OOM

Object Oriented Methodology-lecture notes

PREPARED BY:
Biswa Bandita Mohanti

Object Oriented Programming (OOPs)

CONCEPT Dt: 31/7/2019

Programming Language: It is the instructions that a computer can understand

⇒ A computer is bascially a collection of circuits, that use current as a means to do certain task.

⇒ Machine Language to ~~do~~ communicate with a computer.

Examples of Programming Language: C, C++, Java, Python, FORTAN, COBOL, (Common (Formula Translation) Business oriented language), HTML, (Hyper Text Markup Language),

OBJECT ORIENATED PROGRAMMING:

⇒ Object means a real world entity such as pen, chair, table, watch, computer, etc.

⇒ OOPs is a methodology to design a program using classes and objects.

⇒ It simplifies software development and maintenance by providing some concepts:-

⇒ Object

→ Class

⇒ Inheritance

→ Polymorphism

→ Abstraction

→ Encapsulation

OR → Data Hiding

⇒ OOPs is defined as an approach that provides a way of modularising programs by creating partition memory area for both data and function.

⇒That can be used as templetes, for creating copies of such modules on demand.

OOPs CONCEPTS AND TERMINOLOGY:-

~~OOPs~~

★**OBJECT:** Any entity that has state and behaviour is known as object. Examples are Table, Student, etc. It can be physical and logical.

⇒An object can be defined as an instance of a class.

⇒An object can communicate without knowing the details of eachothers data or code.

⇒Object contains an address and ~~text~~ takes of some space in memory.

⇒ØA 'Dog' is an object because it has states like colour, name, etc.

⇒ as well as behaviour like barking, etc.

★**CLASS:** ⇒Class is a collection of object, ~~and~~

⇒It is a logical entity

⇒A class can also be defined as blueprint, from which we can create individual objects,

⇒Class doesnot consume any space.

★**INHERITANCE**

⇒When one object acquires all the properties and behaviour of parent object, it is known as inheritance.

⇒It provides code reusuability.

(Reusability means we can add additional feature to an ~~execution~~ executing class without modifying it).

⇒ In Java, which one class is a allowed to inherit the features of another class.

Super class: It is a base class or parent class.
Subclass: It is a derived or chid class.

★POLYMORPHISM

⇒ If one task is performed in different ways it is known as polymorphism.

⇒ Example - To convience the customer differently

⇒ In Java, we use, this method to over loading or over riding.

★ABSTRACTION

⇒ Hiding Internal details and showing functionalities is known as abstraction.

⇒ For example: Phone Call, we don't know the internal processing.

⇒ In Java, we use abstract class and the interface to achieve abstraction.

★ENCAPSULATION

⇒ Binding or wraping code and data together into a single unit is called encapsulation.

⇒ For example: Capsule is wrap with different medicines.

⇒ In Java, Java bean is fully encapsulated class because all the data members are private.

Benefits of OOPs

⇒ Improve Productivity

Application Development is facilated by re-use of existing component which can gradely improve the productivity and facilate rapid delivery.

⇒ Deliver high quality system

The quality of the system can be improved as the system is build up in the component manner with the use of existing component which are well tested and well proven.

⇒ Lower, Maintenance, Cost

The associated property of traceability of OOM can help to ensure the impact of change is localised and the promblem area can be easily traced. As a result, the maintenance cost can be reduced.

⇒ Facilated Reuse

With this approach a computer system can be develop on a component basis that enables the effective reuse of existing component.

Application of OOPs

i) User interface design such as windws
ii) Real time system design
iii) Simulation and Modeling System
iv) Object Oriented Database
v) AI (Artificial Intelligence) and Expert System
vi) Neural Network and parallel programming.
vii) Office Automation System
viii) Client Server System

Procedural Oriented Programming (POP)	Object Oriented Programming (OOP)
⇒ In procedural programming program is divided into small part called function.	⇒ In object oriented programming, ~~object~~ program is divided into small part called object
⇒ POP follows top-down approach	⇒ OOP follows bottom ~~of~~-up approach.
⇒ There is no access specifier in POP.	⇒ OOP have access specifier like private, public, protected, etc.
⇒ Adding, new data and function is not easy	⇒ Adding, new data and function is easy.
⇒ POP does not have any proper way ~~any~~ ~~pro~~ for hiding data so it is less secured.	OOP provides data hiding so it is more secure.
⇒ In, Procedural Programming overloading is not possible	⇒ In OOP, overloading is possible.

⇒ In POP, function is more important than data Examples: C, FORTRAN, PASCAL,	⇒ In OOP, data is more important than function. Ex: C++, Java, Python

Ch-2 Introduction to JAVA

What is Java?

Java is purely object-oriented language developed by Sun Micro-system by James Gosling and others in the 1991, June.

⇒ The Java platform was intially developed to address the problems of building software for network consumer electronic devices it was designed to support multiple host architecture and to allow secure delivery of software components

⇒ Before Java Language, the software for this consumer electronic device such as washing machine, microwave oven and micro-controller was developed by C, C++.

⇒ The original name of Java, which was released in the year 1991 was known as "OAK" which is a tree name.

⇒ In the year 1996, JDK 1.0 was released. The software Java contains three categories:

i) JSE (Java Standard Edition)
ii) JEE (Java Enterprise Edition)
iii) JME Java Micro Edition

JSE: It is used for developing or standloner application. It is popularly known as Core Java.

JEE: It is used for developing web application or enter

JME: It is used for developing embedded for mobile wireless application.

Features of Java

1.

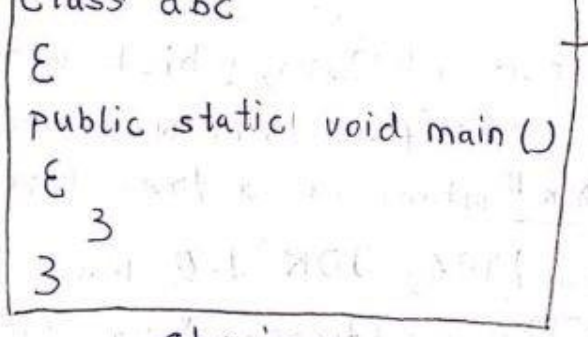
⇒ Java is simple and easy to learn,
⇒ It is object-orienated,
⇒ It is platform ~~ide~~independent,
⇒ It is portable,
⇒ It is secured,
⇒ It is multithreaded,
⇒ It has high-performance
⇒ It is distributed

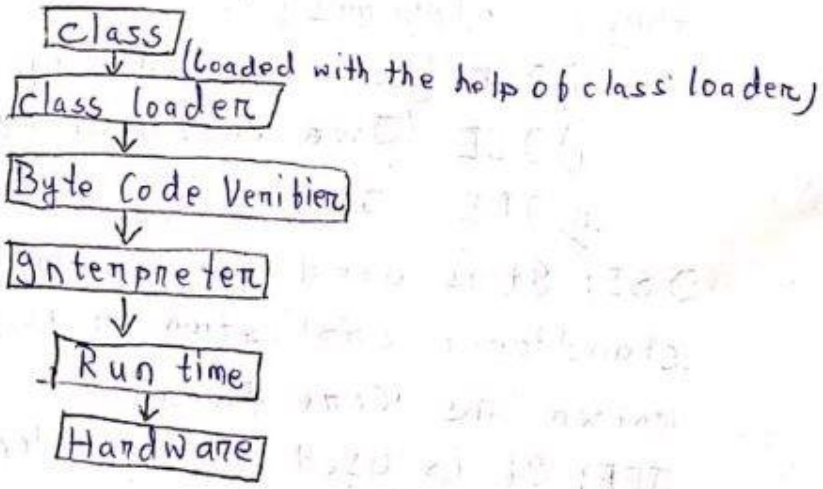
Execution Model of Java

Compile Time :- Java Code → Bytecode

```
class abc
{
public static void main ()
{
  3
3
```

abc.java





Dt-03/08/19

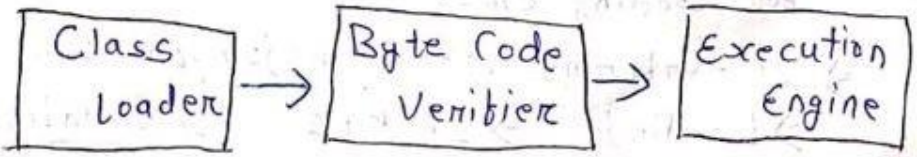
Application of Java

⇒ Web application
⇒ Standalone application
⇒ Enterprise application
⇒ Mobile application

The Java Virtual Machine (JVM)

JVM is a enginea that provides run time environment to drive the java code or application.

⇒ It converts java byte code into machines language
⇒ JVM is a part of JRE (Java Run Environment).
⇒ In other programming languages, the compiler produces machine code for a particular system. However, Java Compiler produces code for a Virtual Machine known as Java Virtual Machine
⇒ First, Java Code is compiled into byte code. This byte code gates interpreted on different machines between host system and java source, byte code is an intermediary language.
⇒ JVM is responsible for allocating memory space.

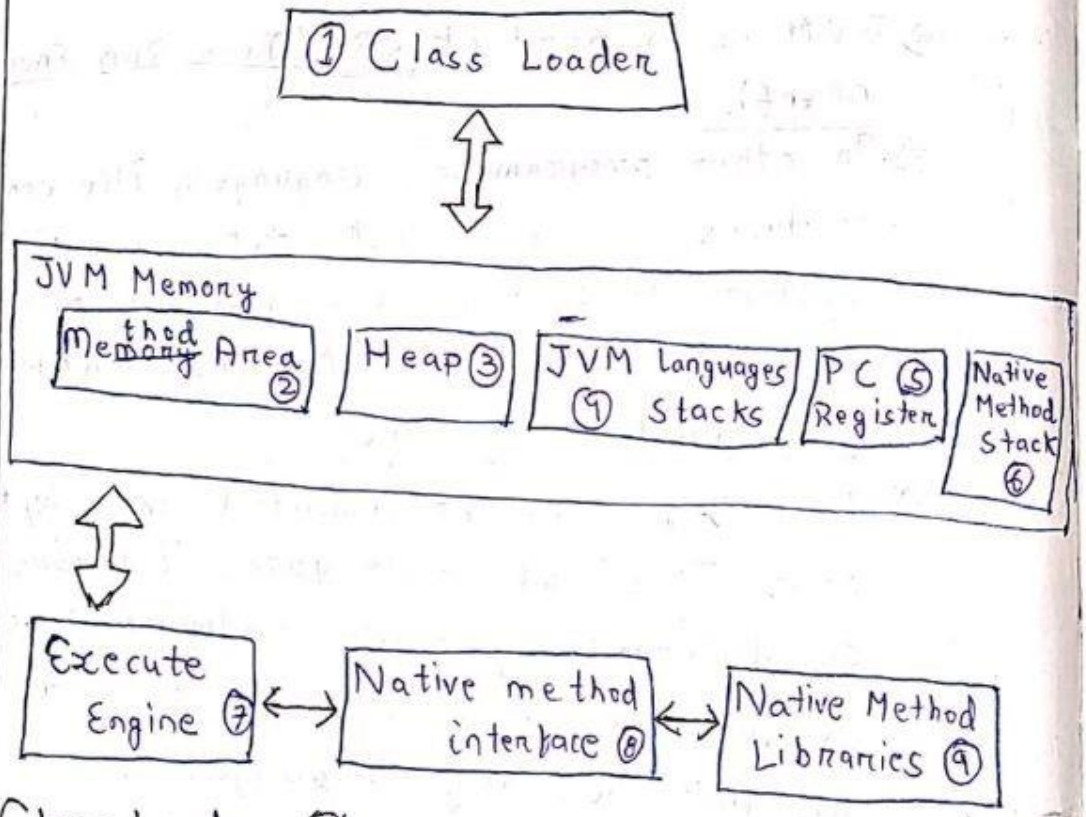


JVM Architecture

⇒ ~~Java~~ JVM is an abstract machine. It is a specification that provide run time environment in which java byte code can be executed.

⇒ JVM are available for many hardware and software platform (i.e. pJVM is platform dependent)

⇒ It contains class loader, memory area execution engine, etc.



① Class Loader ②

⇒ The Class loader is a sub-system used for loading class file.

⇒ It performs three major function i.e:

→ Loading → Linking → Intialisation

② Method Area

⇒ JVM method Area stores class structures like metadata, the constant run time ~~pool~~ pool, and the code for method.

③ Heap

⇒ All the objects they are related instance variables, and arrays are stored in the heap.

⇒ This memory is common and share accross multiple threads

④ JVM Language Stacks

⇒ Java Language stacks store local variable, and its ~~pascal~~ partial results ⇒ Each thread has its own JVM stack, created simultaneously has the thread is created.

⇒ A new frame is created, whenever a method is ~~involve~~ invoked, and it is deleted when method innvocation process is completed.

⑤ PC Register

⇒ PC register store the address of the java virtual machine instruction which is currently executing, in Java, each thread has its separate PC register,

⑥ Native Method Stack

⇒ Native Method stack hold the instruction of native code depends on native libary.

⇒ It is written in another language insted of Java.

⑦ Execution Engine

⇒ It is a type of software used to text hardware, software or complete system.

⇒ The text execution engine never carries, any information about the texted product.

⑧ Native Method Interface

⇒ Native Method Interface is a programming framework it allows java code which is running in a JVM to call by Libraries and native application.

⑨ Native Method Libraries

⇒ Native ~~Li~~ Method Libraries is a collection of native libraries (C, C++) which are needed by the execution engine.

~~Variable~~

A First Java Program

```
class jpro1
{
public static void main(String args[])
{
System.out.println("Hello");
System.out.println("Sumit");
}
}
```

Class : Class keyword is used to declare a class in java.

public : public is an access modifier which represent visibility. It means it is visible to all.

static : static is a keyword if we declare any method as static it is known as static method.

void : It is the intertype of the method. It means it does not return any value.

main : main represent the static point of the program

String args[] :- It is used for command-line argument

System.out.println("---") - It is used to print statement. Here, System is a class and out is object of print stream class, println is the method of print stream class.

Variables

⇒ A variable can be thought of as a container which hold value for you during the life of a java program.

⇒ Every variable is assign a datatype which designates the type and quantity of value it can hold.

⇒ In order to use a variable in a program, we need to perform two steps:

i) Variable Declaration

ii) Variable Initialisation

Variable Declaration:

To declare a variable we must specify the data type and give the variable a unique name.

```
int count;
 ↓    ↓
Type  name
```

Examples: int a,b,c;
float p;
char a;
double d;

Variable Intialisation:

To initialise a variable, we must assign it a valid value.

Example: count = 100;
pi = 3.14f;
a = 'v';

We can combine variable intialisation and declaration

Example: int count = 100;

int a=2, b=4, c=6;

float pi=3.14f;

char a='v';

Types of Variables

⇒ Primitive Variables: It is used to represent primitive value.

Example: int a = 10

⇒ Reference Variable: It is used to refer objects

Example: Abc a1 = new Abc();

Abc → class, a1 → object, new → Refer, Abc() → Object (new class)

In Java, there are three types of variables: ⇒ Instance Variable
→ Static Variable
→ Local Variable

Instance Variable

⇒ The value of variable is varied from object to object.

⇒ It cannot be accessed from static area directly

⇒ It is also known as object level variable.

Static Variable

⇒ The value is not ~~valid~~ varied from object to object.

⇒ It is also known as class level variables.

Local Variable

⇒ It is created inside a block or constructor, method

⇒ Intialisation is necessary before using local variable.

Instance Variable

→ Scope of object

→ Should be declare within the class directly, But outside any block, method or constructor

→ Intialisation is not mandatory.

