Super keyword:

* Super keyword in java is used to access the properties of immediate parent class.
* IS-A relationship is mandatory.
* Super keyword can be use inside inside non-static method and constructors but not in static methods.
* Super keywords is used to access super class variables and methods.
* Super keyword can be written as any line of the method or a Constructor.
* Super keyword is used in the case of method overriding, along with the sub class implementation, if we need super class implementation thus we should go for `Super.methodName`

Note:

* We go for inheritance for code reusability
* We go for method overriding whenever we want to provide new implementation for the old feature.
* We go for method overloading whenever we want to perform common task and operation.

Example-program-01:

```
class PhonePe_V1
{
    void rewards()
    {
        System.out.println("Rewards by money");

    }

}
class PhonePe_V2 extends PhonePe_V1
{
    void rewards()
    {
        System.out.println("Rewards by Coupon");
        Super.rewards();
    }
}
class MainClass
{
    public static void main(String[] args)
    {
        PhonePe_V2 p2 = new PhonePe_V2();
        P2.rewards();
    }
}
```

Output:

Rewards by Coupon

Rewards by money.

Super() - Call to Super

* Super() is use in java call the constructor of immediate parent class.
* It is used to call from sub class constructor to the immediate super class constructor.

Rules:

1. IS-A relationship is mandatory.
2. Call to super should be written first line of code inside the constructor.
3. Call to super is used only inside constructor not in methods.
4. it is used only in the case of inheritance.
5. In every constructor developed or defined with in the class, JVM internally executes call to super(no arguments)
6. Super() is a internal call / implicity call by JVM for all child class constructor. and it should be no argument constructor
7. Super calling can be written implicity or explicity.

Example-program-01:

```
class Demo
{
    Demo(String a)
    {
        System.out.println("hahaha");
    }
}
```

```
class Tester extends Demo
{
    Tester(double y)
    {
        super("hi");
        System.out.println("cool");

    }
}
class Sample extends Tester
{
    Sample(int a)
    {
        super(10.56);
        System.out.println("hii");
    }
}
class MainClass
{
    public static void main(String[] args)
    {
        new Sample(10);
    }
}
```

Output:

```
hahaha
cool
hii
```

difference between this keyword, this() and super keyword and super():

this keyword:

1. this keyword is used to point to current object.
2. this keyword is used in the same class.
3. can be used inside non-static methods and constructor, not in static methods.
4. can be used in any line of the method.
5. this keyword is used by default for non-static members.
6. this keyword can be used n times.

this()

1. this() is used to call the constructor of the same class.
2. IS-A relationship is not required.
3. used only for constructors not for methods.
4. this() is executed only as the first line of the constructor.
5. this() is a explicit call.

super keyword:

1. super keyword is used to access the properties of parent class.
2. IS-A relationship is mandatory.
3. can be used inside non-static methods and constructor not in static methods.
4. super keyword should be explicity used for non-static members
   super keyword can be used n times
5. can be used in any line of the method.

Super( )

1. Call to Super is used to call the Constructor of the Parent Class.
2. IS-A relationship is mandatory.
3. Used only for constructor not for methods.
4. Super( ) is executed only as the first line of the constructor.
5. Super( ) is implicit call and also can Explicit call.

4. Abstraction:

* It is a design process of hiding the implementation (essential Features) and showing only the functionality (only declaration) to the user is known as "Abstraction." (Behaviour)
* It is also known as "Implementation hiding"

How to achieve abstraction in java?

* In java, we can achieve abstraction with the help of:
  * abstract class 0% to 100% abstraction
  * Interface 100% abstraction
* we can provide implementation to the abstract component with the help of Inheritance and Method Overriding.

Abstract Modifier:

* The abstract is a modifier, it is a keyword.
* It is applicable for methods and classes.

Abstract Method:

* A method that is prefixed with an abstract modifier is known as the "abstract method."

* This is also said to be an incomplete method.
* The abstract method doesn't have a body (it has the only declaration)

Syntax to create abstract method:

```
abstract [access_Modifier] [returnType] methodName([Formal-Arg]);
```

Note:

* Only child class of that class is responsible for giving implementation to the abstract method.

Abstract class:

* If the class is prefixed with an abstract modifier then it is known as "abstract class."
* We can't create the Object (Instance) for an abstract class.

Note:

* We can't instantiate an abstract class.
* An abstract class can have both abstract and concrete methods.
* If a class has at least one abstract method either declared or inherited but not overridden it is mandatory to make that class an abstract class.

Example:

```
abstract class Atm
{
    abstract public double withdrawal();
    abstract public void getBalance();
    abstract public void deposit();
}
//hiding implementation by providing only functionality.
```

Atm a = new Atm(); //CTE

NOTE:

* only subclass of Atm is responsible for giving implementation to the methods declared in an Atm class.

Implementation of abstract method:

* If a class extend abstract class then it should give implementation to all the abstract method of the superclass.
* If inheriting class doesn't like to give implementation to the abstract method of superclass then it is mandatory to make subclass as an abstract class.
* If a subclass is also becoming an abstract class then the next level child class is responsible to give implementation to the abstract methods.

Steps to implement abstract method:

Step 1: Create a class

Step 2: Inherit the abstract class/ component

Step 3: Override the abstract method inherited (provide implementation to the inherited abstract method.).

Concrete class:

* The class which is not prefixed with an abstract modifier and doesn't have any abstract method, either declared or inherited is known as a "Concrete class."

NOTE:

* In java, we can create objects only for the Concrete class.

