Object Oriented Programming (17CS2664)

L-T-P-S : 3-0-2-0

Credits : 4

V. Deepak, 9080383080, deepak@bluehills.in, C321B

Syllabus:

Introduction:

Oop principles, Encapsulation, Inheritance and Polymorphism, Java as a OOPs & Internet Enabled language, The Byte code, Data types, Variables, Dynamic initialization, scope and life time of variables, Arrays, Operators, Control statements, Type Conversion and Casting, Compiling and running of simple Java program.

Classes and Objects:

Concepts of classes and objects, Declaring objects, Assigning Object Reference Variables, Methods, Construct-ors, Access control, Garbage collection, Usage of static with data and methods, usage of final with data, Overloading methods and constructors, parameter passing - call by value, recursion, nested classes.

Inheritance:

Inheritance Basics, member access rules, Usage of super key word, forms of inheritance method overriding, abstract classes, dynamic method dispatch, using final with inheritance.

String handling functions

Packages and Interfaces:

Packages, classpath, importing packages, differences b/w classes and interfaces, Implementing & Applying interface.

Exception Handling:

Exception Handling fundamentals, multi threaded programming, java input/output.

Collection framework and swing package based event driven programming.

18/12/19

Introduction to Object-Oriented Programming:

- C++, java : Object-Oriented language
  C : Procedure-oriented language.

Datatype:

- Data type specifies the size and type of values that can be stored in an identifier.
- The java language is rich in its data types.

Types of data types in java:

1. Primitive
2. Non-primitive

1. Primitive: Includes (i) integer
   (ii) Character
   (iii) Boolean
   (iv) Floating point

2. Non-primitive: Includes (i) classes
   (ii) interfaces
   (iii) Arrays

Conditional statements: if-else
else-if ladder
nested if-else

Looping statements: for
while
do-while

- Unconditional statements : (i) goto
(ii) break
(iii) continue

- Variables
- Arrays
- Operators
- Dynamic initialization

Difference b/w C and Java

C	Java
1. Procedural language	1. Object-oriented language
2. It is compiled	2. It is interpreted
3. Error crashes in C	3. Exception handling in Java
4. C supports preprocessors	4. Does not support preprocessors
5. Does not support overloading	5. Supports overloading
6. User based memory management	6. Memory management
7. Uses pointers	7. No use of pointers
8. Top down approach	8. Bottom Up approach
9. Low level language	9. High level language
10. Default members are public	10. Default members are private
11. Not portable	11. Portable

12. platform dependent	12. platform independent
13. Supports structure & union	13. Does not support structure & union.
14. Manual object management	14. It has garbage collector.
15. Supports goto statement	15. Does not supports goto statement
16. Does not support threads	16. Support threads
17. Supports multiple inheritance	17. Does not support multiple inheritance.
18. Supports call by value and call by reference	18. Supports only call by value
19. Supports virtual keyword.	19. Does not support virtual keyword
20. For generic type we use void*	20. For generic we use Object
21. Data hiding using static	21. Data hiding using private
22. Allocating memory using malloc	22. Allocating memory using new.
23. 32 keywords	23. 50 packages
24. Supports storage classes	24. Does not support storage classes.
25. Execution of program starts from main()	25. Execution of program starts from class( )
26. Extension filename: xy.c	26. class A ⇒ A.java

- Program size is less in java compared to C.

3 major families of languages are:

- Machine level language
- Assembly level language
- High-level language

Fundamental principles of OOP:

1. Encapsulation : Hide unnecessary details
2. Inheritance : How class hierarchies
3. Abstraction : Work through abstractions
4. Polymorphism : Abstract behaviour.

23/12/19

- OOPS can be implemented by C++ and java.

Java

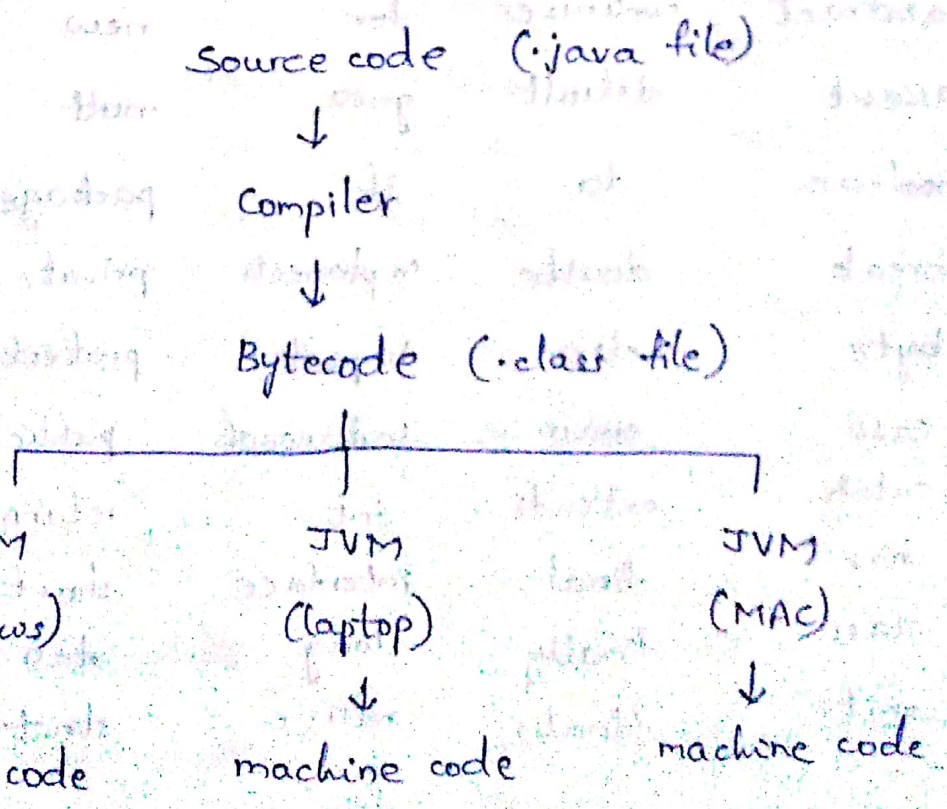
- Invented in 1991 by Sun Microsystems named by "Oak" (initially called) & was renamed as "Java" in 1995.
- Java is platform independent language.
- Latest version: 13
- Case-sensitive language
-
  - For source code → xyz.java
  - For compiling → javac xyz.java
  - For execution → java xyz

Java Virtual Machine (JVM):

- JVM is a virtual machine that enables a computer to run Java programs as well as programs written in other languages that are also compiled to Java bytecode.
- JVM is detailed by a specification that formally describes what is required in a JVM implementation.
- A program which runs precompiled java programs.