Example:

```
class Abc
{
int i=100;
public staic void main (String args[])
{
Abc obj = new Abc();
System.Out.println(obj.i);
}
```

Static Variable

→ Scope of class

→ Variable not modify object to object

→ Intialisation is not mandatory

Example

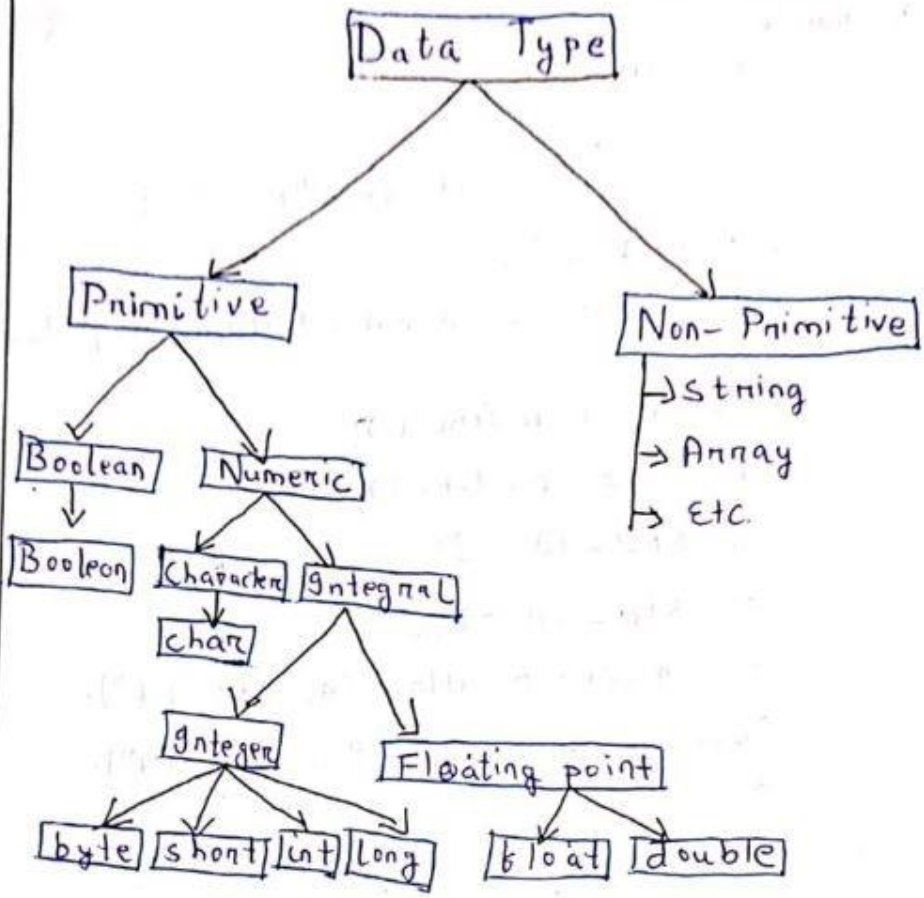
```
Class Abc
{
static string college = "BOSE";
int std-id;
public static void main (String args[])
{
Abc a1 = new Abc();
Abc a2 = new Abc();
a1.std-id = 1;
a2.Std-id = 2;
System.out.println("a1.std-id");
System.out.println("a2.std-id");
}
}
```

Local Variable

Example;

```
public class Dog
{
public void putAge()
{
int age=0; // local variable
age= age+6
System.out.println("Dog age is:"+age);
}
public static void main (String args[])
Dog d = new Dog();
d.putAge();
}
}
```

Dt- 08/08/2019



Primitive Data Type

⇒ Primitive Data Type are pre defined and available within the java language.

⇒ Primitive Value do not share state with other primitive value.

⇒ There are eight primitive data type boolean, char, byte, short, int, long, float, double.

Boolean

⇒ The boolean data type has two possible values, either true or false.

⇒ Default Value is false.

⇒ They are usually used for true or false condition.

Example:

```
class Boolean
{
public static void main (String args[])
{
boolean flag = true;
System.out.println(flag);
}
}
```

Output - True

Character

⇒ It is a 16-bit unicode character.

⇒ The minimum value of character data type is '\u0000' (0)

⇒ The maximum value of character data type is '\uffff' (f)

⇒ Default Value is '\u0000'

Example:

```
class char
{
public static void main (String args[])
{
char letter = '\u0051';
System.out.println(letter);
}
}
```

Output - Q

We get the value Q, because the unicode value Q is 51.

Example

```
class char
{
public static void main (String Args[])
{
char letter 1 = 'q';
System.Out.println("letter 1");
char letter 2 = 65;
System.out.println("letter 2");
}
}
```

Output - A

Byte

=> The byte data type can have value from (-128 to 127. (8 bits ~~and~~ signed 2's Complement integer)

=> It is used instead of int or other integer data type to save memory if its certain that the value of a variable will be within -128 to 127.

=> Default value is 0.

Example

```
class Byte
{
public static void main (String args[])
{
byte range;
range = 124;
System.out.println(range);
}
}
```

Output

124

Short

=> The short data type can have values from -32768 to 32767. (16 bits signed 2's complement).

=> It is used instead of other integer data types to save memory if its certain that value of a variable will be within -32768 to 32767.

=> Default Value is 0.

Example:

```
class short
{
public static void main(String args[])
{
Short temperature
temperature = -200;
System.out.println(temperature);
}
}
```

Output: -200.

Integer

=> The int data type can have value from -2^{31} to $2^{31}-1$ (32 bits signed 2's complement integer).

=> If we are using java 8 or later, we can use unsigned 32 bits integer with minimum value of 0 and maximum value of $2^{32}-1$.

=> Default Value is 0.

Example:

```
class Int
{
public static void main(string Args[])
{
int range =-4250000;
System.out.println(range);
 3
 3
```

Output: -4250000

Long

⇒ The long data type can have values from -2^{63} to $2^{63}-1$ (64-bit signed two's complement integer).

⇒ If you are using Java 8 or later, you can use unsigned 64-bit integer with minimum value of 0 and maximum value of $2^{64}-1$.

⇒ Default Value is 0

Example:

```
Class Long
{
public static void main(string args[])
{
Long amount = 1234567891;
System.out.println(amount);("long value =" + amount);
 3
 3
```

Output: Long value = 1234567891

Float

⇒ The float data type is a single precision 32-bit floating point.

⇒ It should never be used for precise values such as currency.

⇒ Default value: 0.0 (0.0f)

Example:

```
class Float
{
public static void main (String args[])
{
float interest = 12.25f;
System.out.println("interest = " + interest);
}
}
```

Output: interest = 12.25

Double

⇒ The double data type is a double precision 64-bit floating point.

⇒ It should never be used for precise values such as currency.

⇒ Default Value : 0.0 (0.0d)

Example:

```
class Double
{
public static void main (String args[])
{
double value = 12345.234d;
```

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

Non Primitive Data Types

The non primitive data types are created by the programmer during the coding process, they are known as the "reference variables" or "object variables" as they refer to a location where data is stored.

Java String

String are basically a collection of characters, they cannot change once they ~~ha~~ are created

Example:

```
import java.util.Scanner;
public class String Reverse
{
public static void main(String args[])
{Scanner in = new Scanner (System.in);
System.out.println("Enter the string:");
String str = in.nextLine();
String reverse = new StringBuffer(str).reverse().
                                    ~~toStr~~ toString();
System.out..println("String after reverse:"+reverse);
}
}
```

Output Enter the String
java
String after reverse: avaj

Java Array and Object

⇒ An array in Java is a group of variables that are similar in nature and we can allocate them dynamically.

⇒ Their size has to specify in int, and their length can be found ~~out~~ by .length member length. Their indexing always starts with zero.

Example:

```
public class Test {
{
public static void main (String args[])
{
Object[] objectArray = new Object[3];
objectArray[0] = "Hello World";
objectArray[1] = new Integer(10);
objectArray[2] = new Character('a');
System.out.println(objectArray[0]);
System.out.println(objectArray[1]);
System.out.println(objectArray[2]);
}
}
```

Output:- Hello World
10
a

Literals in Java

An constant value, which can be assigned to the variable is called literals or constants.

Example: int x = 100;

Here, 100 is a literals / or constant.

Types of Literals

→ There are five types of Literals in Java

⇒ Integer Literals (Numeric Literals)
⇒ Boolean Literals (Numeric Literals)
⇒ Character Literals (Numeric Literals)
⇒ String Literals
⇒ Float point Literals (Numeric Literals)

Integer Literals

⇒ The integer literal can be used to create int value

⇒ They can be used to intialise the group of data types like byte, short, int and long.

⇒ The number without decimal is the integer literal the java compiler treats all the integer literals as int.

⇒ We can assign different values to integer literals they are: ~~decimal~~

→ Decimal Literal - The decimal Literal is not prefixed with zero '0' or '0b' or '0x'

Ex- int d = 10;

→ Binary Literal - The binary literal is prefixed with '0b'.

Ex- int x = 0b1010; // the value in decimal is 10.

→ Hexadecimal Literal - The hexadecimal literal is prefixed with '0x'

Ex- int a = ~~0x1f~~ 0x17 // the value in ~~he~~decimal is 23

→ Octal Literal- The octal literal are prefixed with zero.

Ex- int y = 023; // the value in decimal is 19.

→ Long Literal- The long literal is are suffixed with 'l' or 'L'.

Ex- i) long b = 345627806; // the value in decimal is 345627806

ii) long c = 345627 80L; // the value in decimal is 3456278.

Boolean Literals

⇒ The boolean literal is used to represent the ~~two~~ values i.e. either true or false.

⇒ We can assign true or false to the variable. Variables are case-senstitive.

Ex- boolean flag 1 = true;
boolean flag 2 = false;

Character Literals

⇒ The character literal is a 16 bits unicode character. Where it is enclosed in single ~~codes~~ quotes.

⇒ It used primitive data type char.

Ex- char c = ~~'8'~~ '\u0065'

⇒ Some of the example to represent the character literals are:

Escape Sequence	Meaning
\\	Back Slash
\a	Alert
\b	Back Space
\'	Single quotation mark
\"	Double quotation mark
\n	New line
\t	Tab
\d	Octal
\x d	Hexadecimal
\ud	Unicode character.

String Literals

⇒ String Literal are the sequence of character which are enclosed in double quotes

⇒ The string literal should occur in single line

Ex - String str = "BOSE";

Floating point

⇒ The float point literal can be represent decimal value with a fractional component

⇒ These literals are double data type.

⇒ These literal contain the fractional part and if the floating point literal is suffixed with letter f or F then it is float type.

⇒ These Literal include the float and double data type

⇒ In floating point literals double is the default data type

⇒ To represent the float literal f is suffixed and to represent the double literal d is suffixed.

Example: float x = 2.7f
double x = 54.8888d

Casting

Dt- 13/8/19

Casting means taking an object of one particular type and "turning it into" another object type this process is called casting a variable.

Type Casting in Java

⇒ Assigning a value of one type to a variable of another type is known as type casting.

Example: int x = 10;
byte y = (byte) x;

⇒ In Java, Type Casting is classify into two types: ⇒ Widening Casting (Implict)

⇒ Narrowing Casting (Explicitly)

Widening or Automatic Type Conversion

byte → short → int → long → float → double

widening.

Automatic Type Casting take place.

When,
- the two types are compatively
- the target type is larger than the source type

```
public class text
{
public static void main (String args[])
int i=100;
long l = i; 100
float f = l; 100
System.out.println(" int value = "+i);
System.out.println(" long value = "+l);
System.out.println(" float "= "f);
```

O/P → 100
100
100.00

Nannowing on Explict type convention

double → float → long → int → short → byte

explicity →

When you are assigning a largen type value to a variable of smallen type, then you need to perform explicit type casting.

Example:

```
public class Text
{
public static void main(String args[])
{
double d = 100.04;
long L= (long)d;
int i = (int)L;
```

```
System.out.println ("Double Value" +d);
System.out.println ("Long Value"+L);
System.out.println ("Int Value" + i);
3
3
```

OUTPUT

Double Value 100.04

Long Value 100

Int Value 100

JAVA Expression

Expressions consist of variables, operators, literals and method calls that evaluates to single value.

E.g.1
```
int score;
score = 90;
```

Here, score = 90 is an expression that returns (int)

E.g.2
```
Double a=2.2, b=3.4, result;
result = a+b-3.4;
```

Here, (a+b-3.4) is an expression,

E.g.3
```
if(num1 == num2)
    System.out.println ("Num1 is larger then num2");
```

Here, (num1==num2) is an expression that returns "Boolean", Similarly, "num1 is larger than num 2" is a string expression.

JAVA Statements

Statements are everything that make up a complete unit of execution.

E.g. `int score = 9*5;`

Here, (9*5) is an expression that returns 45, and int score = 9*5; is a statement.

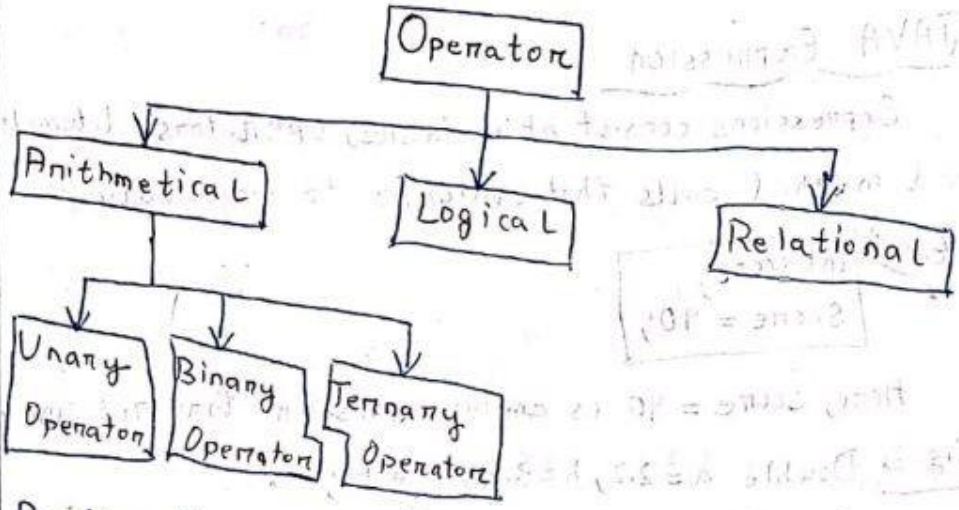
→ Expressions are part of statements,

OPERATOR

Dt- 17/8/19

An operator is a symbol or special character that acts on the operand to give a specific output.

E.g. int y = x+3;
↙ ↙
Operand Operator



(1) Arithmetical Operators → The operators which are applied to perform arithmetical calculations is a program are known as "arithmetical operator."

E.g. '+', -, *, / and %.

(i) Unary Operator :-

An operator of arithmetical, which is applied to single operand is known as Unary Operator.

E.g. +, -, ++, --, etc.

* **Unary (+) Operator :**

→ This operator is applied before the operand,

→ It is just applied as a pointer to a variable which results in the same value of the variable,

E.g. a = 8 , a = -10

+a = 8 +a = -10

* **Unary (-) Operator :**

→ This operator is used in the same way as unary plus (+).

→ It is also applied before the operand.

→ Unary minus (-) reverse the sign of an operand,

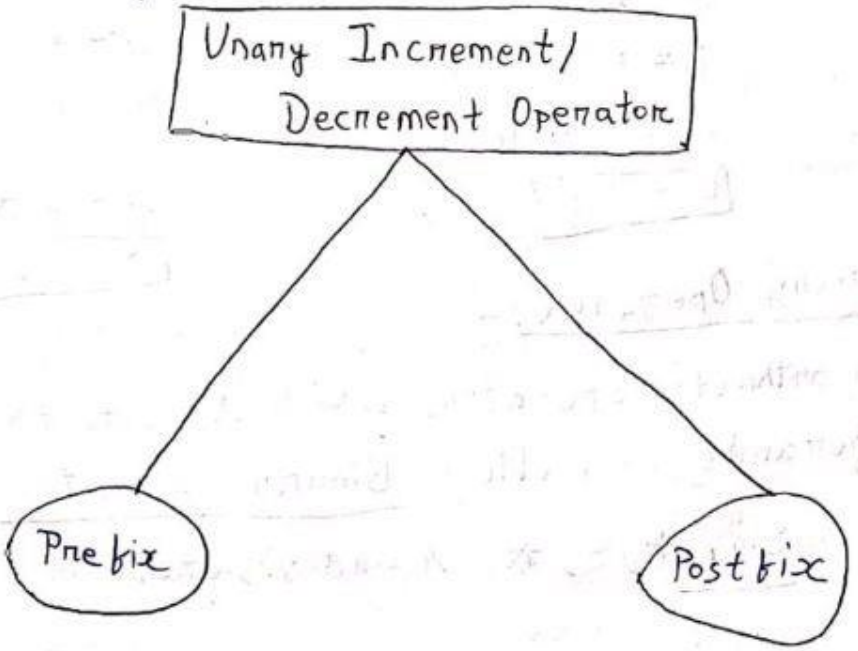
E.g. a = 4 , a = -3.6

-a = 4 -a = 3.6

* **Unary Increment & Decrement Operator ;**

→ Unary increment operator (++) increases the value of an operand by one.