Concrete Method:

* The method which gives implementation to the abstract method is known as the "Concrete method."

Example:

```
abstract class WhatsApp
{
    abstract public void send( );
}
class Application extends WhatsApp
{
    public void send( )
    {
        System.out.println("Send( ) method is implemented");
    }
}
```

Create Object and calling the abstract method:

```
WhatsApp w = new Application( );
w.send( ); //send( ) method is implemented.
```

Interfaces:

* It is a component in java which is used to achieve 100% abstraction with multiple implementation.

Syntax to create an interface

```
[Access Modifier] interface [interfaceName]
{
    //declare members.
}
```

* when an interface is compiled we get a class file with an extension .class only.

Example:

```
interface Demo
{

}
```

* The Demo is an interface.

NOTE:

* In interface, all the members are by default have public access modifier

```
interface Demo1
{
    int a; //CTE: variable a is by default public, static, final.
               A final variable must be initialized.
}
```

```
interface Demo2
{
    public void test() //CTE
    {

    }
}
```

Why do we need an interface?

* To achieve 100% abstraction. Concrete non-static methods are not allowed
* To achieve multiple inheritances.

What all the members are not inherited from an interface?

* Only static methods of an interface are not inherited to both class and interface.

Inheritance with respect to interface:

* An interface can inherit any number of interfaces with the help of an extended keyword.

Example:

```
interface I1
{

}
interface I2 extends I1
{

}
```

NOTE:

* The interface which is inheriting an interface should not give implementation to the abstract methods.

Example 1:

* Interface I1 have 3 methods.

    2 - non static (t1(), t2())

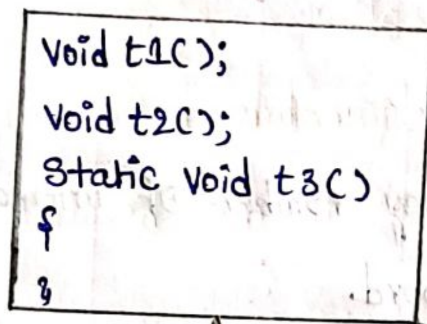
    1 - static (t3())

* Interface I2 have 3 methods

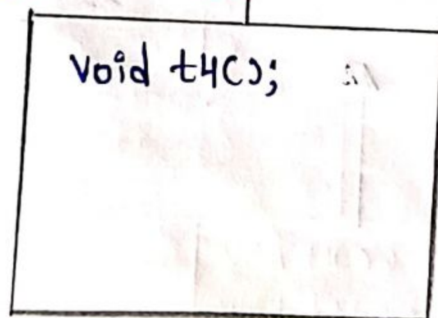
    2 - inherited non static method (t1(), t2())

    1 - declared non static methods (t4()).

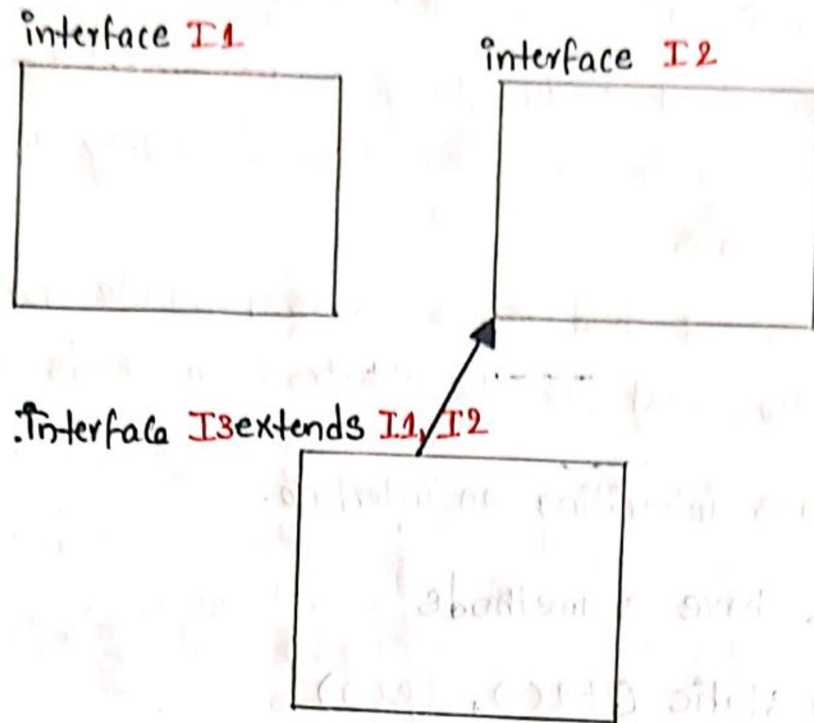
I1



I2 extends I1



Example-3: Interface can inherit multiple interfaces at a time.



NOTE:

With respect to interface there is no diamond problem

The reason:

* They don't have constructor
* Non-static methods are abstract (do not have implementation)
* Static methods are not inherited.

Inheritance of an interface by the class.

* class can inherit an interface with the help of implements keywords.
* class can inherit more than one interface.
* class can inherit a class and an interface at a time.
* Class can inherit

NOTE:

* If a class inherits an interface then it should give implementation to the abstract non-static methods of an interface.
* If the class is not ready to give implementation to the abstract methods of an interface then it is mandatory to make that class an abstract class.
* The next level of child class is responsible for giving impleme-ntation to the rest of the abstract methods of an interface.

Example-1: class inheriting an interface.