JVM procedure:

- Compile Java program from command prompt
- Run " " " "
- O/p of Java compiler is not executable (bytecode).
- Bytecode is highly optimized set of instructions to be executed by JRE called JVM.



Java Runtime Environment:

A s/w program needs to execute and to do that it needs an environment loads class files and ensures there is access to memory & other system resources to run them.

- Includes JVM core libraries.

Java Development Kit:

- JDK allows developers to create java programs that can be executed & run by JVM & JRE.

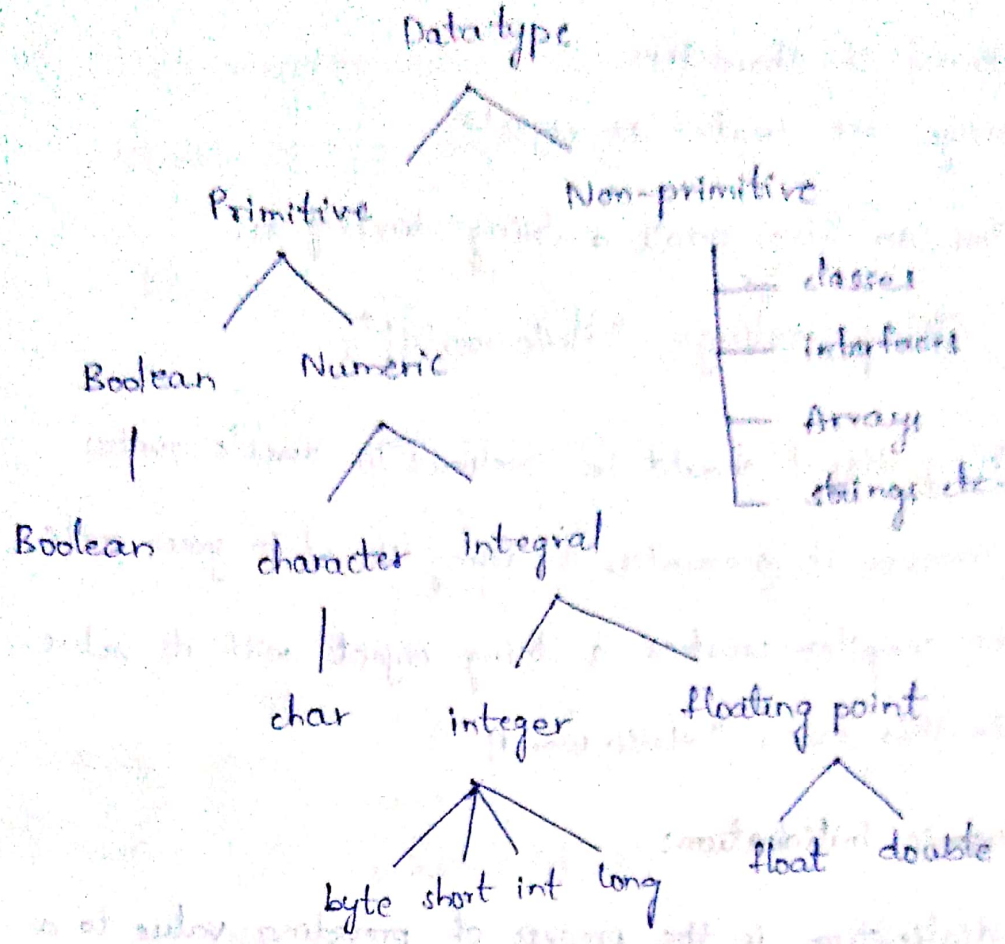
Keywords (50)

Can't be used as names, variable, class or method.

They are:

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

Data types:



Size:

Integers : byte - 8 bits
short - 16 bits
int - 32 bits
long - 64 bits

Floating point : float - 32 bits
double - 64 bits

Character literals:

\'	Single quote	\n	new line
\"	Double quotes	\f	form-feed
\\	Back slash	\t	tab
\r	Carriage return	\b	back space

String literals:

- Sequence of characters
- Strings are treated as objects.
- You can also create a string directly as:

```
String greeting = "Hello world!";
```

- String literal should be enclosed in double quotes
- Whenever it encounters a string literal in your code, the compiler creates a string object with its value. In this case, "Hello world!".

Dynamic initialisation:

- Initialisation is the process of providing value to a variable at declaration time.
- Initializing a variable at runtime is called dynamic initialization.

```
Ex: double a=3.0, b=4.0;
    // c is dynamically initialized
    double c= Math.sqrt(a*a+b*b);
```

Type conversion & Casting:

Rules:

1. Numeric variables are compatible.
2. Numeric variables are not compatible with char,

boolean

3. Char & Boolean are not compatible

4. Auto conversion takes place while assigning integers literals to byte, short, long

Ex: ① int a = char c; ✗

② int a = 100;
int b = a;
printf("%d", b);
} o/p: 100

③ int a;
float b;
printf("%d", b);

Array:

- Like typed variables referred by a common name
- May have one or more dimensions
- A specific element of an array is referred by an index
- Offers convenient means of grouping related info.

Syntax:

<datatype> <Variable name> [ ]; → index, by default '0'

- Only defines variables not the size
- Size is allocated by using a 'new' operator

```
int a[];
a = new int[10];
```

→ 2 types of declaring an array

- In Java, all arrays are dynamically allocated.

Ex: int a1;
a2;
a3;
a4;

a[1];
accessing time will be less in array.

- Array elements are referred by index
- Index value will start from zero.
- It is possible to define & allocate size simultaneously.

int monthdays [] = new int [12]

- Arrays can be initialised : (i) comma separated
(ii) enclosed in curly bracket

- Array size can be defined by initialisation

int monthdays [] = {31, 28, 31, ... }

- Java checks the boundaries at run time.