→ Unary decrement operator (--) decreases the value of an operand by one.



i) Prefix :-

→ When increment / decrement operators are applied before the operand, they are known as prefix operators.

→ This operator works on principle of 'CHANGE BEFORE ACTION'.

E.g. Prefix Increment

```
p = 5;
p = ++p * 4;
p = 6 * 4
p = 24
```

Prefix Decrement

```
d = 11;
t = 4 + (--d);
t = 4 + 10
t = 14
```

ii) Postfix :-

→ This unary operator is used after an operand.

→ This works on the principle of 'CHANGE AFTER THE ACTION'.

E.g. Postfix Increment

```
p = 5;
p = p++ * 4;
p = 5 * 4
p = 20
```

Postfix Decrement

```
p = 5;
m = p-- * 4;
m = 5 * 4;
m = 20
```

ii) Binary Operator :-

→ An arithmetic operator, which deals with two operands is called Binary operator.

E.g. +, -, *, / and %, etc.

Operator	Symbols	Format	Result if a=22, b=5
Addition	+	a+b	27
Subtraction	-	a-b	17
Multiplication	*	a*b	110
Division	/	a/b	4
Modulus / Remainder	%	a%b	2

(iii) Ternary Operator (Conditional Assignment):-

→ Ternary operators deal with three operands,

→ They are also called "conditional assignment statement".

Syntax: Variable = (test expression) ? Expression 1 : Expression 2

Eg.
```
a=5; b=3;
max = (a>b) ? a:b;
```
Here, the value 5 is stored in max as a>b is true.
```
min = (b>a) ? a:b;
```
Value 3 is stored in min as b>a is false.

② Logical Operators → Logical operators yield true/false depending upon the outcome of different expressions

→ Logical Operators are AND (&&), OR (||), NOT (!),

Precedence of Logical Operators:-

NOT (!)
↓
AND (&&) → OR (||)

Logical Operator	Symbol	Format
AND	&&	(a>b) && (a>c)
OR	||	(a==b) || (a==c)
NOT	!	! (a == b)

* Logical AND (&&)

① 5>3 && 3<5 → True

② 6==6 && 3>0 → True

③ 5!=5 && 4==4 → False

* Logical OR (||)

① 5>4 || 8>12 → True

② 3>7 || 5>=4 → False

* Logical NOT (!)

① !(8>3) → False

② !(3<0) → False

③ Relational Operators → These operators are used to show the relationship between the operands. These operator results in true / false.

Symbol	Meaning	Format	Result if a=10, b=6
<	Less than	a<b	false
>	Greater than	a>b	true
<=	Less than or equal to	a<=b	false
>=	Greater than or equal to	a>=b	true
==	Equal to	a==b	false
!=	Not Equal to	a!=b	true

(4) Assignment Operator

Assignment operators are used to assign the value of an expression to a variable. We have seen the usual assignment operator, '='.

Example:

```
int age;
age = 5;
```

Here, the assignment operator assigns the value on its right to the variable on its left. Here, '5' is assigned to the variable age using '=' operator.

Operator	Example	Equivalent to
+=	x+=5	x=x+5
-=	x-=5	x=x-5
*=	x*=5	x=x*5
/=	x/=5	x=x/5
%=	x%=5	x=x/5
<<=	x<<=5	x=x<<5
>>=	x>>=5	x=x>>5
&=	x&=5	x=x&5
^=	x^=5	x=x^5
|=	x|=5	x=x|5

Program

```
class AssignmentOperator
{
public static void main(String args[])
{
int a, b;
a = 5; // Assigning 5 to a
System.out.println(a);
b = a; // Assigning value of variable b to a
System.out.println(b);
}}
```

Bitwise Operator

⇒ These are special types of operators, which perform operations on bit level of the operands.

⇒ Bitwise operators are binary operators.

Operators	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise Complement
<<	Left Shift
>>	Right Shift
>>>	Zero fill right shift
<<=	Left shift assignment
>>=	Right shift assignment
>>>=	Zero fill right shift assignment

Bitwise AND (&)

This operation results in high (1) if both the operands are high, otherwise low (0)

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

Bitwise OR (|)

This operator results in high (1), if either of two operands is high (1), otherwise low (0).

a	b	a|b
0	0	0
0	1	1
1	0	1
1	1	1

Bitwise NOT (!)

This operator results in high (1) if bit operand is low (0) and vice-versa.

a	!a
0	1
1	0

Bitwise XOR (^)

This operator results in low (0) for the same values of the operands. The outcome is high (1) for the different values.

a	b	a^b
0	0	0
0	1	1
1	0	1
1	1	1

E.g.

```
public class bitwise
{
public static void main(String args[])
{
int a,b;
a=12; b=10;
System.out.println("(a&b)=" +(a&b));
System.out.println("(a|b) =" +(a|b));
System.out.println("(a^b) =" +(a^b));
System.out.println("(a<<2) =" +(a<<2));
System.out.println("(12>>2) =" +(12>>2));
System.out.println("(14>>>2)=" +(14>>>2));
3
3
```

Output:-

(a&b) = 8

(a|b) = 14

(a^b) = 6

(a<<2) = 48

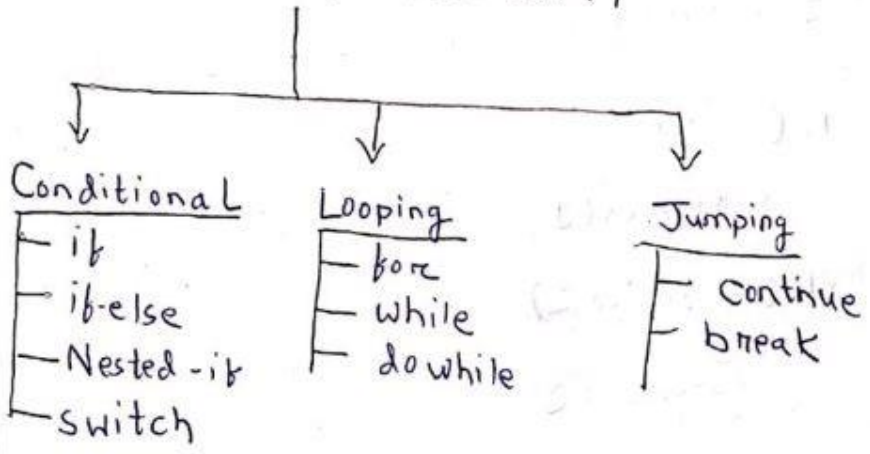
(12>>2) = 3

(14>>>2) = 3

Precedence of Operators

Operators	Order of Operation
(), []	1
++, --, ~, !	2
*, /, %	3
+, -	4
>>, >>>, <<	5
>, >=, <, <=	6
==, !=	7
&	8
^	9
|	10
&&	11
||	12
=	13
=	14

CONTROL STATEMENT



⇒ 'if' is referred to as bi-directional branching.

Conditional Statement :-

1) if statement :

Syntax : if (condition)
{
statement 1;
statement 2;
}

e.g.

```
public class if
{
public static void main(String args[])
{
int n=70;
if(n<100)
{
System.out.println("Number is less than 100");
}
}
}
```

O/P → Number is less than 100.

2) if and only if statements (multiple use of it)

Syntax :-

```
if (condition 1)
    statement 1;
if (condition 2)
    statement 2;
if (condition 3)
    statement 3;
```

E.g

```
public class jpno2
{
public static void main (String args[])
{
int n=70;
if(n<100)
System.out.println("Number is less than 100");
if (n>50)
System.out.println("Number is greater than 50");
}
}
```

O/P → Number is less than 100
Number is greater than 50

3) if-else statement :-

Syntax: if (condition)
{
Statement 1;
}
else
{
statement 2;
}

E.g

```
public class jpno3
{
public static void main(String args[])
{
int n = 120;
if (n<50)
{
System.out.println("Number is less than 50");
}
```

```
else
{
System.out.println("Number is greater than or equal
                                        to 50");
}
}
}
```

4) if-else-if statement (if-else-if ladder):-

```
Syntax: if (condition 1)
        {
        statement 1;
        }
        else if(condition 2)
        {
        statement 2;
        }
        else if(condition 3)
        {
        statement 3;
        }
        else
        {
        statement 4;
        }
```

E.g.

```
public class jpro84
{
public static void main(String args [])
{
int n = 1234;
if (n<100 && n>=1)
{System.out.println("A 2-digit number");
else if (n<1000 && n>=100)
System.out.println("A 3-digit number");
else if(n<10000 && n>=1000)
{System.out.println("A 4 digit number");
```

```
else
System.out.println("No. is not between 1 & 9999");
}
}
```

O/P -> A 4-digit number

5) Nested if statements :-

Syntax :
```
if (condition 1)
{
  if (condition 2)
  statement 1;
  else
  statement 2;
}
else
{
  if (condition 3)
  statement 3;
  else
  statement 4;
}
```

E.g
```
if (a>b)
{
if(a>c)
 max=a;
else
 max=b;
}
else
 {
  if (b>c)
  max =b;
  else
   max = c;
  }
System.out.println ("Maximum =" +max);
```

6) Switch Statement

Syntax:-
```
switch(expression)
{
case value 1:
    statement 1;
    break;
case value 2:
    statement 2;
    break;
case value 3:
    statement 3;
    break;
default;
    statement 4;
    break;
}
```

E.g.
```
switch(n)
{
case 1:
System.out.println("Good Morning");
break;
case 2:
System.out.println("Good Afternoon");
break;
Case 3:
System.out.println("Good Evening");
break;
case 4:
System.out.println("Good Night");
break;
default;
System.out.println("Wrong choice");
}
```

LOOPING STATEMENTS

1) **for loop :-**

Syntax :
```
for (intial value; condition; expression)
{
statement;
}
```

e.g.
```
for (int i=2; i<=20; i++)
{
System.out.println (i);
}
```

2) **while loop :-**

Syntax :
```
intial value;
while (condition)
{
statement;
}
```

e.g.
```
int i=2;
while (i<=20)
{
System.out.println(i);
}
```

3) **Do-while loop :-**

Syntax :
```
intial value;
do
{
statement;
}
while (condition);
```

Eg:
```
int i=2;
do
{
System.out.println(i);
}
while(i<=20);
```

For loop	While loop
→ Used for fixed no. of iteration.	→ Used for unknown no. of iterations.
OR	OR
For loop is used when we know the number of steps previously.	While loop is used when we don't know the no. of steps previously.
→ In this loop, intial value, condition and step value (increment / decrement value) are written in a single statement.	→ In this loop, intial value, condition, step value are written in 3 ~~digit~~ different statement.

Similarity :-

Both for and while loop are "entry-controlled loop."

Do-while loop	While loop
→ It is an exit-controlled loop	→ It is an entry controlled loop.

Similarity :- Both loops are executed for unknown no. of times.

JUMPING STATEMENTS

Break	Continue
It is used to stop the execution of a block in a program	→ It is used to resume the execution of a block from next iteration.
→ It makes an early ending of a block.	→ It makes a delay ending of the block

OBJECT AND CLASS

What is class?

Class is an entity that determines how an object will behave and ~~how~~what the object will contain.

⇒ In otherwords, it is a blueprint or a set of instruction to build a specific type of object.

Syntax

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

What is an object?

⇒ An object is nothing but a self-contained component which consist of method and properties to make a particular type of data useful.

⇒ Object determine the behaviour of the class.

⇒ From a programming point of view, an object can be a data structure, a variable or a function. It has a memory-location allocated the object is designed as hiearachies.

Syntax

```
ClassName ReferenceVariable = new className();
```

Object	Class
⇒ An object is a specific of a class. Software object are often used to model real world objects we find in everyday life.	⇒ A class is a blueprint or prototype that define the variables and the method (functions) common to all object of a certain kind.

CONCEPT OF JAVA CLASSES AND OBJECT WITH AN EXAMPLE

⇒ Let takes an example of developing a pet management system, specially meanedt for dog.

⇒ We will need various information about the dogs like different breeds of dog, the age, the size, etc.

⇒ We need to model real like beings i.e. dogs into software entity or entities.

⇒ We can take different breeds of dog some of the difference we might have listed out may be breed, age, size, colour, etc.

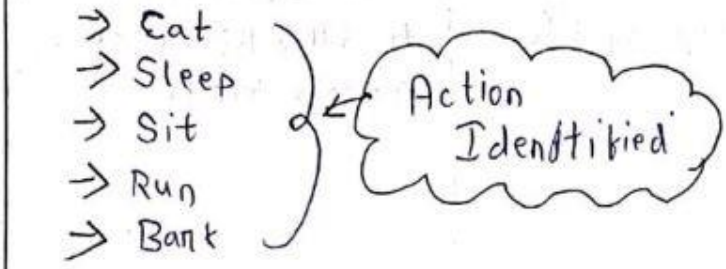
⇒ This characteristics can form a data member for our object.

Common Characteristics

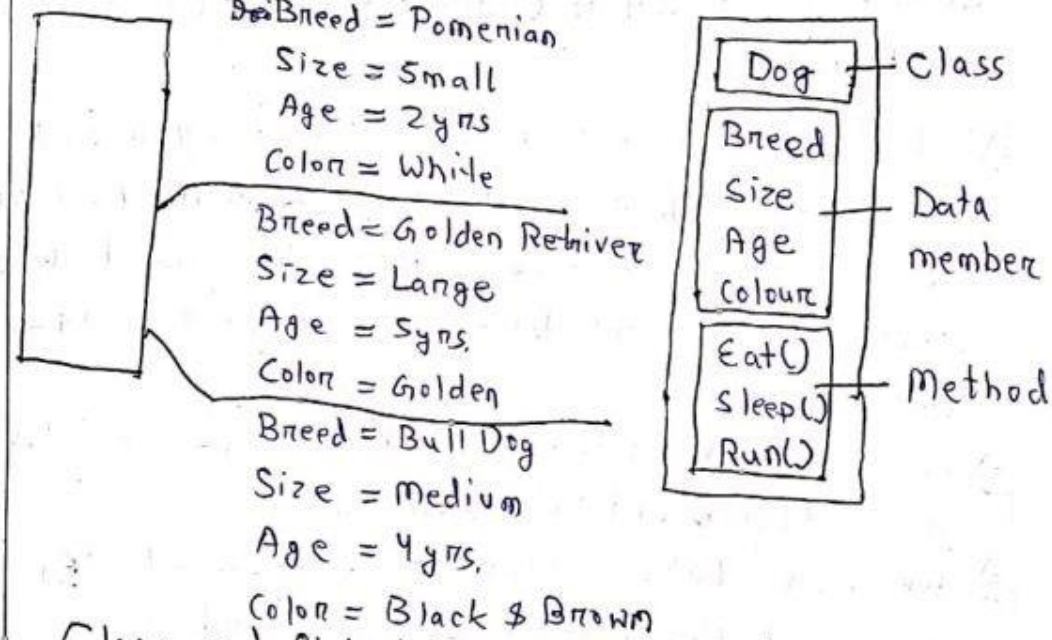
- → Breed
- → Size
- → Age
- → Colour

⇒ Next, list out the common behaviour of this dogs like, sleep, eat, run, ~~etc~~, sit, bark, etc. So, this will be action of our software objects

Common Actions



- Class - Dogs
- Data Member or objects - Size, age, colour, breed, etc.
- Method or Function - Eat, Sleep, Sit and Run.