* Interface I1 have 3 methods

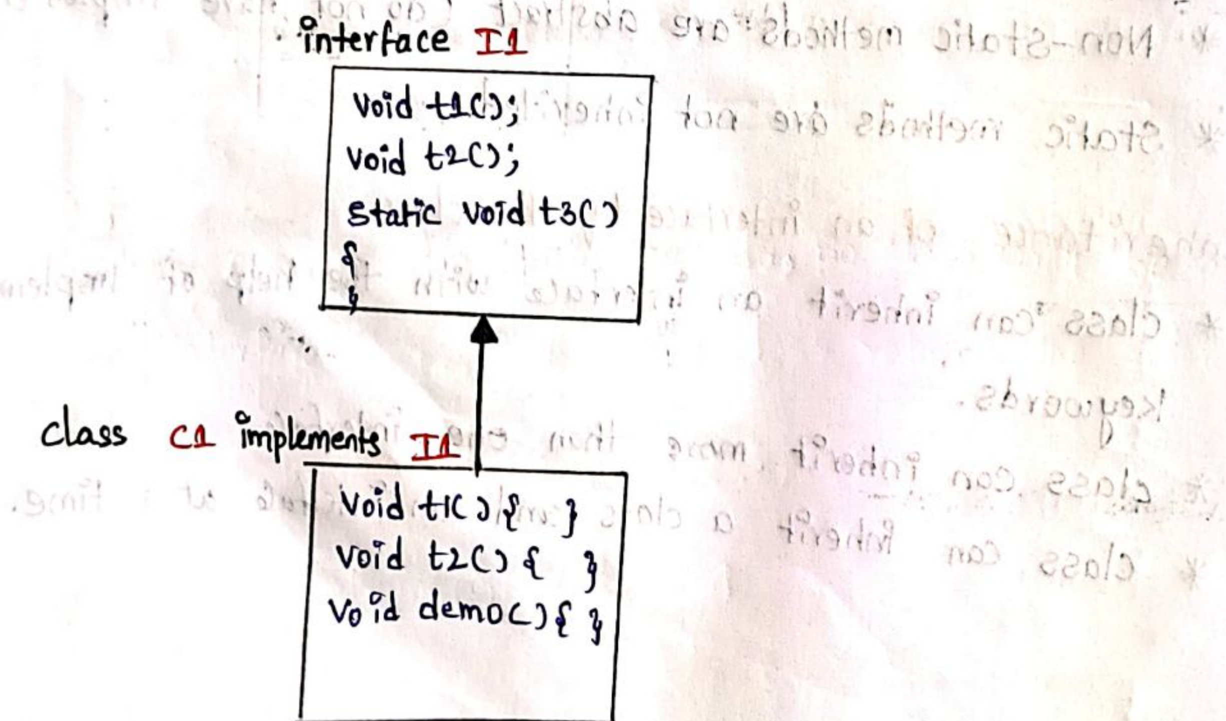
  2 - non static (t1(), t2())

  1 - static (t3())

* class C1 have 3 methods

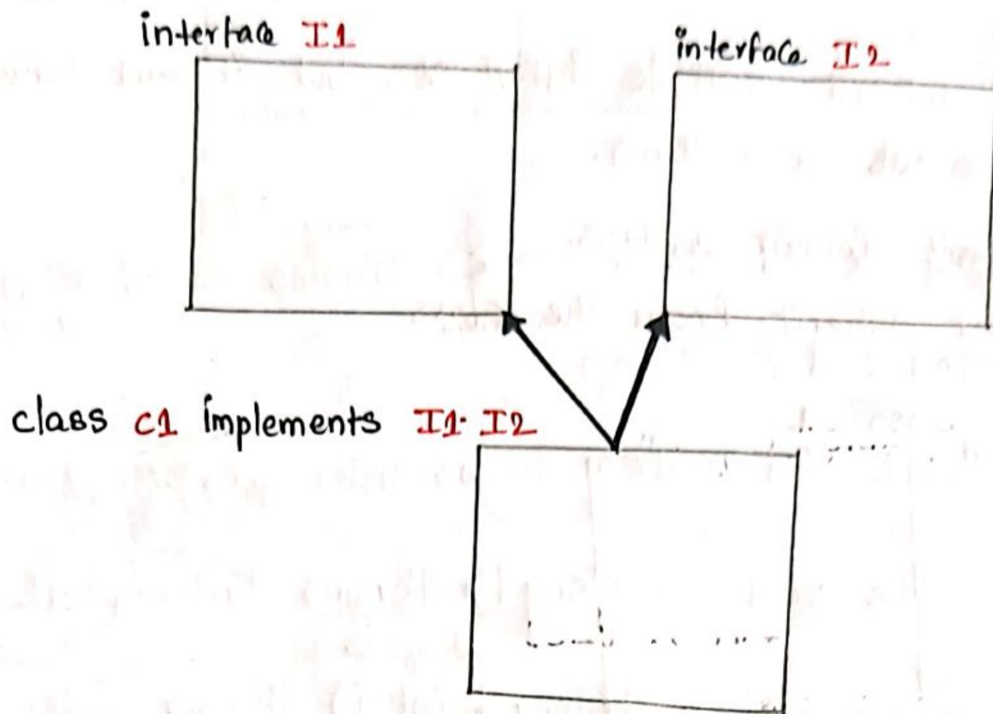
  2 - inherited and implemented non static method (t1(), t2())

  1 - declared non static methods (demo())



Example-2: class inheriting multiple interfaces.