Ex: int a [5] = {3, 6, 7, 4, 1, 2, 9, 8, 23, 49};

⇓

It shows error

Multi-dimension array:

- In Java, multi-D arrays are actually arrays of arrays.
- There are couple of subtle differences.
- To declare a multi-D array variable, specify each additional index using another set.

1-D array:

```
class Array{
    public static void main (String args[]) {
        int month_days[];
        month_days = new int[12];
        month_days[0] = 31;
        month_days[1] = 28;
        month_days[2] = 31;
        month_days[3] = 30;
        month_days[4] = 31;
        month_days[5] = 30;
        month_days[6] = 31;
        month_days[7] = 31;
        month_days[8] = 30;
        month_days[9] = 31;
        month_days[10] = 30;
        month_days[11] = 31;
        System.out.println("April has " + month_days[3] + " days");
```

O/p: April has 30 days.

Java Variables:

Naming Rules & Conventions:

1. Variable names are case-sensitive.
2. Variable name can be, an unlimited-length sequence of unicode letters & digits.

3. Variable name must begin with either a letter or the dollar sign "$", or the underscore character "_".

4. Characters except the dollar sign "$" and the underscore character "_"

Ex: '+' or '?' or '@' are include invalid for a variable name.

5. No white space is permitted in variable names.

Ex: var 1 ✗

6. Variable name we choose must not be a keyword or reserved word.

Class

- A class is a group of objects which have common properties.
- It is a template or blue print from which objects are created.
- It is a logical entity.
- A class can contain:
  1. Fields / Variables
  2. Methods
  3. Constructors
  4. Blocks / Functions
  5. Nested class & interface

(not mandatory to be present in a class)

Syntax to declare class:

```
class <class_name>
{
    field;
}
```

- Objects are the instance of the class through which all the members & variables of the class can be accessed.

Syntax for declaring a object:

```
classname objectname
classname new = objectname classname
```

Ex:

```
class A
{
    int a=10;
    char b;
    float c;
    d_sum()
    {
        ____
        ____
        ____
    }
};
```

```
A obj1;        -> object of class A
obj1.a;        -> to print a
```

=> n no. of values & objects can be stored

```
obj2.a;        ✓ valid
```

=> n no. of fns, objects inside a class

30/12/19

Access specifiers in java

- Scope of a variable (or) the boundary of variable or fn, access specifiers

Access Modifiers	Default	Private	Protected	Public
- Accessible inside the class	Yes	Yes	Yes	Yes
- Accessible within the subclass inside the same package	Yes	No	Yes	Yes
- Accessible outside the package	No	No	No	Yes
- Accessible within the subclass outside the package	No	No	Yes	Yes

If we don't mention any access specifier, the compiler takes default

public:
int a=100
b=200
prevents unwanted

- Improving security in the code

Java Variable Types:

- Local variables
- Instance Variables or Non-static fields
- Class variables or static fields
- Method parameters

Program:

```
class A1
{
 int a123;  //Instance variables or non-static fields
 static int b123; // class variables or static fields
 int b1()
 {
  int c123;  //method parameter
   ------
   ------

 }
  public static void main (String args[])
  {
    int d123;
    c2()
    {
      int e123; //local variables
       ------
       ------
     }
   }
 }
```

- If int b1() is again declared in public then compiler will not show an error.

Constructor:

Fn inside class

fn name = class name

Operators

pre inc, post inc, pre dec, post dec

1. Arithmetic operators (+, -, *, /, %, ++, --)
4. Logical (&&, ||, !)

   Bitwise
3. Relational / Comparison (==, !=, >, <, >=, <=)
2. Assignment (=, +=, -=, *=, /=, %=, ^=)

Conditional statements

1. if
2. nested if
3. if-else
4. switch
5. nested switch

Looping statements

1. while
2. for
3. do-while

Syntax for conditional statements:

1. if

```
public class OddNumbers {

    public static void main (String args[])
    {
        int i;
        for (i=1; i<=100; i++)
        {
            System.out.println
        }
    }
```

O/P:

Classes and Objects:

Class

- New data type
- Objects of new data type can be created
- Class is a encapsulated version of methods and variants to access all the variables or to call fn's which is present inside the class, we require objects.
- Objects are generally known as an instance of a class.
- You can create 'n' no of objects for any particular class.
- It is a template

Syntax for creating an object

classname objectname = new classname()

- The data or variables defined within a class are called instance variables
- Code is contained in the methods.
- Methods & variables defined in the class are cass members of a class.
- Copy of variables is created every time an object of the class is created.
- Any class must be entirely defined in a single source file. ⇒ should have definition

Ex: class A

```
{
    a1()
    {

    }
    a2()
    {

    }
}
```

compilation error → a3()
a4()
{
}
}

Java program using class and object

```
class Box {
    double width;
    double height;
    double depth;
}
// This class declares an object of type Box
class BoxDemo {
    public static void main (String args[]) {
        Box mybox = new Box();
        double vol;
        // assign values to mybox's instance variables
        mybox.width = 10;
        mybox.height = 20;
        mybox.depth = 15;
        // compute vol of box
        vol = mybox.width * mybox.height * mybox.depth;
        System.out.println("Volume is " + vol);
    }
}
```

o/p: Volume is 3000

```
Box mybox[100] = new Box();
```

↓

100 objects

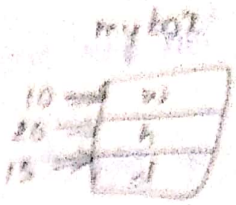
↓

can store 300 values

Java program using class and object

```
class Box {
    double width
    double height;
    double depth
}
```

① } instance variables



If we create another mybox


```
// This class declares an object of type Box
class BoxDemo {

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

② ↓ main is written only once

object for 1st class ← `Box mybox = new Box();`

```
        double vol;
        // assign values to mybox's instance variables
        mybox.width = 10;
        mybox.height = 20
        mybox.depth = 15
        vol = mybox.width * mybox.height * mybox.depth;
// compare vol of box
        System.out.println ("Volume is " + vol);