Class and Object Program

```
// Class Declaration
public class Dog
{
 // Instance Variable
   String breed;
   String size;
   int age;
   String colon;
```

```
//method 1
Void display
{
System.out.println("Breed is: \n" +breed
                   "Size is: \n" + size
                   "Age is: \n" + Age
                   "Colour is: \n" +colour);
}
public static void main (String args[])
{
Dog Pomerian = new Dog();
Pomerian.breed = "Pomerian";
Pomerian.size = "small";
Pomerian.age = 2;
Pomerian.colour = "white";
Pomerian.display();
Dog Golden Retriver = new Dog;
Golden Retriver.breed = "Golden Retriven";
Golden Retriven.size = "Large";
Golden Retriven.age = 5;
Golden Retriven.colour = "Golden"
Golden Retriven.display();
Dog BullDog = new BullDog;
Bull Dog.breed = "Bull Dog";
Bull Dog.size = "medium";
Bull Dog.age = 4;
Bull Dog.colour = "Black & Brown";
Bull Dog.display();
}
}
```

```
public class employee
{
String name;
int age;
String dept;
double salary, basicsalary;
```

Object and Class Example

```
main outside class
void payslip()
{
System.out.println("Name is:\n" + name);
System.out.println("Age is:\n" + age);
System.out.println("Dept is:\n" + dept);
System.out.println("Basic salary is:\n" + basicsalary);
}
public static void main (String args[])
{
employee obj = new employee();
obj.name = "Jane";
obj.age = "23";
obj.dept = "Accounts";
obj.basicsalary = "9500";
obj.payslip();
employee obj1 = new employee();
obj1.name = "Ravi";
obj1.age = 27;
obj1.dept = "HR Dept";
obj1.basicsalary = 15500;
obj1.payslip();
```

```
employee obj2 = new employee();
obj2.name = "Smruti"
obj2.age = 30;
obj2.dept = "Accounts Head";
obj2.basicsalary = 12000;
obj2.payslip();
}
}
```

Methods Declaration :-

A class with only data fields (and without methods that operate on that data) has no life. The objects created by such a class cannot respond to any messages. We must therefore add methods that are necessary for manipulating the data contained in the class. Methods are declared inside the body of the classes but immediately after the declaration of instance variables. The general form of a method declaration is:

```
type methodname (parameter-list)
{
   method-body;
}
```

```
public class cube
{
int length;
int breadth;
int height;
public int getVolume()
{
return (length * breadth * height);
}
}
```

Fields Declaration :-

Data is encapsulated in a class by placing data fields inside the body of the class definition. These variables are called instance variables because they are created whenever an object of the class is instance variables exactly the same way as we declare local variables.

Example:
```
Class Rectangle
{
    int length;
    int width;
}
```

The class Rectangle contains two integer type instance variables. It is allowed to declare them in one line as

```
int length, width;
```

These variables are only declared and therefore no storage space has been created in the memory.

PTO → Method Declaration

class - cube that defines three fields, length, breath, height; Also a class contain member function get volume().

⇒ A method is a block of code which only run when it is called.

⇒ We can pass data known as parameter into a method

⇒ Method are used to perform certain action, and there are also known as functions

Why is Method ?

⇒ Generally, a method call with the help of a object.

→ To reuse code, define the code once, and ~~declare~~ use it many times

Create a Method

A method must be declare within a class. It is defined with a name of the method followed by parenthesis ().

Java provide some pre-defined method such as System.out.println().

But we can also create our own method to perform certain action.

Example:- Create a method inside class

```
public class ABC
{
  static void MyMethod()
  // code to be executed
  {
  }
}
```

MyMethod() is the name of the method.

static means that the method belongs to the ~~MyClass~~ ABC class and not an object of the ~~MyClass~~ ABC class.

void means that this method does not have a return value.

Call a Method

To call a method in Java, write the method's name followed by two parentheses () and a semicolon;

In the following example, myMethod() is used to print a text (the action), when it is called.

Example

Inside main, call the ~~My~~myMethod() method:

```
public class ~~MyClass~~Abc
{
public static void myMethod()
{
System.out.println("I just got executed!");
}
public static void main(String ~~A~~args[])
{
myMethod();
}
}
// Outputs "I just got executed!"
```

A method can called multiple times:

Example

```
public class MyClass
{
static void myMethod()
{
System.out.println("I just got executed!");
}
public static void main(String args[])
{
```

```
myMethod();
myMethod();
myMethod();
}
}
// I just got executed!
// I just got executed!
// I just got executed!
```

Example

```
public class ABC
{
public static void main(String args[])
{
int a=11;
int b=6;
int c = minFunction(a,b);
System.out.println("Minimum value="+c);
}
public static int ~~main~~Fmin Function (int n1, int n2)
{
int min;
if (n1>n2)
  min=n2;
else
    min=n1;
  return min;
}
}
// Minimum Value = ~~(11,6)~~ 11 ~~(11,6)~~ 6.
```

Accessing the class members

Syntax

```
Objectname.Variablename = value;
Objectname.methodname(parameter-list);
```

Difference bet^ Instance Variable (Data) & Class (Static) Variable

Instance Variable	Class (Static) Variable
→ Instance Variable are declared in a class, but outside a method, constructor or any block.	→ Static Variable or Class Variable, are declared with the static keyword in a class, but outside a method, constructor or a block
→ Instance Variable are created when an object is created, with the use of keyword (new), and destroyed when the object is destroyed	→ Static Variable are created when the program is start and destory when the program stop.
→ Instance Variable can be access directly by calling the variable name inside the class	→ Static variable can be access by calling with the class name.
→ Object Reference. Variable name ↳ Syntax	→ Classname. Variablename ↳ Syntax
Instance Variable hold value that must be reference by more than one method, block or constructor, all essential part of an object's state that must be present through out the class.	→ There would only be one copy of each class variable regardless of how many object are created from it.

Example:

```
public class jpro11
{
int myVariable;
Static int datatype= 30;
public static void main(String args[])
{
Abc jpro11 obj = new jpro11();
System.out.println("Value of instance variable"
                                + obj.myVariable);
System.out.println("Value of static variable"
                                + jpro11.data);
}
}
```

O/p -> Value of instance variable : 0
Value of static variable : 30

Constructors

-> Constructor is defined as special type of method ie. used to intialise the object.

-> Constructor name must be same as its class name

-> Constructor must have ~~no~~ no explict return type

```
Class Abc
{
Abc() {
  // Constructor body
  }
}
```

Here, ~~Abc is~~ ~~con~~ Abc() is constructor, it has same name as that of the class and does not have a return type.

How does a Constructor work?

```
Abc obj = new Abc();
```

→ Let say, we have take Abc as class when we create the object of Abc

→ The new keyword, creates the object of class Abc and invokes the constructor to intialise this newly created object

Example

```
public class Hello
{
 String name;
 //Constructor
 Hello()
 {
 this.name = "Java Book";
 }
 public static void main(String args[])
 {
 Hello obj = new Hello();
 System.out.println(obj.name);
 }
 }
```

O/P → Java Book

Types of Constructors

→ There are three types of constructors

- → Default
- → No-Argument
- → Parameterised.

Default or No-Argument Constructor	Parameterised Constructors

⇒ It contains no-parameter argument
⇒ class Abc
{
Abc()
{
// body
}
}
	→ It contains argument or parameters.
	⇒ class Abc
	{
	Abc(int i, int j)
	{
	// body
	}
	}

Default Constructor

When the constructor does not accept any argument or parameter then it is called default constructor.

Parameterised Constructor

When the constructor accept some argument or parameter then it is called parameterised constructor.

Copy Constructor

Copy constructor is a special constructor which takes the class type as its argument.

Syntax

```
Constructor name (class name  object name)
{
variable name = object name. variable name;
}
```

Constructor Overloadding

When more than one constructor is available in a program then it is known as constructor overloading.

Destructor

⇒ Destructor is a special member function whose name is same as the class name

⇒ It is preceded with till (~) symbol.

Syntax

```
class XYZ
{
   ~XYZ()
   {
   }
};
```

⇒ Java doesnot support destructor.

Note: Constructor are called ~~we can~~ when object is created. And Destructor are called when object is erased.

Example:

```
class demo
{
public
int i;
int *p;
demo()
{
p=(int*)malloc(40);
}
~demo()
{
free(p);
}
};
```

Example of Constructor Overloading

```
class Box
{
double width, height, depth;
//constructor is used when all dimensions specified
Box(double w, double h, double d)
{
width = w;
height = h;
depth = d;
}
//compute and return volume
double volume()
{
return width * height * depth;
}
}
```

Example with output

```
public class Test
{
public static void main(String args[])
{
//create boxes using the various constructors
Box mybox1 = new Box(10,20,15);
Box mybox2 = new Box();
Box mycube = new Box(7);
double vol;
//get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is" +vol);
//get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is" +vol);
//get volume of cube
vol = mycube.volume();
System.out.println("Volume of mycube is" +vol);
```

3
3

Output

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

Example of Copy Constructor

```
class demo
{
int i,j,k;
demo(demo d) // copy constructor
{
i = d.i;
j = d.j;
k = d.k;
}
}
Class hello
{
public static void main (String [] args)
{
demo d = new demo
demo d1 = new demo(d)
// here copy constructors invoked
}
}
```

Example of Default constructor

```
class Bike1
{
Bike1()
{
System.out.println("Bike is created");
}
public static void main(String args[])
{
Bike1 b = new Bike1();
```

Output:

Bike is created.

Example of Parameterised Constructor

```
Class Student4
{
int id;
String name;
Student4(int i, String n)
{
id = i;
name = n;
}
void display()
{
System.out.println(id+" "+name);
}
public static void main(string args[])
{
Student4 s1 = new Student4(111, "Karan");
Student4 s2 = new Student4(222, "Aryan");
s1.display();
s2.display();
}
}
```

Output

111 Karan

222 Anyan

Characteristics of Constructors

⇒ A constructor has the same name as the class name.

⇒ A constructor has no return type, not even void.

⇒ Access specifies of the constructors are generally public.

⇒ A constructor is automatically called on object creation.

Access Specifiers

→ It deals with scope of uses of a function it can be either public, protected, private & default type.

Or

→ The access specifier specifies the visibility mode of class members

→ Java support the following types of access specifier they are:-

⇒ public

⇒ private

⇒ protected

⇒ default

	Inclass	Outside class	Derived class
Private	✓	X	X
Public	✓	✓	✓
Protected	✓	X	✓

Private

→ Private member are accessed within the class only.

Public

Public member are accessed within the class, outside of the class and in the derived class.

Protected

→ Protected members are accessed within the class and in the derived class

Default

→ Default members are accessed within the package. Package ~~member~~ is a collection of classes and interface

Access Modifier

→ Access Modifier are used to modify the accessibility of class members

→ Java uses the following access modifier they are:-

⇒ Final
⇒ Static
⇒ Abstract
⇒ Transient
⇒ Volatile

Final

→ Final keyword is used to declare the variable whose value is fixed ~~toward~~ through out the program execution.

→ It is used to declare constants in java

Example:- final int x=10;

The value of 'x' can be change during program execution.

Static

→ Static keyword is used to creating class method and variables.

Abstract

It is used to creating abstract classes and method.

Volatile or Synchronisations

→ It is used for threads

Transient

→ It is not serialised

Access Control

Access Control specifies what parts of a program can access the member of a class. and so prevent misuse.

Chapter 4

USING JAVA OBJECTS

classmate Date Page

String Builder

⇒ Strings in java are a sequence of immutable characters. String Buffer, on the other hand, is used to create a sequence of mutable characters. String Builder is similar to String Buffer class, but it does not provide any of synchronization. However, it is much faster in nature under most implementations.

→ String Builder class is not safe for use by multiple threads.

→ Using string builder, we can solve performance problem (which is created by String buffer object) because multiple thread are allowed to operate on String Builder Object. This concept is introduced in Java 1.5 version.

Methods of String Builder

→ Append Method : This method adds the specified elements to the existing string.

→ Insert Method : This method inserts a string at the specified index.

→ Replace Method : This method replace the string from the specified begin Index to the end Index

→ Delete Method : This method deletes the specific string from the begin Index to the end Index.

→ Reverse Method : This method reverses the string ~~from~~ present in the String Builder.

→ Capacity Method : The current capacity of the Builder can be determined by this method. ~~It must~~ The capacity of the builder can be increased by : (old capacity * 2) + 2

Program →

```
public class Main
{
public static void main(String args[]) {
StringBuilder str = new StringBuilder("Rock");
str.append("Roll");
System.out.println(str);
StringBuilder str1 = new StringBuilder("Rock");
str1.insert(2,"Roll");
System.out.println(str1);
StringBuilder str2 = new StringBuilder("Rock");
str2.replace(1,3,"Roll");
System.out.println(str2);
StringBuilder str3 = new StringBuilder("Rock");
str3.delete(1,3);
System.out.println(str3);
StringBuilder str4 = new StringBuilder("Rock");
str4.reverse();
System.out.println(str4);
StringBuilder str5 = new StringBuilder();
System.out.println(str5.capacity());
str5.append("Rock");
System.out.println(str5.capacity());
str5.append("It is a good day to rock");
System.out.println(str5.capacity());
}
}
```

Output →

RockRoll // append	kcoR // reverse
Ro Rollck // insert	16 // default capacity
RRollK // replace	16 // capacity
Rk // delete	34 // capacity.

STRING BUFFER

→ String Buffer class is used to create a mutable string object i.e. its state can be changed after it is created.

→ It represents growable and writable character sequence.

→ String Buffer class is used when we have to make lot of modifications to our String.

→ It is also thread safe i.e. multiple threads cannot access it simultaneously.

→ It defines 4 constructors such as:

i) StringBuffer()
ii) StringBuffer (int size)
iii) StringBuffer (String str)
iv) StringBuffer (charSequence [] ch)

Methods of StringBuffer

→ append()
→ insert()
→ reverse()
→ replace()
→ capacity()
→ ensureCapacity(): This method is used to ensure minimum capacity of StringBuffer object.

Program →

```
Class Test2 {
public static void main(String args[]) {
StringBuffer str = new StringBuffer("Hi!");
str.append("Hello!"); S.O.P(str);
StringBuffer str1 = new StringBuffer("Sb");
str1.insert(1, "tta");
System.out.println(str1);
```

```
StringBuffer str2 = new StringBuffer("Hello World");
str2.replace(6,11,"java");
System.out.println(str2);
StringBuffer str3 = new StringBuffer("Rock");
str3.delete(1,3);
System.out.println(str3);
StringBuffer str4 = new StringBuffer("Hello");
str4.reverse();
System.out.println(str4);
StringBuffer str5 = new StringBuffer();
System.out.println(str5.capacity());
3
3
```

Output —

```
Hi, Hello!    // append
Sttar         // insert
Hello java    // replace
Rk            // delete
olleH         // reverse
16            // capacity.
```

Difference Between

String Buffer	String Builder
→ Every method present in String Buffer is synchronized.	→ No method present in String Builder is synchronized.
→ At a time only one thread is allow to operate on String Buffer object. Hence String Buffer object is thread safe.	→ At a time multiple threads are allow to operate on String Builder object and hence String Builder object is not Thread Safe.
→ It increases waiting time of threads and hence relatively	→ Threads are not required to wait to operate on

performance is low.	StringBuilder object and hence relatively performance is high.
⇒Introduced in 1.0 version	⇒Introduced in 1.5 version

String Method

1. int length(): Returns the number of characters in the String.
   Ex- "String".length(); // return 6

2. Char charAt(int i): Returns the character at i^{th} index
   Ex- "String".charAt(3); // return 'i'

3. String substring(int i): Returns the substring from the i^{th} index character to end.
   Ex- "Geeks for Geeks", substring(3); // return "ks for Geeks"

4. String substring(int i, int j): Returns the substring from i to j-1 index.
   Ex- "String", substring(2,5); // returns "rin"

5. String concat(String str): Concatenates specified string to the end of this string.
   Ex- String s1 = "Sumit";
   String s2 = "Das Mohapatra";
   String output = s1.concat(s2); // return
   // returns "Sumit Das Mohapatra"

6. int indexOf (String s): Returns the index within the string of the first occurence of the specified string.