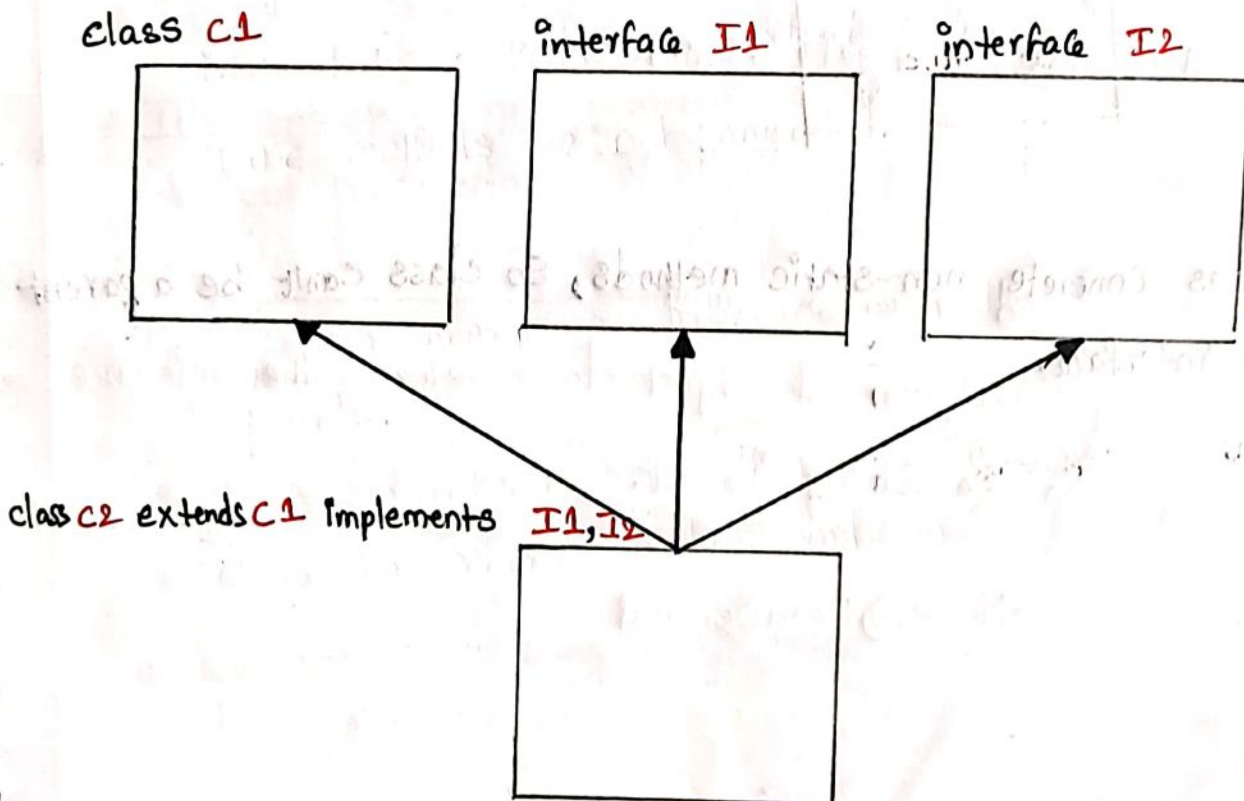
* The diamond problem is solved by implementing multiple interfaces by the class at a time.



Reason:

* Non-static methods are not implemented in an interface
* Static methods are not inherited.

Example-3: class can inherit interface and class at a time.



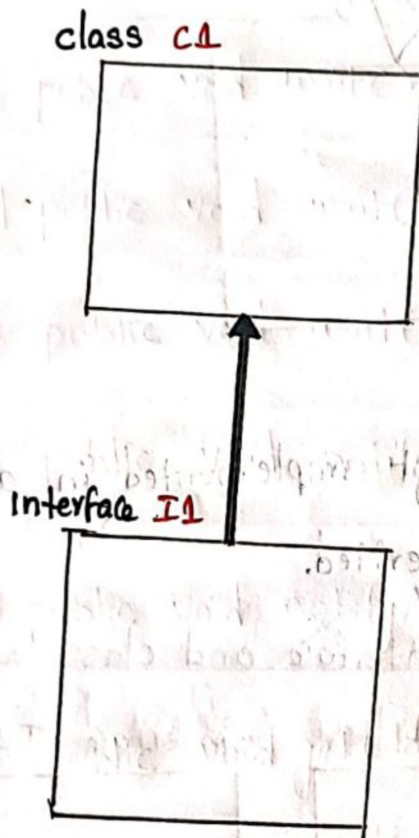
Rule:

* use the extends first and then implements.

Note:

* class can inherit multiple interfaces but it can't inherit multiple classes at a time.
* interface can't inherit a class.

Interface can't inherit from the class.