    }
}
```

O/P: Volume is 3000.

→ `Box mybox[100] = new Box();`

⇓

100 objects

⇓

can store 300 values

Simple class

- You can create an instance of a class, you are creating an object, that contains its own copy of each instance variable defined by the class.

Assigning Object Reference Variables

class object name class name

Box b1 = new Box();

object name Box b2 = b1;

After this fragment executes, b1 and b2 will both refer to same object.

- The assignment of b1 & b2 did not allocate any memory or copy any part of the original object.
- It simply makes b2 refer to same object as does b1.
- Thus, any changes made to the object through b2 will effect the object to which b1 is referring.
- Although b1 & b2 having san

To avoid this use

b2 = null;

Introducing methods

```
class Box {
        double width;
        double height;
        double depth;

// display volume of a box

void volume() {
        System.out.println("Volume is ");
        System.out.println(width * height * depth);
    }
}

class BoxDemo3 {
public static void main(String args[]){
Box mybox1 = new Box();
Box mybox2 = new Box();
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
// assign diff values to mybox2's instance variables
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;

mybox1.volume(); // display vol of first box
mybox2.volume(); // display vol of second box
}
}
```

Area & perimeter of a circle.

```
import java.util.*;
class AreaOfCircle
{
    private float radius = 0.0f;
    private float area = 0.0f;
    private float perimeter = 0.0f;

    public void readRadius()
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter radius:");
        radius = sc.nextFloat();
    }
    public float getArea()
    {
        area = 3.14 * radius * radius;
        return area;
    }
    public float getPerimeter()
    {
        perimeter = 2 * 3.14 * radius;
        return perimeter;
    }
}
public class circle
{
```

```
public static void main(String args[])
  {
    AreaOfCircle area = new AreaOfCircle();
     area.readRadius();
    System.out.println("Area of circle:"+area.getArea());
    System.out.println("Perimeter of circle:" +
                                area.getPerimeter());

          }
  }
```

O/p: Enter radius: 5

Area of circle: 78.50

Perimeter of circle: 31.4

No. of digits in a number

→
```
import java.util.*;
class DigitsOpr
  {
   public int num;
   public void getNum(int x)
                num=x;
     public int countDigits()
     {
       int n, count;
       n=num;
       count=0;
       while (n>0)
```

```
{
    n/=10;
    count++;
  }
  return count;
 }
}
public class number
{
  public static void main(String args[])
  {
    DigitsOpr dig = new DigitsOpr();
    int n;
    Scanner sc= new Scanner(System.in);
    System.out.println("Enter an +ve integer
                        number:");
    n= sc.nextInt();
      dig.getNum(n);
    System.out.println("Total no.of digits are:"
                       + dig.countDigits());
  }
}
```

dig → object, countDigits → method name

o/p: Total no.of digits are: 4

10/1/20

Static variable & static method:

- Static variable in Java is variable which belongs to the class & initialised only once at the start of the execution.
- It is a variable which belongs to the class and not to object (instance).
- static variables are initialised only once, at the start of the execution. These variables will be initialised first, before initialization of any instance variables
- A single copy to be shared by all instances of the class
- A static variable can be accessed directly by the class name and doesn't need any object.

Syntax: <class-name>.<variable-name>

Ex: static class A {

static int a=10;

static a();

- static variable → method →
- To call or access a non-static variable, we need object

- Normal procedure — 6 values

```
class A
{
  int a,b,c;
  ---
  ---
}

class B
{
  A a1=new A()
  A a2=new A()
}
```

a1: a, b, c

a2: a, b, c

- In static, 3 values are taken

a1, a2: a, b, c

Static method:

- Static method in Java is a method which belongs to the class and not to object.
- A static data method can access only static data.
- It is a method which belongs to class not to object.
- A static method can access static variables

Ex program for static variable and static method

```
public class StaticDemo{        //class
  int a;                        // non-static variable
  static int b;                 // static variable
  void init(int p, int q)       // 2 non-static methods
  {
  a=p;
  b=q;
  }
  void showa()
   {
    System.out.println(a);
   }
  static void showb()           // static method
   {
    System.out.println(b);
   }
   public static void main(String n[]){
    StaticDemo s=new StaticDemo();
   s.init(45,55);
   s.showa();
    StaticDemo.showb();
   s.showb();
    }
  }
```

O/p: 45 55 55

Example

```
class VariableDemo                    // declaring class
{
    static int count=0;               // : static variable
    public void increment()           // non-static method
    {
      count++;
    }
    public static void main(String args[])
    {
      VariableDemo obj1 = new VariableDemo();
      VariableDemo obj2 = new VariableDemo();

        obj1.increment();
        obj2.increment();

      System.out.println("obj1: count is = "+obj1.count);
    }
```

o/p: 2 2

- If static is removed then

```
int count=0;
```

o/p: 1 2

Differences b/w static & non-static variables

Static variable	Non-static variable
1. SV can be accessed using class name	1. NSV can be accessed using instance of a class.
2. SV can be accessed by static and non-static methods.	2. NSV cannot be accessed inside a static method.
3. SV reduce the amount of memory used by a program.	3. NSV do not reduce the amount of memory used by a program.
4. SV are shared among all instances of a class.	4. NSV are specific to that instance of a class.
5. SV is like a global variable and is available to all methods.	5. NSV is like a local variable & they can be accessed through only instance of a class.

21/1/20 (Monday) → Submission

Home Assignment #1:

Program to demonstrate type casting & automatic type conversion in java.

ALM #1:

Program to demonstrate control & loop statements in java.

Call by value and Call by reference in Java

- Call by value means calling a method with a parameter as value. Through this, the argument value is passed to the parameter.

- While call by reference means calling a method with parameter as a reference. Through this, the argument reference is passed to the parameter.

Call by value program:

```
class CallByValue {
    public static void main (String [] args) {
    int x=3;
    System.out.println ("Value of x before inc() is"+x);
      increment (x);
      System.out.println ("Value of x after inc() is", +x)
    }
    public static void increment (int a) {
    System.out.println ("Value of a before
    incrementing is "+a);
        a=a+1;
```

```
System.out.println("Value of a after incrementing a"
+a);
}
}
```

Call by reference program:

```
class Number{
    int x;
}

class CallByReference{
    public static void main(String[] args){
    Number a=new Number();
    a.x=3;
    System.out.println("Value of a.x before calling
    increment() is"+a.x);
      increment(a);
    System.out.println("Value of a.x after calling
    increment() is "+a.x);
    }
    public static void increment(Number n){
    System.out.println("Value of n.x before incrementing
    x is"+n.x);
    n.x=n.x+1;
    System.out.println("Value of n.x after incrementing
    x is"+n.x);
    }
}
```

Constructors

- A constructor initialise an object immediately upon creation.
- It has the same name as the class in which it resides & is syntactically similar to a method.