Ex- String s = "Learn Share Learn";
int output = s.indexOf("Share"); // returns 6

7. int indexOf (String s, int i): Returns the index within the string of the first occurence of the ~~object~~ specified string, starting at the specified index.

Ex- String s = "Learn Share Learn"
int output = s.indexOf("ea", 3); // returns 13

8. int lastIndexOf (String s): Returns the index within the string of the last occurence of specified string.

Ex- String s = "Learn Share Learn";
int output = s.lastIndexOf("a"); // returns 19

9. boolean equals (Object otherObj): Compares the string to the specified object.

Ex- Boolean out = "String".equals("String");
// returns true
Boolean out = "String".equals("string");
// returns false

10. boolean equalsIgnoreCase (String anotherString): Compares string to another string ignoring case considerations

Ex- Boolean out = "String".equals("String"); // returns true
Boolean out = "String".equals("string"); // returns true

11 int compareTo(String anotherString): Compares two string lexicographically.

```
int out = s1.compareTo(s2); // where s1 and s2 are
                            // strings to be compared
```

This returns difference s1-s2. If:

out < 0 // s1 comes before s2

out = 0 // s1 and s2 are equal

out > 0 // s1 comes after s2

12. int compareToIgnoreCase(String anotherString):

Compares two string lexicographically, ignoring case considerations.

```
int out = s1.compareToIgnoreCase(s2); // where s1 & s2 are
                                      // string to be compared
```

This returns difference s1-s2, If:

out < 0 // s1 comes before s2

out = 0 // s1 and s2 are equal.

out > 0 // s1 comes after s2

13. String toLowerCase(): Convert all characters in a string to lower case.

```
String word1 = "Hello";
String word3 = word1.toLowerCase();
                // returns "hello"
```

14. String toUpperCase(): Convert all characters in a String to upper case.

```
String word1 = "Hello";
String word3 = word1.toUpperCase();
                // returns "HELLO"
```

15. String replace (char oldChar, char newChar);

Return new string by replacing all occurrences of oldChar with newChar

Ex- String s1 = "keeks fonkeeks";
String s2 = "keeks fonkeeks".replace('k', 'g');
//returns Geeks for geeksgorgeeks.

Message Passing

~~Method pa~~

=> Message Passing in terms of computers is communication between processes.

=> It is the form of communication used in object-oriented programming as well as parallel programming.

=> Message passing in Java is like sending an object i.e. message from one thread to another thread.

=> It is used when threads do not have shared memory and are unable to share monitors or any other shared variables to communicate.

=> Ex.- Produce & Consumer, what producer will produce, the consumer will be able consume only.

Methods in Message Passing (Message Passing using Methods)

```
public class MsgPass {
void displayInt(int x, int y) {
int z = x+y;
System.out.println("Int value is: "+z);
}
void displayFloat(float x, float y) {
float z = x*y;
System.out.println("Float Value is: "+z);
}
}
```

```
class MainClass {
public static void main(String args[]) {
MsgPass obj = new MsgPPass();
obj.displayInt(10,20);
obj.displayFloat((float)2.3.5,(float)5.89);
}
}
```

Output

Int Value is: 30

Float Value is: 13.841499

Message Passing using Constructors

```
public class MsgPass1 {
MsgPass1(int x, int y) {
int z=x+y;
System.out.println("Int Value is:"+z);
}
MsgPass1(float x, float y) {
float z=x*y;
System.out.println("Float Value is:"+z);
}
}
class MainClass {
public static void main(String args[]) {
MsgPass1 obj1 = new MsgPass1(5,10);
MsgPass2 obj2 = new MsgPass1((float)12.23,(float)11.23);
}
}
```

Output

Int Value is: 15

Float Value is: 137.3429

Parameters Passing

There are different ways in which parameter data can be passed into and out of methods and functions.

Let us assume that a function B() is called from another function A(). In this case, A is called the "caller function" and B is called the "called function or callee function". Also, the argument which A sends to B are called actual arguments and the parameters of B are called formal arguments.

Types of Parameters

→ Formal Parameter
→ Actual Parameter

Formal Parameter

A variable and its type as they appear in the prototype of the function or method.

Or

The parameters that appears in function defination are called formal parameters.

Syntax: function_name(datatype variable_name)

Actual Parameter

The variable or expression corresponding to a formal parmater that appears in the function or method call in the calling environment.

Or

The parameters that appear in function call statement are called actual parameters

Syntax: func_name(variable name(s));

Program Example 1:

```
public class Abc
{
int func(int, int); // prototype in the function
int func( int a =3, int b=4);// formal parameter
{
int x,y;
x=a;
y=b;
int z= x+y;
return z;
}
public static void main (String args[])
{
int result;
result = func(7,9); //actual parameter
System.out.println ("The result =" +result);
}
}
```

Output;
The result = 16

Program Example 2:

```
public class Area
{
int public int mult( int x, int y)  // Method Definition
                                    // x & y are formal parameter
{
return x * y;
}
public static void main (String args[])
{
int length = 10;
int breadth =5;
int area = mult(length, width); //length & width are
                                actual parameter
```

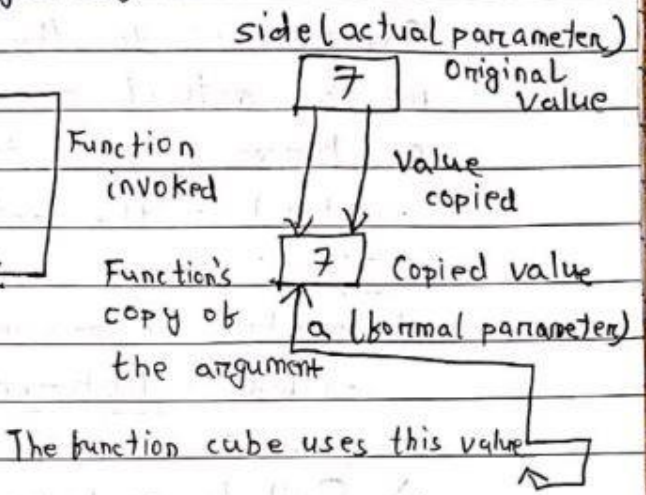
System.out.println("Area : " + area);
3
3
Output
Area : 50

Methods of Paramater Passing

1. Pass By Value : (Call by Value)
Changes made to formal parameter do not get transmitted back to the caller. Any modifications to the formal parameter variable inside the called function or method affect only the separate storage location and will not be reflected in the actual parameter in the calling environment. This method is also ~~called~~ called as call by value.
OR
⇒ Pass by value method copies the value of the actual parameters into the formal parameters i.e, the function creates its own copy of argument values and then uses them
⇒ Call by value method, the changes are not reflected back to the original values
⇒ The main benefit of call by value method is that you cannot alter the variables that is used to call the function because any change that occurs inside function is on the function's copy of the argument value. The original copy of argument values remain intact

Example 1 :-

```
class passByVal {
public static void main(String args[]) {
int vol, side=7;
vol=cube(side);
System.out.println(vol);
}
public static int cube(int a) {
return a*a*a;
}
}
```



Output : 343

Example 2 :-

```
public class MyDemoClass {
public static void main(String args[]) {
// Call By Value
int n1 =10;
modify (n1);
System.out.println(n1);
}
public static void modify (int n2) {
n2= 40;
System.out.println(n2);
}
}
```

Output

2. Call By Reference (Pass by Reference)

Changes made to formal parameter do get transmitted back to the caller through parameter passing. Any changes to the formal parameter are reflected in the actual parameter in the calling environment as formal parameter receives a reference (or pointer) to the actual data.

⇒ In place of passing a value to the function being called, a reference to the original variable is passed.

⇒ Call by reference method, the changes are reflected back to the original value.

⇒ It is useful in situatations where the values of the original variables are to be changed using a function.

Example 1:-

```
public class JavaApplication
static void displayArray(int[]a)
{
System.out.println("\nInside Display Array method");
a[0]=0; a[1]=0; a[2]=0;
for(int i=0; i<3; i++)
System.out.print(a[i]+" ");
}
public static void main(String[] args) {
int arr[]={2,2,2};
System.out.println(arr[i]+" ")("Before the function");
for(int i=0; i<3; i++)
System.out.println(arr[i]+" ");
```

```
displayArray(arr);          // Reference
System.out.println("\n After the function");
for(int i=0; i<3; i++)
System.out.print(arr[i] + " ");
}
```

Output -

```
Before the function
2 2 2
Inside Display Array method
0 0 0
After the function
0 0 0
```

Example 2 -

```
class CallByReference {
int a, b;
// Function to assign the value to the class variables
CallByReference(int x, int y)
{
a=x;
b=y;
}
// Changing the value of class variables
void ChangeValue(CallByReference obj)
{
obj.a += 20;
obj.b += 10;
// Caller
public class Main {
public static void main(String args[]) {
// Instance of class is created and value is assigned using
// constructor
```

```
CallByReference object = new CallByReference(10,20);
System.out.println("Value of a:"+object.a+"&b:"+object.b);
//Changing values in class function
object.ChangeValue(object);
// Displaying the values after calling the function
System.out.println("Values of a:"+object.a+"&b:"+object.b);
}
}
```

Output:

```
Value of a: 20 &b:10
Value of a: 40 &b:20
```

Comparision of String

Mainly there are ~~four~~ three ways for comparision such as: i) ==
ii) equals()
iii) compareTo()

i) == (Equality)

→ It include comparision like a == b or a != b
The use of == with
→ It compares references, not values
→ The use of == with object references is generally limited in the following:
⇒ Compairing to see if a reference is null.
⇒ Compairing two enum values, This work because there is only one object for each enum constant
⇒ We should have to know if two references are to the same object

iii) equals()

⇒ It compares values for equality. because this method is defined in the object class from which all other classes are derived, it's automatically defined for every class.

⇒ However, it doesn't perform an intelligent comparision for most classes unless the class overrides it.

⇒ It compares the original content of the string.

iii) CompareTo()

⇒ It is a comparable interface

⇒ It compares value and returns ~~as~~ an int which tells if the values compare less than, equal or greater than.

⇒ If we want to implement it in our class then we want to use it with Collections.sort() or Arrays.sort() methods.

Program for == operator

```
Class String2
{
public static void main(String args [])
{
String s1 = "Sona";
String s2 = "Sona";
String s3 = new String ("Sona");
System.out.println (s1==s2); //true (because both refer to same instance
System.out.println (s1==s3); //false (because s3 refers to instance
                                      created in nonpool)
3
3
```

Output :-

true

false

Program for equals() method

```
Class String1
{
public static void main(String args[])
{
String s1="Sona";
String s2="Sona";
String s3="Sumit";
System.out.println(s1.equals(s2));
System.out.println(s1.equals(s3));
3
3
```

Output

true

false

Program for compareTo() method

```
class String3
{
public static void main(String args[])
{
String s1="Sona";
String s2="Sona";
String s3="Sumit";
System.out.println("s1.compareTo(s2));
System.out.println("s1.compareTo(s3));
System.out.println("s3.compareTo(s1));
3
3
```

Output

0

-6 // (because s3 > s1)

6 // (because s3 < s1)

Example of Pass By Value

```
public class Java
{
static void displayPrimitive(int a)
{
System.out.println("Inside Display Primitive method");
a=a+5;
System.out.println("a:"+a);
}
public static void main(String args[]) {
int a=5;
System.out.println("Before the function a:"+a);
displayPrimitive(a);
System.out.println("After the function a:"+a);
}
}
```

Output: Before the function a:5
Inside Display Primitive method
a:10
After the function a:5

Unit-5

INHERITANCE

Inheritance in Java

⇒ The process of obtaining the data members and methods from one class to another class is known as inheritance.

OR

⇒ Inheritance is a mechanism in which one object acquires all the properties and behaviours of a parent object.

⇒ It is an important part of OOPs (Object Oriented Programming).

⇒ The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

⇒ It represent the IS-A relationship which is also known as parent-child relationship.

Why we use Inheritance in Java

→ For method Overriding (so runtime polymorphism can be achived)

→ For code Reusability

→ It's main uses are to enable polymorphism and to be able to reuse code for different class by putting it in a common super class.

Terms used in Inheritance

→ **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.

→ **Sub Class / Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.

→ **Super Class / Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class, or parent class.

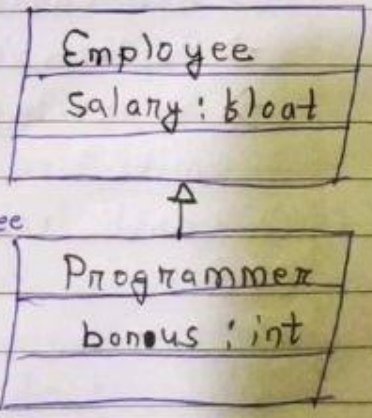
→ **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. We can use the same fields and methods ~~of~~ which is already defined in the previous class.

Syntax of Inheritance

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

Java Inheritance Example

As displayed in figure, Here, Programmer is a subclass and Employee is the superclass. The relationship between the two classes is Programmer is-A Employee. It means that



Programmer is a type of Employee.

Program →

```
Class Employee {
float salary = 40000;
}
class Programmer extends Employee {
int bonus = 10000;
public static void main (String args[])
Programmer P = new Programmer();
System.out.println("Programmer salary is:" +P.salary);
System.out.println("Bonus of Programmer is:"+P.bonus);
}
}
```

Output →

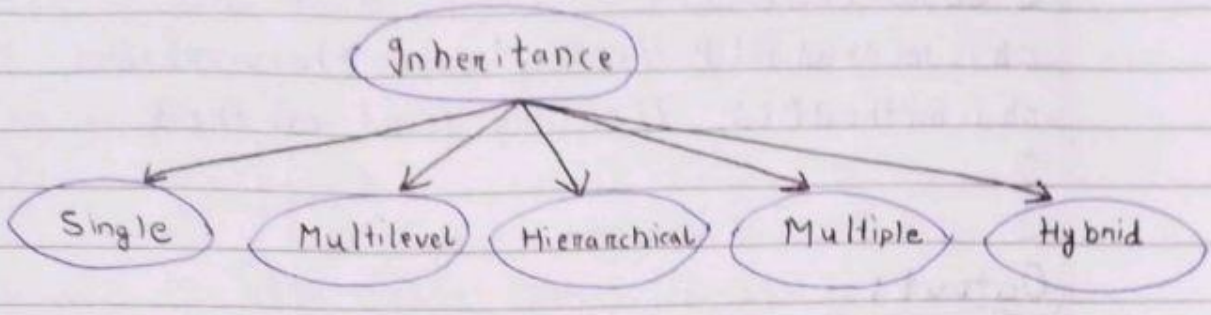
```
Programmer salary is: 40000.0
Bonus of Programmer is: 10000
```

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. reusability

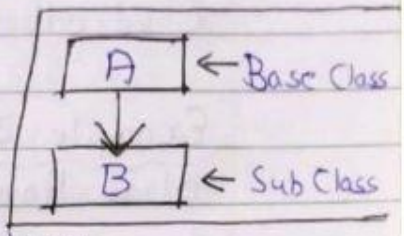
Advantage of Inheritance

→ Application development time is less
→ Application take less memory.
→ Application execution time is less.
→ Application performance is enhance.
→ Repetition of the code is reduced or minimised so that we get consistence results and less storage cost.