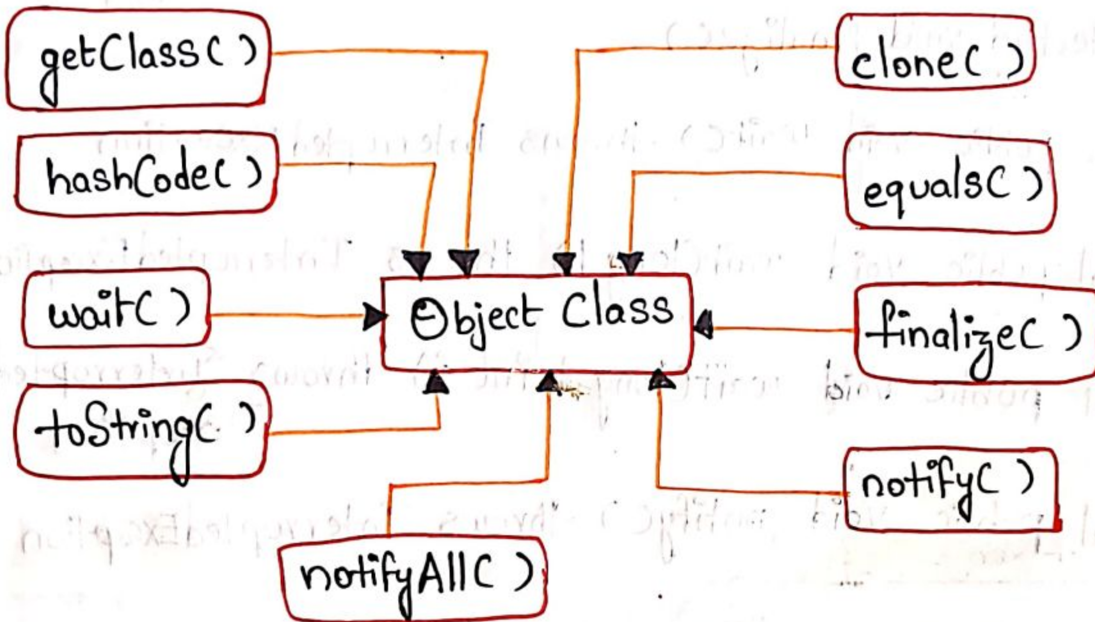


Reason:

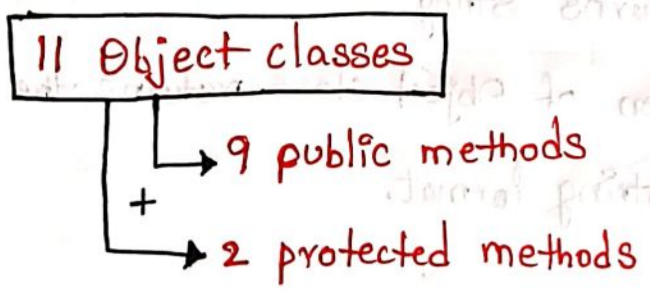
* Class has concrete non-static methods, so class can't be a parent to the interface.

Object class: [Object Parent class:]

* Object class is a supermost class for all the classes in java
* Object class is defined in java.lang package.
* In Object class there are (11) Non static methods.



* It is a predefined class.
* It's consist of no argument constructor only but not parameterized constructor.



* We are discussing only three in core java
    1. toString()
    2. equals()
    3. HashCode()

1	public String toString( )
2	public boolean equals(Object o)
3	public int hashCode( )
4	~~public~~ protected Object clone( ) throws CloneNotSupportedException
5.	protected void finalize( )
6.	final public void wait( ) throws InterruptedException
7	final public void wait(long l) throws InterruptedException
8	final public void wait(long l, int i) throws InterruptedException
9.	final public void notify( ) throws InterruptedException
10	final public void notifyAll( ) throws InterruptedException
11	final public void getClass( )

toString( ):

* toString( ) method returns String
* toString( ) implementation of Object class returns the reference of an Object in the String format.

Return Format : className@HexaDecimal

Without toString we get address of ~~variable~~ Objects.

Example-program:

```
class Demo
{
    public static void main(String[] args)
    {
        Demo d = new Demo();
        System.out.println(d); //d.toString() --> Demo@HD
    }
}
```

NOTE:

* Java doesn't provide the real address of an Object.
* Whenever programmer tries to print the reference variable toString is implicity called.

equals(Object obj) :

* The return type of equals(Object obj) method is boolean.
* To equals(Object obj) method we can pass reference of any Object
* The java.lang.Object class implementation of equals(Object obj) method is used to compare the reference of two Objects.

purpose of overriding equals(Object obj)

* We override to equals(Object obj) method to compare the state of an two Objects instead of comparing reference of two Objects.

NOTE:

* If equals(Object obj) method is not overridden it compares the reference of two Objects similar to == Operator.
If equals(Object obj) method is overridden it compares the state of two Object, in such case comparing the reference of two Objects is -

- possible only by == Operator.

Design tip:

* In equals method compare the state of an current(this) object with the passed object by downcasting.

hashCode() :

* The return type of hashCode() method is int.
* The java.lang.Object implementation of hashCode() method is used to give the unique integer number for every Object created.
* The unique number generated based on the reference of an Object.

Example:

```
       accessModifier/specifier
      ↗          ↗ methodName
public int hashCode()
{       ↳ modifier
    return 1234;
}
```

* hashcode() is using in collection.