```
class A
{
  A()
  {
    —
    —
  }
  A()
  {
    —
    —
  }
  A()
  {
    —
    —
  }
}
```

- Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes.

- Constructors look little strange because they have no return type, not even void.
- This is because the implicit return type of a class constructor is the class type itself.
- Constructor creates a fully initialized, usable object.

Rules for creating Java constructor

- Constructor name must be the same as its class name.
- A constructor must have no explicit return type
- A Java constructor cannot be abstract, static, final and synchronized

3 types of constructor

1. Default constructor
2. No-Args constructor
3. Parameterized constructor

1. Default constructor

Syntax:

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

```
    Data d= new Data();
    }
}
```

- It is not req to

No-Args Constructor

Syntax:

```
public class Data
{
  //no-args constructor public Data()
    Data()
    {
     System.out.println("No-Args Constructor");
     }

     public static void main (String[] args)
     {
       Data d = new Data();
       }
    }
```

O/p : No-Args Constructor

Parameterized Constructor

Syntax:

```
public class Data
{
```

```
public Data (int n)
{
System.out.println ("Parameterised constructor");
}
public static void main (String[] args)
 {
  Data d= new Data("1234");
  }
}
```

Ex

①
```
class Demo {
    int value1;
    int value2;
    Demo() {
      value1 = 10;
      value2 = 20;
       System.out.println ("Inside Constructor");
      }
    public void display()
    {
    System.out.println ("Value1 === " + value1);
    System.out.println ("Value2 === " + value2);
    }
    public static void main (String args[])
      {
```

```
Demo d1 = new Demo();
d1.display();
}
}
```

(2) No Args

```
class consMain() {
int x;
consMain() {
System.out.println("Const called");
x=5;
}
public static void main (String args[]) {
consMain obj = new consMain();
System.out.println("Value of x = " +obj.x);
}
}
```

(3)

```
public class constDemo1 {
int a,b;
constDemo1()
{
a=55;
b=66;
}
```

```
void show()
{
 System.out.println("a="+a\n "b="+b);
}
public static void main (String a[])
{
  constDemo1 obj1 = new constDemo1();
  obj1.show();
  new constDemo1[].show();
 }
```

④ Cons overloading

```
public class constDemo3{
int a,b;
float s;
 constDemo3(int p, int q)
  {
    a=p;
    b=q;
  }
 constDemo()
  {
    a=b=100;
   }
 constDemo3(float p) {
        s=p;
    }
 void show() {
  System.out.println("a="+a+\n "b="+b\n "c="+s);
 }
```

```
public static void main (String n[])
 {
   new constDemo3 (99,33). show();
   new constDemo3 ( ). show();
   new constDemo3 (0.88F). show();
  }
}
```

O/p:

a=99
b=33

a=100
b=100

s=0.88

1/20

Garbage collection

- In java, garbage means unreferenced objects
- GC is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects
- To do so, we were using free() fn in C language & delete() in C++.
- But in java it is performed automatically. So, java provides better memory management.

Advantages of garbage collection:

- It makes java memory efficient bcoz GC removes the unreferenced objects.
- It is automatically done by the GC (a part of JVM) so we ~~don't~~ need to make extra efforts.

How can an object be unreferenced?

- There are many ways

(i) By nulling the reference

(ii) By assigning a ref to another

(iii) By anonymous object etc.

1. By nulling a reference

```
Employee e = new Employee();
e = null;
```

2. By assigning a ref to another:

```
Employee e1=new Employee()
Employee
e1=e2    //now the 1st obj ref by e1 is available
                                    for GC
```

3. By anonymous object

```
new Employee();
```

Usage of "FINAL" keyword

The final keyword in java is used to restrict the user.
The java final keyword can be used in many content.

Final can be : (i) Variable
(ii) Method
(iii) Class

Final variable ⟶ To create constant variables
Final methods ⟶ Prevent method overriding
Final classes ⟶ Prevent Inheritance

Program:

```
class Demo
{
  final Int MAX_VALUE=99;
  void myMethod()
  {
    MAX_VALUE=101;
  }
```

```
public static void main (String args[])
{
    Demo obj = new Demo();
    obj.myMethod();
}
}
```

```
public class MyClass
{
    final int x = 10;
    public static void main (String[] args)
    {
        MyClass myObj = new MyClass();
        myObj.x = 25; // will generate an error: cannot assign a value to final variable x
        System.out.println(myObj.x);
    }
}
```

Recursion:

- Java supports recursion.
- Recursion is the process of defining something in terms of itself.
- As it relates to Java programming, recursion is the process that allows a method to call itself.
- A method that calls itself is said to be recursive.

Program:

```
public class RecursionExample2 {
 static int count=0;
 static void p() {
 count++;
  if (count<=5) {
  System.out.println("hello" + count);
  p();
  }
  }
  public static void main (String[] args) {
  p();
  }
 }
```

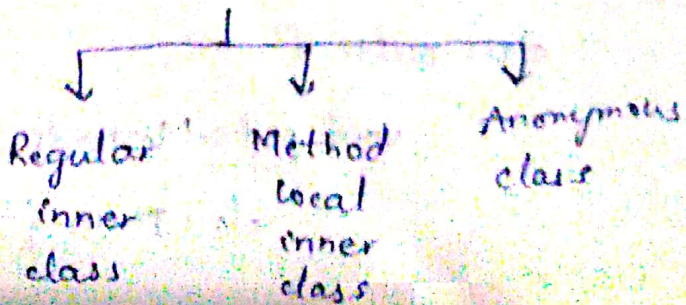
Nested Classes

- Class inside another class