Types of Inheritance



Single Inheritance



→ Single inheritance can be defined as when a subclass inherits only from one base class. Or

→ When a class extends one class only then we call it a single inheritance.

→ Here, the diagram shows that class B extends only one class which is A. A is a parent class of B and B would be a child class of A.

Example 1:

```
class A
{
public void methodA()
{
System.out.println("Base of class method");
}
}
class B extends A
{
public void methodB()
{
System.out.println("Child class method");
}
public static void main (String args[])
```

```
{
B obj = new B();
obj.methodA(); // calling super class method
obj.methodB(); // calling local method
}
}
```

Output :-

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

Example 2 :-

```
class Animal
{
void eat()
{
System.out.println("eating...");
}
class Dog extends Animal
{
void bark()
{
System.out.println("barking...");
}
Class TestInheritance
public static void main(String args[])
{
Dog d = new Dog();
d.bark();
d.eat();
}}
```

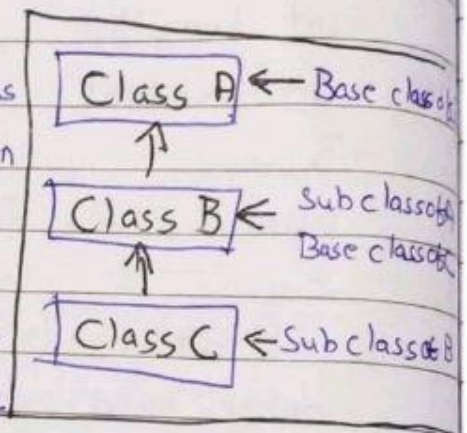
Output- barking...
eating...

Multilevel Inheritance

⇒ When a class extend class, which extends another class then this is called multilevel inheritance.

→ For example class C extends class B and class B extends class A then type of inheritance is called multilevel ~~prog~~ inheritance.

⇒ The transitive nature of inheritance is reflected by this form of inheritance.



Example 1:-

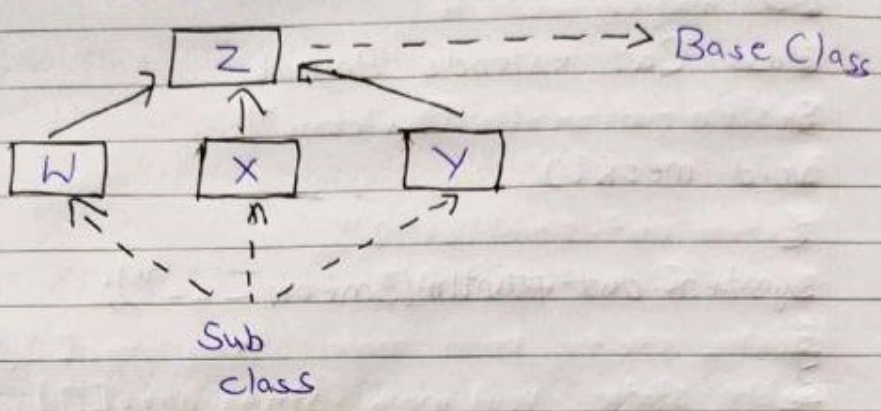
```
Class Car
{
public Car()
{
System.out.println("Class Car");
}
public void vechileType()
{
System.out.println("Vechicle Type : Car");
}
}
Class Maruti extends Car
{
public Maruti()
{
System.out.println("Class Maruti");
}
public void ~~break~~ brand()
{
System.out.println("Brand Maruti");
```

```
}
Public void speed()
{
System.out.println("Max. 90 Kmph");
}
public class Maruti800 extends Maruti
{
public Maruti800()
{
System.out.println("Maruti Model: 800");
}
public void speed()
{
System.out.println("Max: 80Kmph");
}
public static void main(String args[])
{
Maruti 800 obj = new Maruti800();
obj.VechicleType()
obj.Brand();
obj.speed();
```

Output:-

```
Class Car
Class Maruti
Maruti Model: 800
Vechicle Type: Car
Brand: Maruti
Max: 80Kmph
```

Hierachical Inheritance



⇒ When many sub classes inherit from a single base class, it is known as hierachical inheritance. Or

⇒ When more than one classes inherit a same class, then this is called hierachical inheritance

⇒ For example class W, X and Y extends a same class Z. According to diagram, we came to know that, when a class have more than one child classes (sub classes) or in other words more than one child classes have the same parent class then the type of inheritance is called hierachical inheritance.

Example:

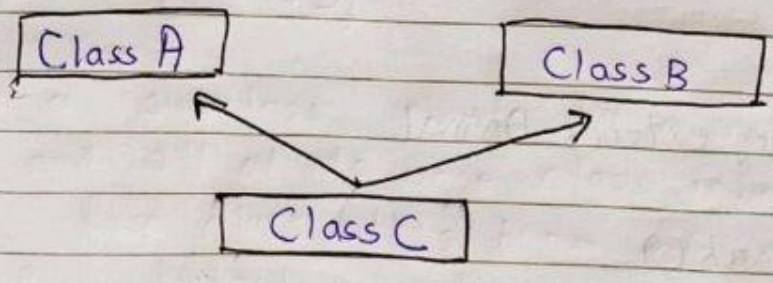
```
class Animal {
void eat()
 {
System.out.println("eating---");
}
}
Class Dog extends Animal
{
void bark()
{
```

```
System.out.println("barking---");
}
}
class Cat extends Animal
{
void meow()
{
System.out.println("meow---");
}
}
class Inheritance3
{
public static void main(String args[])
{
Cat c= new Cat();
c.meow(); Dog d= new Dog();
c.eat();
d.bark();
}
}
```

Output:

meow---
eating---
barking---

Multiple Inheritance



⇒ Multiple Inheritance refers to the concept of one class extending more than one base class.

⇒ The problem with "mutiple inheritance" is that the derived class will have to manage the dependency on two baseclasses.

Why it is not supported in Java?

⇒ To reduce the complexity and simplify the language, multiple inheritance is not supported in java

⇒ Consider a seenario, where, A, B, C are three classes. The C inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

⇒ Since compile time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So, whether you have same method or different, there will be compile time error.

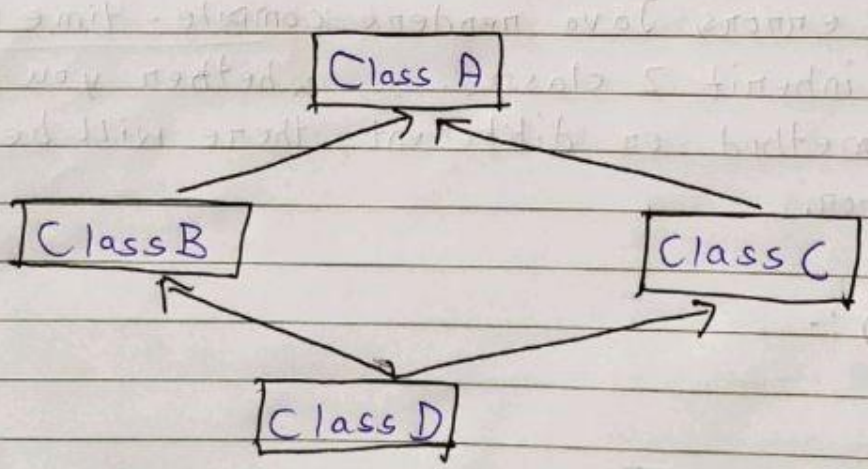
Program :-

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

```
class B
{
void msg()
{
System.out.println("Welcome");
}
}
class C extends A,B
{
public static void main (String args[])
{
C obj = new C();
obj.msg();
}
}
```

Output:
Compile Time Error

Hybrid Inheritance



⇒ When a subclass inherits from multiple base classes and all of its base class inherit from a single base class, this form of inheritance is known as hybrid inheritance.

⇒ For example when class A and B extends class C & another class D extends class A then this is a hybrid inheritance, because it is a combination of single and hierachical inheritance.

Program:-

```
class C
{
public void disp()
{
System.out.println("C");
}
}
class A extends C
{
public void disp()
{
System.out.println("A");
}
}
Class B extends C
{
public void disp()
{
System.out.println("A");
}
}
class B extends C
{
public void disp()
{
System.out.println("B");
```

⇒ For example when class A and B extends class C & another class D extends class A then this is a hybrid inheritance, because it is a combination of single and hierachical inheritance.

Program:-

```
class C
{
public void disp()
{
System.out.println("C");
}
}
class A extends C
{
public void disp()
{
System.out.println("A");
}
}
Class B extends C
{
public void disp()
{
System.out.println("A");
}
}
class B extends C
{
public void disp()
{
System.out.println("B");
```

```
}
}
class D extends A
{
public void disp()
{
System.out.println("D");
}
public static void main(String args[])
{
D obj = new D();
obj.disp();
}
}
```

Output :-

D

Chapter 6

POLYMORPHISM

Polymorphism in Java

⇒ Polymorphism in Java is a concept by which we can perform a single action in different ways. Polymorphism is derived from two greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

⇒ There are two types of polymorphism in Java:
  → Compile time polymorphism
  → Runtime polymorphism

⇒ We can perform polymorphism in java by method overloading and method overriding.

⇒ Polymorphism is one of the OOPs feature that allow us to perform a single action in different ways.

⇒ Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Real life example of polymorphism: A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So, the same person posses different behaviour in different situations. This is called polymorphism.

For Example: Lets say we have a class Animal that has a method sound(). Since this is a generic class so we can't give it a implementation like Roar, Meow, Oink etc. We had to give a generic message.

```
public class Animal
{
---
public void sound() {
System.out.println("Animal is making a sound");
}
}
```

⇒ Now lets say we two subclasses of Animal class, Horse and Cat that extends (see Inheritance) Animal class. We can provide the implementation to the same method like this:

```
public class Horse extends Animal {
---
@Override
public void sound() {
System.out.println("Neigh");
}
}
```

and

```
public class Cat extends Animal {
---
@Override
public void sound() {
System.out.println("Meow");
}
}
```

As you see above, we had the common action for all subclasses sound() but there were different ways to do the same action. This is a perfect example of polymorphism. It would not make any sense to just call the generic sound() method as each Animal has a different sound. Thus we say that

P.T.O. >>

the action this method performs is based on the type of object.

TYPES OF POLYMORPHISM

→ Static Polymorphism or Compile time Polymorphism

→ Dynamic Polymorphism or Runtime Polymorphism

Compile-time Polymorphism

⇒ Polymorphism that is resolved during compiler time is known as static polymorphism. Method Overloading is an example of complile time polymorphism.

⇒ Method Overloading: This allows us to have more than one method having the same name, if the parameters of methods are different in number, sequence and data types of parameters. ~~We have~~

⇒ Example of Static Polymorphism

Method overloading is one of the way java supports static polymorphism. Here we have two definitions of the same method add() which add method would be called is determined by the parameter list at the compile time. That is the reason this is also known as compile time polymorphism.

Program →

```
class SimpleCalculator
{
int add(int a, int b)
{
return a+b;
}
int add(int a, int b, int c)
{
return a+b+c;
}
}
public class Demo
{
public static void main(String args[])
{
SimpleCalculator obj = new SimpleCalculator();
System.out.println(obj.add(10,20));
System.out.println(obj.add(10,20,30));
}
}
```

Output

30

60

Runtime Polymorphism

⇒ It is also known as Dynamic Method Dispatch.

⇒ Dynamic polymorphism is a process in which a call to an overridden method is resolved at runtime, thats why it is called runtime polymorphism.

⇒ Example

In this example we have two classes ABC and XYZ. ABC is a parent class and XYZ is a child class. The child class is overriding the method myMethod() of parent class. In this example we have child class object assigned to the parent class reference so in order to determine which method would be called, the type of the object would be determined at run-time. It is the type of object that determines which version of the method would be called.

```
class ABC
{
public void myMethod()
{
System.out.println("Overridden Method");
}
}
public class XYZ extends ABC
{
public void myMethod()
{
System.out.println("Overriding Method");
}
public static void main(String args[])
{
ABC obj = new XYZ();
obj.myMethod();
}
}
```

Output :-
Overriding Method

Before types of polymorphism

⇒ When a overridden method is called through reference of parent class, then type of the object determines which method is to be executed, Thus, this determination is made at run time.

Method Overloading

⇒ Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different. It is similar to constructor overloading.

⇒ To call an overloaded method in java, it is must use the type and/or the number of arguments to determine which version of the overloaded method actually to call.

⇒ The overloaded methods may have varied return types and the return type single-handedly is insufficient to make out two versions of a method.

⇒ As and when Java compiler encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

⇒ It permits the user to obtain compile time polymorphism with name method name.

⇒ An overloaded method can throw different kinds of exceptions.

⇒ A method which is overloaded can contain different access modifiers

For example, the argument list of a method add (int a, int b) having two parameter in different ~~from~~ from the argument list of the method add (int a, int b, int c) having three parameters

⇒ There are three ways to overload a method

→ Numbers of parameters.

Ex:- add (int, int)
add (int, int, int)

→ Data type of parameters

Ex:- add (int, int)
add (int, float)

→ Sequence of Datatype of parameters

Ex :- add (int, float)
add (float, int)

⇒ When I say argument list, I am not talking about return type of the method, for example if two methods have same name, same parameters and have different return type, then this is not a valid method overloading example.

Example:

```
class Display
{
public void disp (char c)
{
System.out.println(c);
}
public void disp (char c, int num)
{
System.out.println (c + " " + num);
```

```
}
}
class Sample
{
public static void main (String args[])
{
Display obj = new Display();
obj.disp('a');
obj.disp('a', 10);
}
}
```

Output

a

a 10

Method Overridding

⇒ In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass.

⇒ When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

⇒ It is an example of Dynamic Polymorphism.

```
class Human
{
// Overridden method
public void eat()
{
System.out.println("Human is eating");
}
}
class Boy extends Human
{
// Overriding method
public void eat()
{
System.out.println("Boy is eating");
}
public static void main(String args[])
{
Boy obj = new Boy();
// This will call the child class version of eat()
obj.eat();
}
}
```

Output:

Boy is eating

Advantage of method overriding ~~in Java~~

→ The main advantage of method overriding is that the class can give its own specific implementation to a inherited method without even modifying the parent class code.

Rules for Overriding

⇒ The argument list should be exactly the same as that of that overridden method.
⇒ The return type should be the same or a subtype of the return type declared in the original overridden method in superclass.
⇒ The access level cannot be more restrictive than the overridden method's access level.
⇒ Instance methods can be overridden only if they are inherited by the subclass.
⇒ A method declared final cannot be overridden.
⇒ Constructor cannot be overridden

Super keyword in Method Overridding

⇒ The super keyword is used for calling the parent class method / constructor.
⇒ super.myMethod() calls the myMethod() method of base class while super() calls the constructor of base class.

Method Overloading	Method Overridding
⇒ Method Overloading is used to increase the readability of the program	⇒ Method Overridding is used to provide the specific implementation of the method that is already provided by its superclass
⇒ It is performed within class	⇒ It occurs in two classes that have inheritance relationship.
⇒ In case of method overloading parameter must be different	⇒ In case of method overridding parameter must be same

⇒ It is a example of complile time polymorphism ⇒ It can't be performed by changing return type of method only. Return type can be same or different in method over-loading	⇒ It is the example of run time polymorphism, ⇒ Return type must be same or covariant in method overridding.