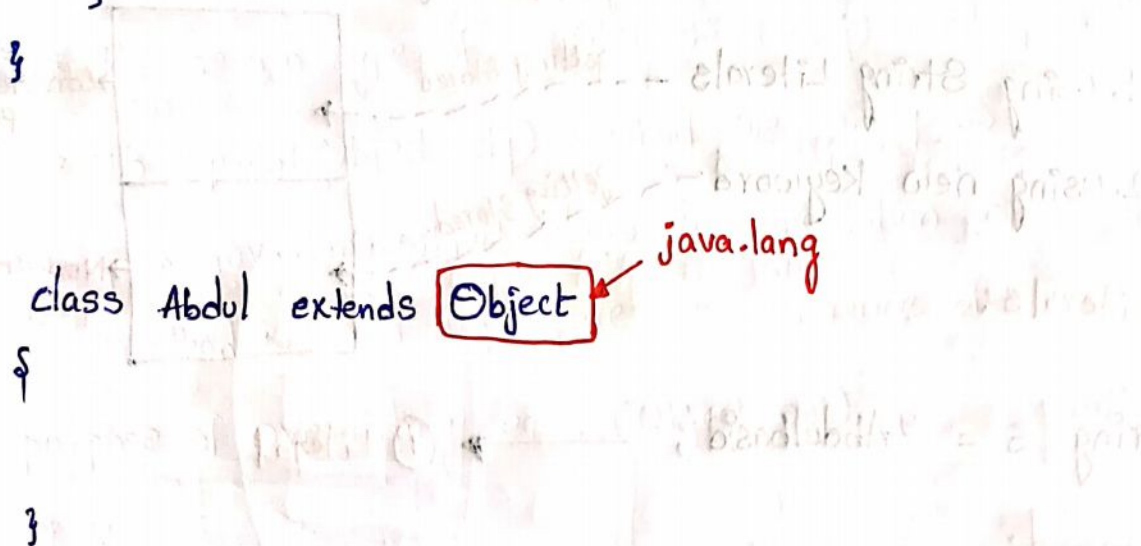
Example: //equals()

```
public class Student
{
    int id;
    Student(int id)
    {
      this.id = id;
    }
```

```
public void StudentDetails( )
{
    System.out.println(id);
}
}
```

```
publis boolean equals(Object o)
{
    Student s =(Student) o;
    return this.id = s.id;
}
```

```
public class Driver
{
    public static void main(String [] args)
    {
        Student s1 = new Student(111);
        System.out.println(s1);//
        Student s2 = new Student(111);
        System.out.println(s1 == s2); //false - Comparing Address.
        System.out.println(s1.equals(s2)); //false - Comparing Address
                      //true - after Creating Object and upcating
    }
}
```



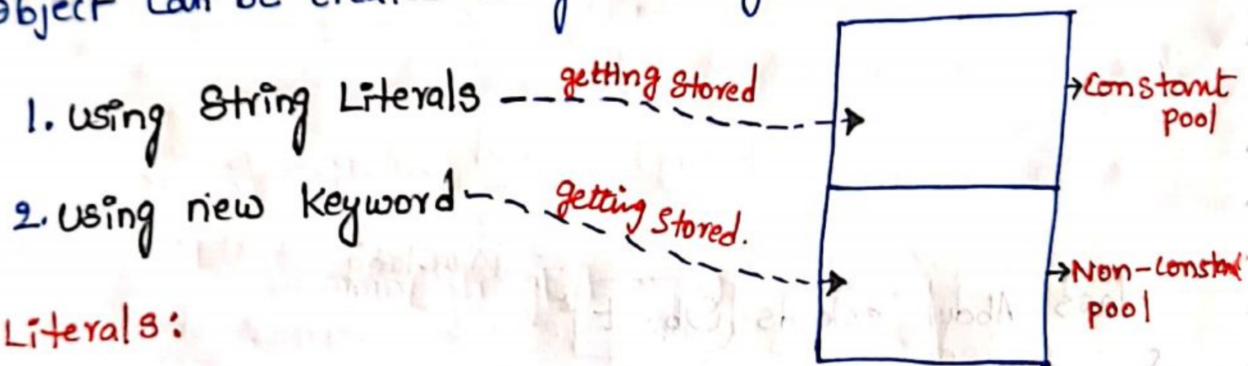
```
class Abdul extends Object   <- java.lang
{

}
```

String :

* "String" is surrounded by double quotes (" ").
* A "String" in java is an Object that represents a sequence of characters.
* A "String" class is a non-primitive or reference data Type in Java.
* A String is a predefined class in java and non-static method in java.
* The "String" class is part of java.lang package and provides methods for string manipulation.
* The "String" class in java is declared as "final" keyword and modifier
* "String" in java is "immutable", meaning that their content cannot be changed once they are created.

* String object can be created using two ways.



1. String Literals:

String s = "Abdulbasid"; → ① Literal

2. new keyword:

String s = new String("Abdulbasid"); → ② Object

Example: 01

```
Static int a = 10;
a = 20;
Sopln(a); //20
```

a | ~~10~~ 20

Example-02:

```
final int b = 5;
b = 10; //CTE
```

b | 5

Example-03:

```
double d = 10.1;
d = 20.2;
```

d | ~~10.1~~ 20.2

Example-04:

```
String s = "Abdulbasid";
String s = "CseHacker";
```

~~s~~ | "Abdulbasid"

s → | "CseHacker"

* Every time new, new container will be created for each and everytime in String.