```
class otherclass {
        class nested class {
                }
        }
```

- There are 2 types of nested classes

1. Static
2. Non-static



Regular inner class:

```
class outer {
        class inner {
            void innerMethod() {
              System.out.println("inner class method");
            }
        }
    void outerMethod() {
     System.out.println("Outer class method");
       inner i= new Inner();
            i.innermethod();
        }
    public static void main (String args[])
     {
       outer o = new outer();
          o.outerMethod();
          }
      }
```

Output: Outerclass method
inner class method

Method Local inner class:

```
class outer {
      void outerMethod() {
          class Inner {
```

```
void Innermethod() {
    System.out.println("Inner class method");
    }
Inner i = new Inner();
      i = Innermethod();
  }
public static void main(String args[])
{
  outer o = new outer();
      o.outerMethod();
    }
  }
```

Inheritance

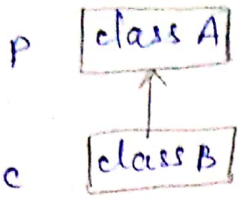
- Inheritance can be defined as the process where one class acquires the properties (methods & fields) of another.

- Syntax:

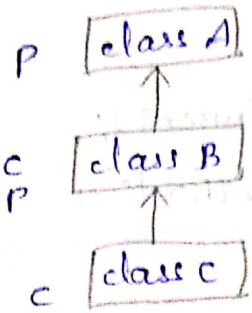
```
class super
 {
  ...
 }
 class sub extend super
  {
   ...
  }
```

- Types of inheritance in java:

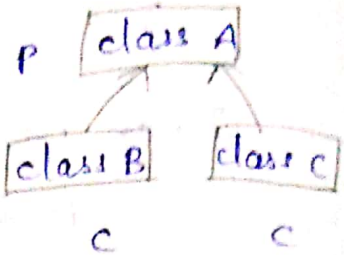
1. Single



2. Multi-level



3. Hierarchical



4. Hybrid

Ex: ① 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();
}
}
```

② Example for multi-level inheritance

```
class X
{
  public void method X()
  {
    System.out.println("class X method");
  }
}
class Y extends X
  {
    public void method Y() {
      System.out.println("class Y method");
    }
  }
class Z extends Y {
    public void method Z() {
      System.out.println("class Z method");
    }
public static void main (String args[])
{
   Z obj = new Z();
  obj.method X(); // calling grandparent class method
  obj.method Y(); // calling parent class method
  obj.method Z(); // calling local method
 }
}
```

O/p : class X method
class Y method
class Z method

(3)

```
class Hierarchicalinheritance{
    void Display A() {
        System.out.println("This is a content of
                            parent class");
    }
}
class A extends Hierarchicalinheritance {
    void Display B() {
        System.out.println("This is a content of
                            child class1");
    }
}
class B extends Hierarchicalinheritance {
    void display() {
        System.out.println("This is a content
                            of child class 2");
    }
}
```

```
class Hierarchicalinheritance main {
    public static void main (String args[]) {
        S.O.P ("calling for child class C");
        B.b = new B();
        b.display A();
        b.display C();
        S.O.P ("calling for child class A");
        A.a = new A();
        a.display A();
        a.display B();
```

2/2/20

Member Access Rules

Access Specifiers ↓	Members of same class	Members of sub-class present in same folder	Members of different classes present in same folder	Members of sub-class present in diff folder	Members of diff classes present in diff folder
Private	✓	×	×	×	×
Protected	✓	✓	✓	✓	×
Public	✓	✓	✓	✓	✓
friendly/ default	✓	✓	✓	×	×
Private-protected	✓	✓	×	✓	×

Practices Question-1

Create a class with a method that prints "This a parent class" and its subclass with another method that prints "This is child class". Now, create an object for each of the class & call

1- method of parent class by object of parent class

2- method of child class by object of child class

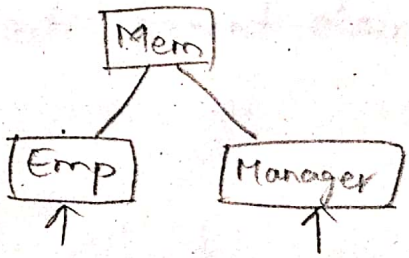
3- method of parent class by object of child class

2. Create a class named "Member" having the following members:

Data members:

1- Name
2- Age
3- Phone number
4. Address
5. Salary

It also has a method named 'printSalary' which prints the salary of the members. Two classes 'Employee' and 'Manager' inherits the 'Member' class. The 'Employee' and 'Manager' classes have data members 'specialization' and 'department' respectively. Now, assign name, age, phone number, address and salary to an employee and a manager by making an object of both of these classes and print the same.



Java super keyword & Constructors

- The super keyword in Java is a reference variable which is used to refer immediate parent class object.
- Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

- Super can be used to refer immediate parent class instance variable
- super can be used to invoke immediate parent class method.
- super() can be used to invoke immediate parent class constructor.

Ex:

```
class A
{
  int a1;
  ___
  ___
}
class B extends A
{
  int a1;
  ___
  s.o.p(___);
}
```

```
class C
{
  b b1;
  b1.a1 = 10;
}
```

value is stored in child class but not in parent class

In order to use in parent class we use 'super' keyword

Example programs in super keyword using methods:

① class Superclass

```
{
  int num=100;
}
class Subclass extends Superclass
  {
    int num=110;
    void printNumber() {
        System.out.println(super.num);
    }
    public static vo
        Subclass obj=new Subclass();
        obj.printname
    }
  }
```

o/p: 100

②

```
class Person
  {
    void message() {
      System.out.println("This is person class");
    }
  }
class Student extends Person
  {
```

```
void message()
 {
  System.out.println("This is student class");
 }void display()
  {
   message();
   super.message(); }
  class Test
   {
    public static void main(String args[])
     {
      Student s=new Student();
     s.display();
     }
   }
```

o/p: This is student class
This is person class

- The super keyword can also be used to invoke or call the parent class constructor.
- To establish the connection b/w base class constructor & derived class constructors JVM provides two implicit methods they are:

Super() → default constructor
Super(__) → parameterized constructor

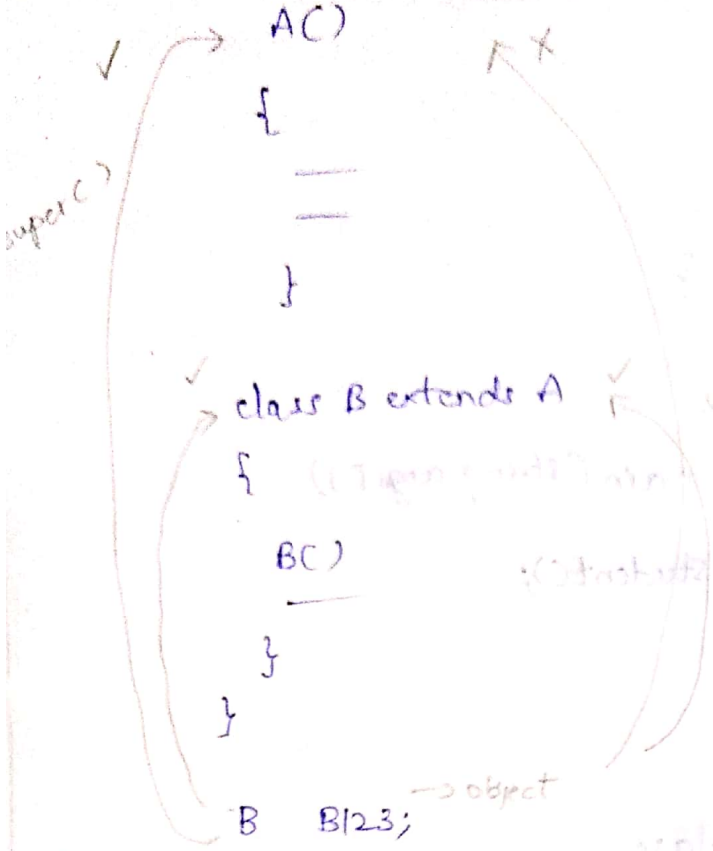
- Super() used for calling super class default constructor from the context of derived class constructors.

Ex:

```
class A
{
   A()
   {
   ——
   ——
   }

class B extends A
{
   B()
   ——
   }
}

B  B123;
```




```
class Employee
{
   Employee()
   {
    System.out.println("Employee class Constructor");
   }
   }

   class HR extends Employee
   {
      HR()
      {
        super();
        S.O.P("HR class constructor");
```

```
class Employee
{
 Employee()
{
 S.O.P("Em:
}
}
class Supercons
{
public static void
```

```
class Base
{
 int a;
 Base()
  {
   a=5;
  }
  Base(int p)
    {
     a=p;
  }
  void show()
  {
   S.O.P("a in Base class Accessed from base class
                 method ="+a);
  }
}
```

```
class Derived extends Base
{
  int a;
  Derived()
  {
    a=10;
  }
  Derived(int p,int q)
  {
    super(p);
    a=q;
  }
  void show()
  {
    super.show();
    S.O.P("a is Derived class Accessed from Derived
    class method="+a);
    S.O.P("a in Base class Accessed from Derived
    class method="+super.a);
  }
}

public class SprDemo{
    public static void main(String args[])
    {
      Derived d1=new Derived();
```

d1.show();
d2.show(); → Derived d2 = new Derived (100, 200);

}
}

O/P: 5 10 5 100 200 100

4/8/20

Method overriding

- Declaring a method in sub class which is already present in parent class is known as method overriding
- Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class.
- In this case the method in parent class is called overridden method & the method in child class is called overriding method.

Example:

```
class Vehicle {
    void run()
    {
      s.o.p("Vehicle is running");
    }
}
class Bike2 extends Vehicle {
    void run()
    {
        s.o.p("Bike is running safely");
```

```
}
public static void main (String args[])
{
Bike2 obj = new Bike2();
obj.run();
}
}
```

O/p : Bike is running safely

Rules for Method Overriding:

- The argument list should be exactly the same as that of overridden method.
- The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass
- The access level cannot be more restrictive than the overridden method's access level.

  For ex: If the superclass method is declared public then the overriding method in the subclass cannot be either private or protected.

- Instance methods can be overridden only if they

are inherited by the subclass.

- A method declared final cannot be overridden.
- A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
- A subclass in a different package can only override the non-final methods declared public or protected.
- Constructors cannot be overridden.

Method Overriding & Dynamic method dispatch

```
class ABC{ //overridden method
  public void disp()
  {
    System.out.println("disp() method of parent class");
  }
}
class Demo extends ABC{
  //overriding method
  public void disp(){
      s.o.p("disp() method of child class");
  }
  public void newMethod(){
      s.o.p("new method of child class");
  }
```

```
public static void main (String args[]) {
/* When parent class ref refers to parent class object then
   in this case overridden method (the method of parent
   class) is called */
    ABC obj = new ABC();
    obj.disp();
/* When parent class ref refers to child class object then
   overriding method (method of child class) is called.
   This is called dynamic method dispatch & runtime
   polymorphism */
    ABC obj2 = new Demo();
    obj2.disp()
  }
}
```

o/p : disp() method of parent class
disp() method of child class

17/2/20

Abstract Class:

- Abstraction is another important OOP's principle in Java.
- It is the process of hiding internal implementation details from the user & providing only necessary functionality to the users. It removes all non-essential things and shows only imp things to user.

- A class that is declared using "abstract" keyword is known as abstract class.
- An abstract class can contain both abstract method and a non-static abstract method.
- A non-abstract class can not contain abstract method.
- An abstract class cannot have an object.
- A child class should define all the methods of the abstract base class.
- A abstract method can only be declared not defined. (It have to defined in child class).

Ex: Difference b/w definition & declaration

```
class A
{
  int a1()
  {
    ==
    ==
  }
  int a2();
}
```

A method having set of instructions to execute is called method definition

→ No set of instructions to execute only declaration with ;

Conditions for abstract class in Java

- An abstract class is a class, which is declared with abstract keyword.
- It is just like a normal class but has 2 differences:
  1. We cannot create an object of this class. Only objects of its non-abstract (or concrete) sub-classes can be created.
  2. It can have zero or more abstract methods which are not allowed in a non-abstract class (concrete class)

Keypoints:

1. Abstract is a non-access modifier in Java which is applicable for classes, interfaces, methods & inner classes

Abstract method in Java

- A method which is declared with abstract keyword & has no implementation

Uses of abstract method in Java:

1. An abstract method can be used when the same method has to perform different tasks

2.