Static polymorphism	Dynamic Polymorphism
→ Static Polymorphism decides which method to execute during compile time. → Method Overloading is an example of static polymorphism → Static Polymorphism is achieved through static binding. → Static Polymorphism happens in same class. → Inheritance not involved for static polymorphism	→ Dynamic Polymorphism decides which method to execute in run time → Method Overridding is an example of dynamic polymorphism → Dynamic Polymorphism is achived through dynamic binding. → Dynamic Polymorphism happen between different classes. → Inheritance involved for dynamic polymorphism

Prepared By: Biswa Bandita Mohanty

Example of Static Binding

```
Class A
{
void method () {
System.out.println ("From Class A");
}
}
Class B extends A
{
@ Overnide
void method()
```

```
{
System.out.println("From Class B");
}
}
public class Binding
{
public static void main(String args[])
{
A a1 = new A();
a1.method();
A a2 = new B();
a2.method();
}
}
```

Static Binding

Example of Dynamic binding

```
public class Binding
{
public static void main(String args[])
{
A a1 = new A();
a1.method();
A a2 = new B();
a2.method();
}
}
```

Class A type object

```
void method()
{
System.out.println("From Class A");
}
```

```
void method()
{
System.out.println("From class B");
}
```

Class B type object

Chapter 7

Packages: Putting Classes Together

Packages in Java

⇒ Java Package is a mechanism of grouping similar type of classes, interfaces, and sub-classes collectively based on functionality.

⇒ When software is written in Java programming language, it can be composed of hundreds or even thousands of individual classes. It makes sense to keep things organized by placing related classes and interfaces into packages

⇒ A package is container of a group of related classes where some of the classes are accessible are exposed and others are kept for internal purpose.

⇒ Using packages while coding offers a lot of advantages like:

- → Re-usability: The classes contained in the packages of another program can be easily reused.
- → Name Conflicts: Packages help us to uniquely identify a class, for example, we can have company sales.Employee and company marketing Employee classes.
- → Controlled Access: Offers access protection such as protected classes, default classes & private classes.
- → Data Encapsulation: They provide a way to hide classes, preventing other programs from accesing classes that are ~~made~~ meant for internal use only.
- → Maintainance:- With packages, you can organize your project better and easily locate related classes.

⇒ It's a good pratice to use packages while coding in java. As a programmer, you can easily figure out the classes, interfaces, enumerations and annotations that are related.

Advantages of Java Package

⇒ Java package is used to categorize the classes and interfaces so that they can be easily maintained.

⇒ Java package provides access protection,

⇒ Java package removes naming collision,

Types of Packages in Java

Based on whether the package is defined by the user or not, packages are divided into two categories:

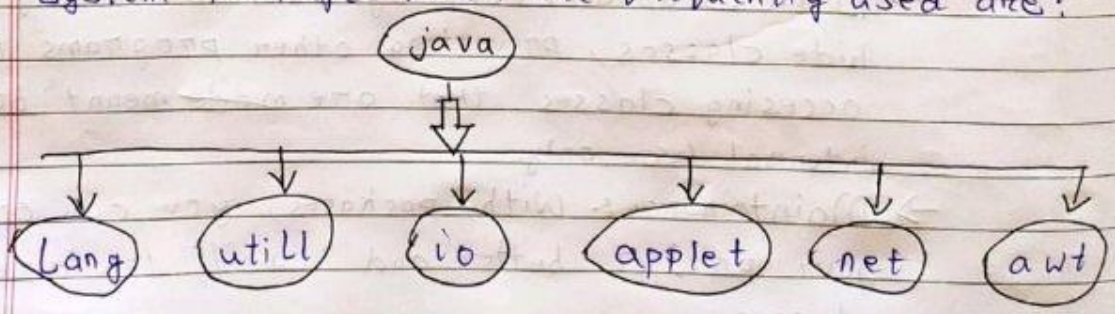
→ Built-in Packages

→ User Defined Packages

Java API Packages

Java API (Application Program Interface) provides a large numbers of classes grouped into different packages according to functionality. Most of the time we use the packages available with the Java API.

System Package which are frequently used are:



java.lang	Language support classes. They include classes for primitive types, string, math functions, thread and exceptions.
java.util	Language utility classes such as vectors, hash tables, random numbers, data, etc.
java.io	Input/output support classes. They provide facilities for the input and output of data.
java.applet	Classes for networking. They include classes for communicating with local computers as well as with internet servers
java.awt	~~Java~~ Set of classes for implementing graphical user interface. They include classes for windows, buttons, list, menus and so on.

Built-in Packages

Built-in packages or predefined packages are those that come along as a part of JDK (Java Development Kit) to simplify the task of Java programmer. They consist of a huge number of predefined classes and interfaces that are a part of Java API's. Some of the commonly use built-in packages are java.lang, java.io, etc. Below there is a simple program using a built-in package

↳ P.T.O.

```
package Sumit;
import java.util.ArrayList;
class BuiltInPackage
{
public static void main (String args[])
{
ArrayList<Integer> myList = new ArrayList<>(3);
myList.add(3);
myList.add(2);
myList.add(1);
System.out.println("The elements of list are:"+myList);
}
}
```

Output

The elements of list are: [3,2,1]

The ArrayList class belongs to java.util package. To use it, we have to import the Package using the import statement. The first line of the code import java.util.ArrayList imports the java.util package and uses ArrayList class which is present in the sub package util.

⇒ The Java API is a library of prewritten classes that are free to use, included in the Java Development Environment.

⇒ The library contains components for managing input, database programming, and much much more.

⇒ The library is divided into packages and classes. Meaning you can either import a single class, or a whole package that contain all the classes that belong to the specified package.

⇒ To use a class on a package from the library, you need to use the import keyword:

Syntax

```
import package.name.Class;   // Import a single class
import package.name.*;       // Import the whole class
```

Import a Class

If you find a class you want to use, for example, the Scanner class, which is used to get user input, write the following code:

Example

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

In the example above, java.util is a package, while Scanner is a class of the java.util package.

To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class documentation. In our example, we will use the nextLine() method, which is used to read a complete line:

Example:

Using the Scanner class to get user input:

```
import java.util.Scanner;
~~Import a Package~~
class MyClass
{
public static void main(String args[]) {
Scanner myObj = new Scanner(System.in);
System.out.println("Enter username");
String userName = myObj.nextLine();
System.out.println("Username is:" +userName);
}
}
```

Import a Package

There are many packages to choose from. In the previous example, we used the Scanner class from the java.util package. This package also contains date and time facilities, random-number generator and other utility classes.

To import a whole package, end the sentence with an asterisk sign (*). The following example will import all the classes in the java.util package;

Example

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

User-Defined Packages

⇒ User-defined packages are those which are developed by users in order to group related classes, interfaces and subpackages.

⇒ To create your own package, you need to understand that Java use a file system directory to store them. Just like folders on your computer;

Example

```
└root
    └mypack
        └MyPackageClass.java
```

⇒ To create a package, we use the package keyword

Program :—

```
package mypack;
class MyPackageClass
{
public static void main (String args[])
```

```
{
System.out.println("This is my package!");
}
}
```

⇒ Save the file as MyPackageClass.java, and compile it:

C:\Users\Documents> javac MyPackageClass.java

Then compile the package

C:\Users\Documents> javac -d . MyPackageClass.java

This forces the compiler to create the "mypack" package

The -d keyword specifies the destination for where to save the class file. You can use any directory name, like c:/user (windows), or, if you want to keep the package within the same directory, you can use the dot sign ".", like in the example above.

Note: The package name should be written in lower case to avoid conflict with class names

When we compiled the package in the example above, a new folder was created, called "mypack". To run the MyPackageClass.java file, write the following:

C:\Users\Documents\ java mypack.MyPackageClass

Output will be:

This is my package!

Naming Convention

⇒ It should be a lowercase letter such as java, lang.

⇒ If the name contains multiple words, it should be separated by dots (.) such as java.util, java.lang.

→ Example :-

```
package com.javapoint;
class Employee
{
// code - - - -
}
```

Creating a Package in Java

Creating a package in Java is a very easy task. Choose a name for the package and include a package command as the first statement in the Java source file. The java source file can contain the classes, interfaces, enumerations and annotation types that you want to include in the package. For example, the following statement creates a package named MyPackage.

```
package MyPackage;
```

The package statement simply specifies to ~~when~~ which package the classes defined belongs to..

Note: If you omit the package statement, the class names are put into the default package, which has no name. Though the default package is fine for short programs, it is inadequate for real applications.

Including a Class in Java Package

To create a class inside a package, you should declare the package name as the first statement of your program. Then include the class as part of the package. But, remember that, a class can have only one package declaration.

Program:

```
package MyPackage;
public class Compare
{
int num1, num2;
Compare(int n, int m)
{
num1 = n;
num2 = m;
}
public void getmax()
{
if(num1 > num2)
{
System.out.println("Maximum value of two numbers is "+num1);
}
else
{
System.out.println("Maximum value of two numbers is" num2);
}
}
public static void main (String args[])
{
Compare current[] = new Compare[3];
current[1] = new Compare(5, 10);
current[2] = new Compare(123,120);
for(int i=1; i<3; i++) {
current[i].getmax(); }
}
}
```

O/P => Maximum value of two numbers is 10
Maximum value of two numbers is 123

As you see, I have declared a package named MyPackage and created a class Compare inside the package. Java uses file system directories to store packages. So, this program would be saved in a file as Compare.java and will be stored in the directory name MyPackage. When the file gets compiled, Java will create a .class file and store it in the same directory.

Creating a class inside package while importing another package

Here is a sample program demonstrating the above concept.

```
package Sdm;
import MyPackage.Compare;
public class Demo
{
public static void main(String args[])
{
int n=10; m=10;
Compare current = new Compare(n,m);
if(n != m)
{
current.getmax();
}
else
{
System.out.println("Both the values are same");
}
}
}
```

O/p-> Both the values are same

Here, I have declared the package Sdm, then imported the class Compare from the package MyPackage. So, the order when we are creating a class inside a package while importing another package is,

->Package Declaration

-> Package Import

Well if you don't want to use import statement, there is another alternative to access a class file of the package from another package. You can just use fully qualified name while importing a class.

// Using fully qualified name while importing a class

Here is a sample example demonstrating the above concept.

```
package Sdm;
public class Demo {
public static void main (String args[]) {
int n=10, m=11;
// Using fully qualified name instead of import
MyPackage.Compare current = new MyPackage.Compare(n,m);
if (n != m)
{
current.getmax();
}
else
{
System.out.println("Both the value are same");
}
}
}
```

O/p → Maximum value of two numbers is 11.

In the Demo class, instead of importing the package, I have used the fully qualified name such as MyPackage.Compare to create the object of it. Since we are talking about importing packages.

Access Protection in Java Packages

While using packages and inheritance in a program, we should be aware of the visibility restrictions imposed by various access protection modifiers. Packages act as containers for classes and other packages, and classes act as containers for data and methods. Data members and methods can be declared with the access protection modifiers such as private, protected and public as well as default.

Access Protection

Access location \ Access modifier →	public	protected	default	private protected	private
Same class	Yes	Yes	Yes	Yes	Yes
Subclass in same package	Yes	Yes	Yes	Yes	No
Other classes in same package	Yes	Yes	Yes	No	No
Subclass in other package	Yes	Yes	No	Yes	No
Non-subclass in other packages	Yes	No	No	No	No

Hiding Classes

⇒ When we import a package using asterisk (*), all public classes are imported. However, we can prefer to "not import" certain classes. That is, we may like to hide these classes from accessing from outside of the package. Such classes should be declared "not public". Example:

```
package p1;
public class X                    //public class, available outside
{
   // body of X
}
class Y                           // not public, hidden
{
   // body of Y
}
```

⇒ Here, the class Y which is not declared public is hidden from outside of the package p1. This class can be seen and used only by other classes in the same package. Note- Java source file should contain only one public class and may include any number of non-public classes. We can also add a single non-public class using the procedure suggested in the ~~above~~ previous example.

Now, consider the following code, which imports the package p1 that contains classes X and Y:

```
import p1.*;
X objectX;        //OK; class X is available here
Y objectY;        //Not OK: Y is not available
```

Java compiler would generate an error message for this code because the class Y, which has not been declared public, is not imported and therefore not available for creating its objects

Using Static Import

Static import is a feature introduced in Java ~~pr~~ that allows members defined in a class as public static to be used in Java code without specifying the class ~~as~~ in which the field is defined.

Following program demonstrates static import:

```
//Note static keyword after import.
import static java.lang.System.*;
class StaticImportDemo
{
public ~~class~~static void main(String args[])
{
// We don't need to use 'System.out' as imported
// using static.
out.println("Sumit");
}
}
```

Output:

Sumit

Chapter 9

Exception Handling

- Exception Handling - An Overview

⇒ Exception handling in Java is one of the powerful mechanism to handle the runtime errors so the normal flow of the application can be maintained

→ An exception is a condition that is caused by a runtime error in the program. When the Java interpreter encounters an error such as dividing an integer by zero, it creates an exception object and throws it.

If the exception object is not cau

→ If we want the program to continue with the execution of the remaining code of a program, then we should try to catch the exception object thrown by the error condition and then display an appropriate message for taking corrective actions. This task is known as exception handling.

→ The purpose of exception handling mechanism is to provide a means to detect and report an "exceptional circumstances" so that appropriate action can be taken. The mechanism suggests incorporation of a separate error handling code that performs the following task :-

⇒ Find the problem (Hit the exception)

⇒ Inform that an error has occured (Throw the exception)

⇒ Receive the error information (Catch the exception)

⇒ Take corrective actions (Handle the exception)

=> Exceptions in Java can be categorized into two types:

→ Checked exceptions:- These exceptions are explicitly handled the code itself with the help of try-catch blocks. => Checked exceptions are extended from the java.lang.Exception class.

=> It is an exception is that is checked by the compiler at compilation-time, these are also called as compile time exceptions.

=> These exceptions cannot be simply be ignored, the programmer should take care of these exceptions.

For Example: If we use FileReader class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a FileNotFoundException occurs, and the compiler prompts the programmer to handle the exception.

Program →

```
import java.io.File;
import java.io.FileReader;
public class Demo
{
public static void main(String args[])
{
File file = new File("E://file.txt");
FileReader fr = new FileReader(file);
}
}
```

O/P → C:\> javac Demo.java

Demo.java:8: error: unreported exception FileNotFoundException; must be caught or declared to

be thrown

```
FileReader fr = new FileReader (file);
                ^
1 error
```

Note:- Since the methods read() and close() of FileReader class throws IOException, you can observe that the compiler notifies to handle IOException, along with FileNotFoundException.

→ Unchecked exceptions :-

⇒ These exceptions are not essentially handled in the program code; instead the JVM handles such exceptions.

⇒ Unchecked exceptions are extended from the java.lang.RuntimeException class.

⇒ It is an exceptions that occurs at the time of execution. These are also called as Runtime Exceptions.

⇒ These include programming bugs such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

For example:- If we have declared an array of size 5 in our program, and trying to call the 6th element of the array then an ArrayIndexOutOfBoundsExceptionexception occurs.

Program

```
public class Demo1
{
public static void main (String args[])
{
int ~~same~~ num[] = (1,2,3,4);
System.out.println (num[5]);
}
}
```

O/p ⇒ Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException : 5
at Exceptions.VsDemo1.main(Demo1.java:8)

→ Errors:

⇒ These are not exceptions at all, but problems that arise beyond the control of the user of the programmer.

⇒ Errors are typically ignored in your code because you can rarely do anything about an error.

⇒ For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation

Exception Handling Example

```
public class Example
{
public static void main(String args[])
{
try {
//code that may raise exception
int data = 100/0;
} catch(ArithmeticException e){System.out.println(e);}
//rest code of the program
System.out.println("rest of the code---");
}
}
```

O/p ⇒ Exception in thread main java.lang.ArithmeticException : / by zero
rest of the code---

Java Exception Keywords

There are 5 (five) keywords which are used in handling exceptions in java.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
t	The "throws" keyword is used to declare exceptions. It doesn't throw an exception.
throws	It specifies that there may occur an exception in the method. It is always used with method signature.

Catching Exceptions

A method catches an exception using a combination of try and catch keywords. A try/catch block is placed around the code that might generate exception. Code within a try/catch block is

referred to as protected code, and syntax for using try/catch is below:

Syntax

```
try {
   // Protected Code
} catch (Exception Name e1) {
      // catch block
}
```

Program →

```
import java.io.*;
public class ExcepTest
{
public static void main(String args[])
{
try {
int a[] = new int[2];
System.out.println("Acces Element three: " + a[3]);
} catch (ArrayIndexOutOfBoundsException e) {
System.out.println("Exception thrown  " + e);
}
System.out.println("Out of block");
```

O/P →

```
Exception thrown : java.lang.ArrayIndexOutOfBoundsException:3
Out of the block
```

Multiple Catch Blocks

A try block can be followed by multiple catch blocks.