Example-05:

```
class Demo
{
    p s v m(—)
    {
        String s1 = "Hello";
        String s2 = "Hello";
        Sopln(s1 == s2);
    }
}
```

Heap area

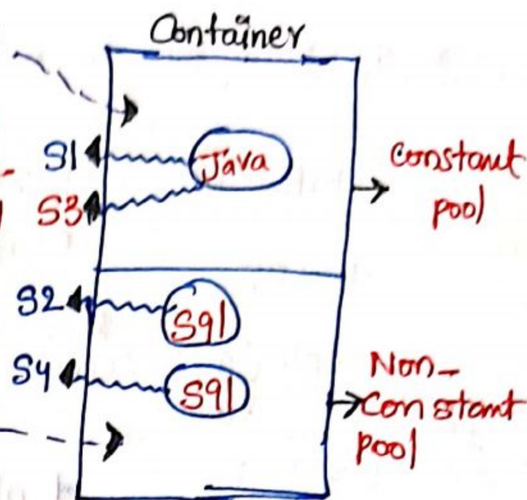
String Constant pool

java.lang@123

s1, s2 → "Hello" → Space

here no new space will be created, only pointing with new variable for same value/space.

```
String s1 = "Java";
String s2 = new String("sql"); → New space will Created in Non-constant pool
String s3 = "Java"; // NO-space will created
String s4 = new String("sql");
```
↓ New Space will be created in non-constant pool.



Non-Static methods in String:

* The String class in java provides a variety of non-static methods for manipulating and examining strings.
* These methods operate on in individual String Objects and do not require a static context.
* These are the methods which are present in String class.
* We are not overriding, Just calling them.

Example-program:

```
public class NonStaticMethod
{
    public static void main(String[] args)
    {
        String s = "Software Developer";

        System.out.println("---------");
        System.out.println(s.length());   // 18
        Sopln("---------");
        System.out.println(s.toUpperCase());  // SOFTWARE DEVELOPER
        Sopln("---------");
        System.out.println(s.toLowerCase());  // software developer
```

```
Sopln("---------");
System.out.println("s.startsWith("soft")); //false
Sopln("---------");
System.out.println("s.startsWith("Soft")); //true
Sopln("--------");
System.out.println("s.endsWith("Per")); //-false
System.out.println(s.endsWith("per")); //-true
Sopln("--------");
System.out.println(s.contains("dev")); //-false
System.out.println(s.Contains(" Dev")); // true
Sopln("---------");
System.out.println(s.concat(" is Abdulbasid")); //Software Developer
                                                    is Abdulbasid
Sopln("---------");
System.out.println(s.indexOf('f'); //2
Sopln("---------");
System.out.println(s.charAt(5)); //a
Sopln("---------");

String a = "Java";
String b = "java";
String c = "Java";
System.out.println(a.equals(b)); //false
System.out.println(a.equals(c)); // true
Sopln("--------");
System.out.println(a.equalsIgnoreCase(b)); //true
System.out.println(c.equalsIgnoreCase(b)); //true
```

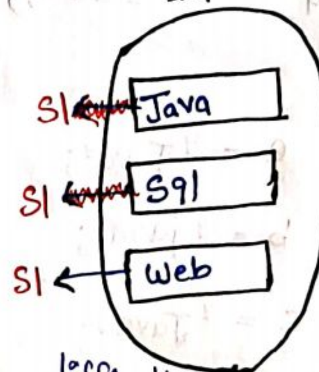
```
String x = "Hello dinga";
String y = "CSEH4-CK3R";

System.out.println(x.substring(3)); //lo dinga
System.out.println(y.substring(3)); //H4-CK3R
Sopln("------");
System.out.println(x.substring(2,7)); //llo d
System.out.println(y.substring(3,7)); //H4-CK
Sopln("------");
}
}
```

Why String is immutable ?

* String Object is given as immutable for storing this Object as a key in Collection map Object.
* Once a String Object is created, its value cannot be changed.

```
String S1 = "Java";
S2 = "Sql";
S3 = "web";
```



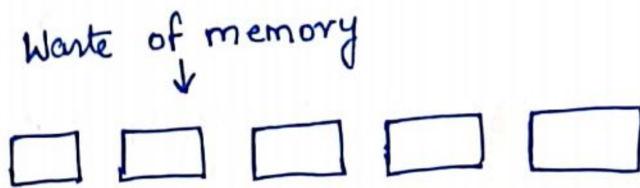
Advantages:

* Improving Security by making it more difficult for malicious code to alter the value of String Object.
* providing a more intuitive way of working with Strings, as it is clear that the value of a String Object cannot be changed after it is created.

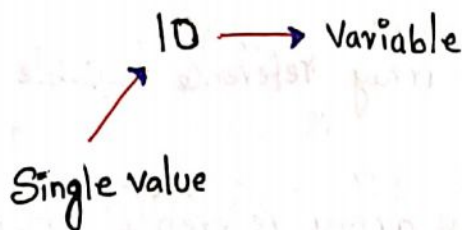
* Enhancing performance by allowing the JVM to optimize the way it manages String Objects.

* Simplifying the process of creating and managing String Objects, as developers do not need to worry about the Object's value being modified. [data security]

Disadvantage:



Variable:



* If we want store multiple values, we need "array"

ARRAY:

Array:

* Array is a Continuous block of memory which is used to store multiple values.

Characteristic of an Array:

1. The size of an array must be defined at the time of declaration.
2. Once declared, the size of an array can't modified.
3. Hence array is known as fixed size.
4. In an array we can access the elements with the help of an index or subscript. It is an integer number that starts from 0 and ends at - length of the array - 1.

5. In an array we can store only homogeneous type value. It is also known as homogeneous collection of an Object.

NOTE: In Java array is an Object.

Declaring an array: We are able to declare arrays in two ways.

Syntax to declare an array:

```
dataType[] variable;

        (or)

dataType variable[];
```

Example:

int a[] → Single dimensional array reference variable of int Type

float f[] → Single dimensional array reference variable of flat type.