Syntax:

```
try {
  // Protected Code
} catch (ExceptionType1 e1) {
    // Catch block
} catch (ExceptionType2 e2) {
    // Catch block
} catch (ExceptionType3 e3) {
    // Catch block
}
```

The previous statements demonstrates three catch blocks, but we can have any number of them after a single try. ~~If the~~

Program →

```
class Error
{
public static void main(String args[])
{
int a[]={5,10};
int b =5;
try
{
int x = a[2] /b - a[1];
}
catch (Arithmetic Exception e)
{
System.out.println ("Division by zero");
}
```

```
catch (ArrayIndexOutOfBoundsException e)
{
System.out.println("Array index error");
}
catch(ArrayStoreException e)
{
System.out.println("Wrong data type");
}
int y = a[1] /a[0];
System.out.println("y = "+y);
}
}
```

O/p →

Array index error

y = 2

Using Finally Statement

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurance of an Exception.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and the syntax is given below:-

Syntax

```
try {
  // Protected Code
} catch (ExceptionType1 e1) {
    // Catch block
} catch (ExceptionType2 e2) {
```

// Catch block
} finally {
// The finally block always executes.
}

Program →

```
public class Test {
public static void main (String args [])
{
int a [ ] = new int [2];
try {
System.out.println ("Access element three: " + a[3]);
} catch (ArrayIndexOutOfBoundsException e) {
System.out.println ("Exception thrown :" + e);
} finally {
a[0] = 6;
System.out.println ("First element values: " + a[0]);
System.out.println ("The finally statement is executed");
}
}
}
```

O/p →

Exception thrown : java.lang.ArrayIndexOutOfBoundsException:3
First element value : 6
The finally statement is executed.

Exception Methods

1. public String getMessage() - Returns a detailed message about the exception that has occurred. This message is intialized in the Throwable constructor.

2. public Throwable getCause() - Returns the cause of exception as represented by a Throwable object

3. public String toString() - Returns the name of the class concatenated with the result of getMessage().

4. public void printStackTrace() - Prints the result of toString() along with the stack trace to System.err, the error output stream

5. public StackTraceElement [] getStackTrace() - Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.

6. public Throwable fillInStackTrace() - Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

Declaring an Exception in Java

Apart the built in exceptions which Java provides, exception classes can be written by the programmer. These can then be thrown Like the built in Java exceptions.

Syntax

```
modifier exceptionName extends Exception {
 // field and methods
}
modifier exceptionName extends RuntimeException {
 // for runtime exceptions, fields and methods
}
```

⇒ If the exception thrown is a runtime exception, the class declared must extend the RuntimeException class
⇒ If the exception is thrown is handled or declared, the class must extend the Exception class.

Example Program →

```
//exception for negative numbers
public negativeNumberException extends Exception
{ //class header
private int errorNumber;
public negativeNumberException (int number) {
//exception class constructor
this.errorNumber = number;
}
public int getNumber() { // method within class
return this.errorNumber;
}
}
//a class that only takes positive numbers
public class positiveNumbers {
private int number;
// method within class throws negative number exception
public positiveNumber (int number) throws negativeNumbersException {
if(number >=0) this.number = number;
else {
throw new negativeNumberException(number);
}
}
}
// trying the exception, elsewhere in the code
try {
positiveNumbers negativeOne = new positiveNumbers(-1);
```

```
} catch (negativeNumberException e) {
System.out.println("ERROR: This number is not positive: "
                    + e.getNumber());
}
}
```

O/P->

Throwing an Exception in Java

⇒ Methods prone to an error in runtime can throw an exception, instead of having program crash or having a logic error within the program.

⇒ Exceptions are typically thrown ~~are~~ when a method itself doesn't handle a checked exception.

Syntax

```
public methodName function throws exceptionName {
// checking for regular or error condition
if (booleanExpression) { //good condition
}
else { //error condition
throw new exceptionName();
}
}
```

Example:

```
class Example
{
static void divide_m () throws ArithmeticException
{
int x = 22, y = 0, z;
z = x/y;
}
public static void main (String args[])
{
try
{
divide_m();
}
catch (ArithmeticException e)
{
System.out.println("Caught the exception =" e);
}
}
}
```

O/p->

Caught the exception java.lang.ArithmeticException: / by zero

Chapter - 8

Java Files and I/O

Concept of Stream

⇒ A stream can be defined as a sequence of data

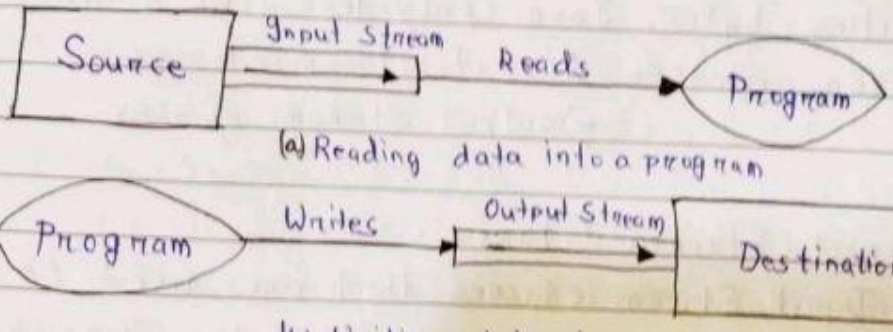
⇒ All these streams represent an input source and an output source destination. The stream class available in java.io packages.

⇒ There are two kinds of Streams

→ Input Stream :- The inputstream is used to read data from a source, or extracting data from the source and sending it to program is called input stream.

→ Output Stream :- The output stream is used for writing data to a destination. or An output stream takes data from the program and sends to the destination.

⇒ The below figure illustrates the use of input and output streams,

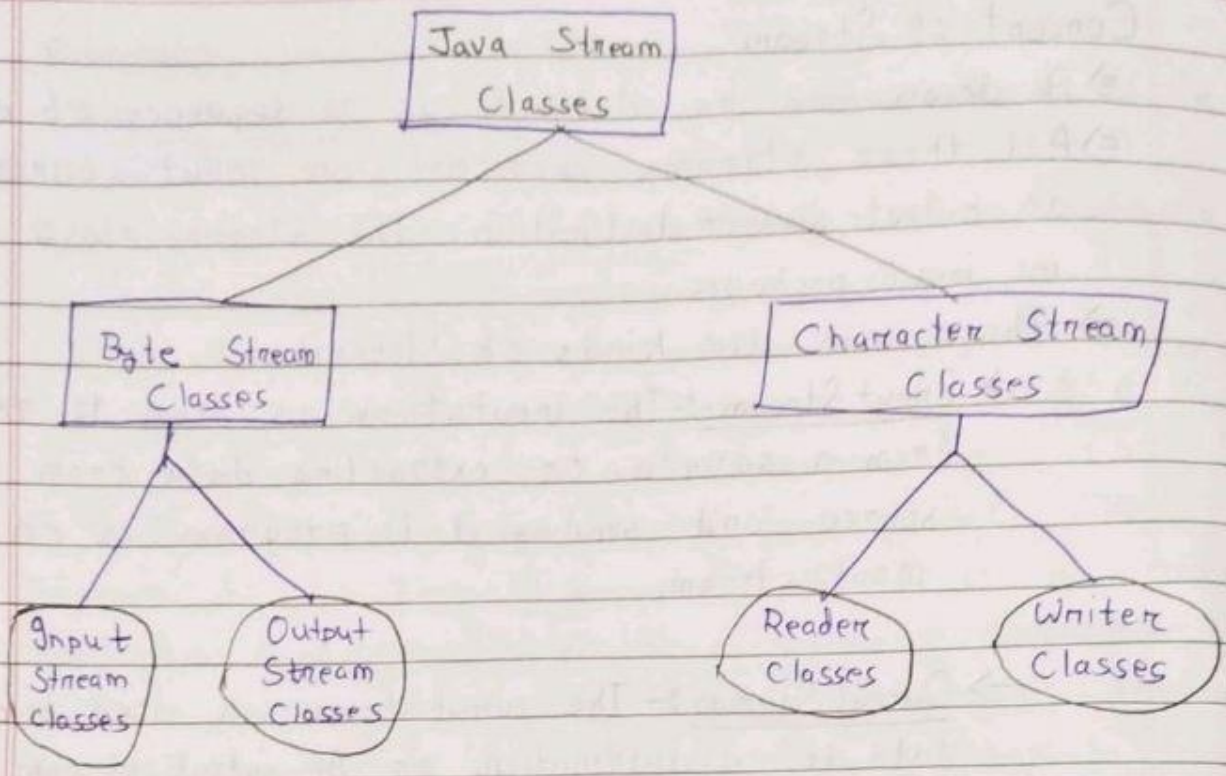


(a) Reading data into a program

(b) Writing data to a destination

Stream Classes

The java.io packages contains Stream classes which provide capabilities for processing all type of data. These classes may be categorized into two groups based on their data type handling capabilities :- 1. Byte Stream
2. Character Stream



BYTE STREAM CLASSES

Byte stream classes have been designed to provide functional features for creating and manipulating streams and files for reading and writing bytes. Java provides two kinds of byte-stream classes →input stream classes
→output stream classes

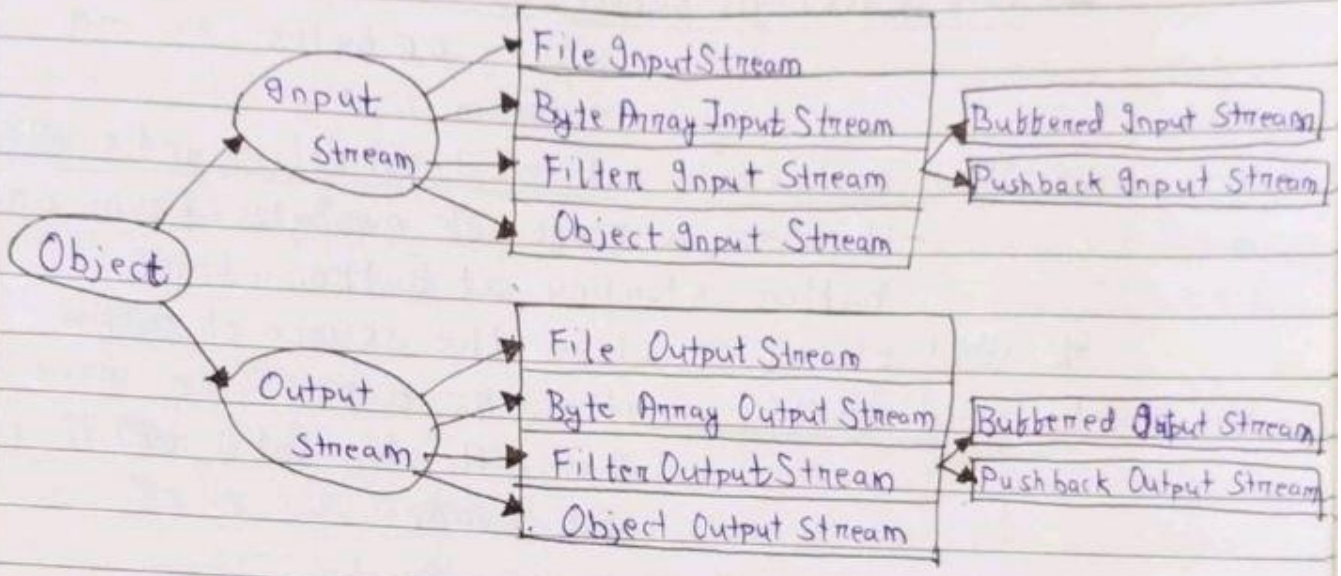
Input Stream Classes

Input Stream classes that are used to read 8-bit bytes include a super class known as Input Stream and a number of subclasses for supporting various input-related functions.

Output Stream Classes

Output stream classes are derived from the base class OutputStream. Like InputStream, the Output stream is an abstract class and therefore

we cannot instantiate it. The several subclasses of the OutputStream can be used for performing the output operations.



Hierachial ~~Rep~~ of Input & Output Stream Classes

Methods of Input Streams

1. int read() - it returns integer of next available input of byte
2. int read(byte buffer[ ]) - it reads up to the length of byte in buffer
3. int read(byte buffer[ ], int offset, int numBytes) - - it read up to the length of byte in buffer starting from offset
4. int available() - it gets the no. of bytes of input available.
5. void reset() - it reads the input pointer.
6. long skip(long numBytes) - it returns the byte ignored.
7. void close() - close the source of input.

Methods of Output Stream class

1. void write(int b) - it writes a single byte as output.
2. void write(byte buffer[]) - it writes a full array of bytes to an output stream
3. void write(byte buffer[], int offset, int numBytes) - it writes a range of numBytes from array buffer starting at buffer offset
4. void close() - close the source of output
5. void flush() - it marks a point to input stream that will stay valid untill numBytes are read.

Character Stream Classes

Java Character streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, Reader and Writer. Though internally FileReader uses FileInputStream and FileWriter uses the FileOutputStream but the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

Reader

-> It is an abstract class used to input character
-> It implements 2 interfaces closable & Readable
-> Methods in the class throws IO Exception,

Writer

- Writer is an abstract class used to output character
- It implements 3 interfaces Closable, Flushable & Appendable.

Methods of Reader Abstract Class

1. abstract void close() - close the source of input.
2. void mark(int numChars) - it makes a point to input stream that will stay valid until numChars are read.
3. boolean markSupported() - it returns true if mark() support by stream.
4. int read() - it returns integer of next available character from input.
5. int read(char buffer[]) - it read up to the length of byte in buffer.
6. void reset() - it reads the input pointer.
7. long skip(long numBytes) - it returns the character ignored.
8. boolean ready() - if input request will not wait returns true, otherwise false.

Methods of Writer Abstract Class

1. write append(char ch) - It append the 'ch' character at last of output stream.
2. write append(char Sequence chars) - append the chars at the end of output stream.
3. abstract void close() - close the output stream.
4. abstract void flush() - it flush the output buffer.
5. void write(char buffer[]) - writes array of character to output stream.
6. void write(String str) - it writes a the str to the output stream

Pre-defined Streams

Java provides following three pre-defined streams.

Standard Input:

This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as System.in.

Standard Output:

This is used to output the data produced by the user's program and usually a computer screen is used to standard output stream, and represented as System.out.

Standard Error:

This is used to output the error data produced by the user's program and usually a computer screen is used to standard error stream and represented as System.err.

Opening & Closing I/O Streams

We open an I/O stream by constructing an instance of the stream. Both the Input Stream and the Output Stream provides a close() method to close the stream, which performs the necessary clean-up operations to free up the system resources.

```
public void close() throws IOException // close this Stream
```

classmate
Date
Page

It is a good pratice to explicity close the I/O stream, by running close() in the finally clause of try-catch finally to free up the system resources immediately when the stream is no longer needed. This could prevent serious resource leaks. Unfortunately, the close() method also throws a IO Exception, and needs to be enclosed in a nested try-catch statement, as follows

```
FileInput Stream in=null;
-----
try {
in = newFile Input Stream(---); // Open stream
 - - -
 - - -
} catch (IO Exception ex) {
 ex. print Stack Trace();
} finally { // always close the I/O stream
try {
   if (in != null) in.close();
} catch (IOException ex) {
  ex. print Stack Trace();
}
}
```

But in JDK 1.7, a new try-~~catch~~ with resources syntax, which automatically closes all the ~~operand~~ opened resources after try or catch, as follows,

```
try (File Input Stream in = new File Input Stream(---) {
 - - -
 - - -
} catch (IO Exception ex) {
   ex. print Stack Trace();
} // Automatically closes all opened resources in try(---)
```