String s[] → Single dimensional array reference variable of String type

Instantiating an Array:

Syntax to instantiati an array:

```
new dataType[size];
```

Example :

new int[5];

	0X1
0	0
1	0
2	0
3	0
4.	0

new String[5];

	0X1
0	null
1	null
2	null
3	null
4	null

new · boolean[4];

	0X1
0	false
1	false
2	false
3	false

NOTE : Once the array is instantiated it is assigned with default value

Initializing an array :

Adding Elements:

* We can add an element into an array with the help of array index.

Syntax to add an element into an array :

```
array-ref-variable[index] = value;
```

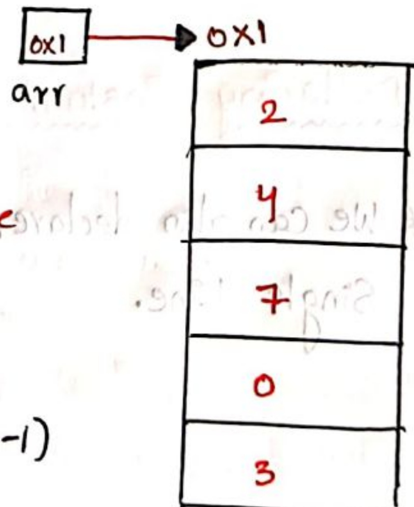
Example:

```
int[] arr = new int[5]; //declaration
arr[0] = 2;
arr[1] = 4;
arr[2] = 7;
arr[3] = 0;
arr[4] = 3;
```

(arr → arrayName)

int[5] ↓ It can not be reduce, It cannot be Increase once size is given

Length = 5
Index = 4 (n-1)



Accessing Elements from an array:

Accessing Elements:

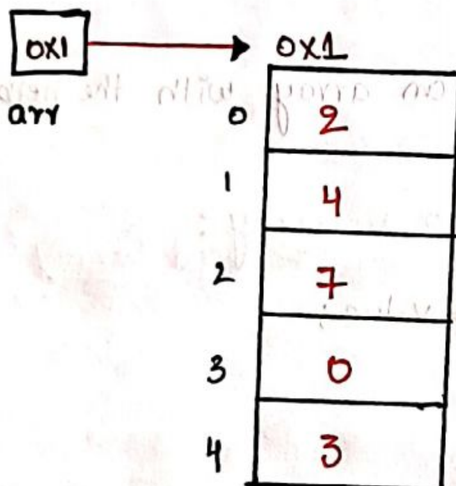
* We can access an element from an array with the help of array reference variable.

Syntax to access an elements from an array:

```
array-ref-variable[index];
```

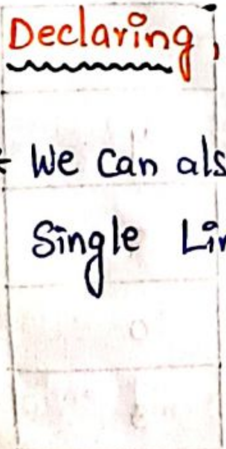
Example:

```
System.out.println(arr[0]); //2
System.out.println(arr[2]); //7
System.out.println(arr[4]); //3
System.out.println(arr[5]); //ArrayIndexOutOfBoundsException
```



Declaring, Instantiating, Initializing An array in a Single Line:

* We can also declare, Instantiate and Initialize an array by using Single Line.

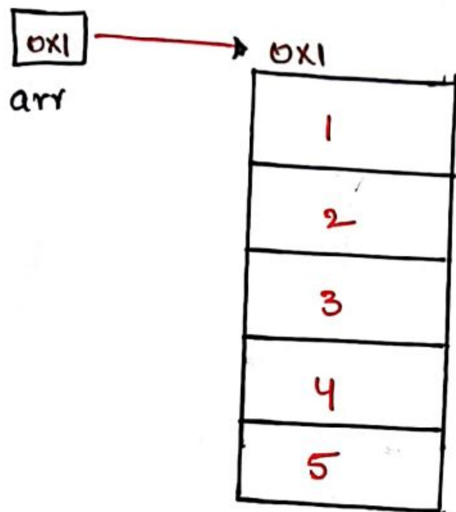


Syntax:

```
dataType[] arr_ref_Var = { element1, element2, element3, etc };
```

Example:

```
int arr[] = { 1, 2, 3, 4, 5 };
```

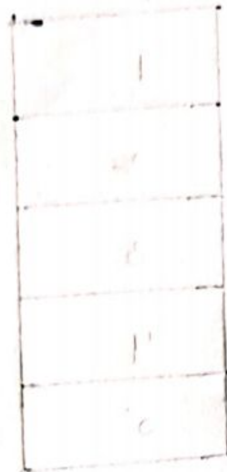


Example-program:

```
public class Demo3
{
    public static void main(String[] args)
    {
        System.out.println("main Starts");
        int[] a = new int[5];
        a[0] = 10;
        a[1] = 20;
        a[3] = 40;
        a[4] = 50;
        System.out.println(a[0]);
        System.out.println(a[1]);
```

```
        System.out.println(a[2]);
        System.out.println(a[3]);
        System.out.println(a[4]);

        System.out.println("Main ends");
    }
}
```



Output:

```
main starts
10
20
0
40
50
main ends
```

Example-program-2:

```
public class Demo3
{
    public static void main(String[] args)
    {
        int[] a = new int[5];
        a[0] =10;
        a[1] = 20;
        a[3] = 40;
        a[4] = 50;
        for(int i=0; i<a.length; i++)
        {
            System.out.println(a[i]);
        }
    }
}
```

Output:

10
20
0
40
50