Example program:

```
//abstract parent class
    abstract class Animal {
// abstract method
     public abstract void sound();
    }
    // Dog class extends Animal class
public class Dog extends Animal {
     public void sound() {
          System.out.println("woof");
     }
     public static void main(String args[]) {
Dog  Animal obj = new Dog();
          obj.sound();
     }
}
```

correct → ⇒ woof

O/P: No o/p because class name is not Animal.

Example program using Abstract keyword:

```
abstract class AbstractDemo {
     public void myMethod() {
          System.out.println("Hello");
     }
```

```
abstract public void anotherMethod();
{
public class Demo extends AbstractDemo{
    public void anotherMethod(){
        S.O.P("Abstract method");
    }
    public static void main(String args[])
    {
        AbstractDemo obj = new AbstractDemo();
        obj.anotherMethod();
    }
```

Using final with inheritance:

There are 3 distinct meanings for the final keyword in Java.

1. A final class cannot be extended
2. A final method cannot be overridden
3. A final variable cannot be assigned to after it has been initialized.

Example program using both final with inheritance

```
class Parent {
/* Creation of final method parent of void type tells the
   implementation of this method is fixed throughout
   all the derived classes or not overridden */
```

```
final void parent(){
    S.O.P("Hello, we are in parent method");
  }
}

class Child extends Parent {
    void child() {
        S.O.P("Hello, we are in child method");
    }
}

class Test {
    public static void main (String[] args) {
        Parent p = new Parent();
        p.parent();
        Child c = new child();
        c.child();
        c.parent();
    }
```

18/2/20

String handling functions

There are 2 ways to create a string in Java

1. String literal
2. Using new keyword

1. String literal

String str1 = "Welcome";
String str2 = "Welcome";

2. Using New keyword

String str1 = new String ("Welcome");
String str2 = new String ("Welcome");

- String class falls under java.lang.String hierarchy.
But there is no need to import this class. java platform provides them automatically.

public char charAt(int index)
method returns the character located at the String's specified index

String x = "airplane"
System.o.p (x.charAt(2)); // o/p is 'r'

public String concat (String s)
method returns a string with the value of the String passed in to the method appended to the end of the String.

String x = "book"
S.O.P (x.concat ("author")); // o/p is "book author"

public boolean equalsIgnoreCase (String s)
method returns a boolean value (true or false) depending on whether the value of the String is same or not.

```
String x = "Exit"
s.o.p(x.equalsIgnoreCase("EXIT"));  // is "true"
s.o.p(x.equalsIgnoreCase("Hike"));  // is "false"
```

public int length()

method returns the length of the String

```
String x = "01234567";
s.o.p(x.length());    //return "8"
```

public String toUpperCase()

```
String x = "A New Java Book"
s.o.p("x.toUpperCase();
```

o/p: "A NEW JAVA BOOK"

public String toLowerCase()

```
String x = "A New Java Book";
s.o.p(x.toLowerCase());
```

o/p: "a new java book"

public String replace(char old, char new)

```
String x = "oxoxoxox";
s.o.p(x.replace('x', 'X'));
```

o/p: "oXoXoXoX"

public boolean contains("searchString")

This method returns true of target String is containing search String provided in the argument.

String x = "Java is programming language"

S.O.P(x.contains("Amit"));

↳ This will print false

S.O.P(x.contains("Java"));

↳ This will print true

public String substring(int begin)

public String substring(int begin, int end)

The substring() method

Example program for all String handling functions.

```
Public class StringMethodsDemo {
 public static void main (String[] args) {
   String targetString = "Java is fun to learn";
   String s1 = "JAVA";
   String s2 = "Java";
   String s3 = "Hello Java";
   S.O.P("Char at index 2 (third position): "
         + targetString.charAt(2));
   S.O.P ("After concat : " + targetString.concat ("Enjoy"));
   S.O.P ("Checking equals ignoring case : "
         + s2.equalsIgnoreCase(s1));
   S.O.P ("Checking equals with case : " + s2.equals(s1));
   S.O.P ("Checking length : " + targetString.length());
   S.O.P ("Replace function : " + targetString.replace
         ("fun", "easy"));
   S.O.P ("SubString of targetString : "
         + targetString.substring(8));
   S.O.P ("SubString of targetString : "
         + targetString.substring(8,13));
   S.O.P ("Converting to lower case : "
         + targetString.toLowerCase());
   S.O.P ("Converting to upper case : "
```

+ targetString.toUpperCase());

Output:

V

Java is fun to learn Enjoy

True

False

20

Java is easy to learn

fun to learn

fun t

Java is fun to learn

JAVA IS FUN TO LEARN

```
import java.io.*;
public class File1
{
  public sta
  {
    try {
    FileInputStream
```

reserved space or reserved capacity

StringBuffer class in Java

- StringBuffer class is used to create a mutable (Usable to change) string object i.e., its state can be changed after it is created.
- It represents growable & writable character sequence.
- As we know that String objects are immutable, so if we do alot of changes with String object

we will end up with alot of memory leak.

- So StringBuffer class is used when we must make alot of modifications to our string. It is also thread safe ie, multiple threads cannot access it simultaneously.

- StringBuffer defines 4 constructors. They are:

  StringBuffer()

  StringBuffer (int size)

  StringBuffer (String str)

  StringBuffer (charSequence[] ch)

- StringBuffer() : creates an empty string buffer & reserves room for 16 characters.

- StringBuffer (int size):

- Differences b/w String & StringBuffer.

Important methods of stringBuffer class

append()

insert()

replace()

reverse()

delete()

capacity()

ALM-2 : By Wednesday

Explain in detail about the 'String Buffer class', with example program, advantages & disadvantages.

HA-2

Explain the concept of CallByValue & CallByReference Its advantages & disadvantages & example program.

25/8/20

CO-3

Packages:

- Pack (group) of classes, interfaces & other packages
- In Java, we use packages to organize our classes & interfaces.
- Types of packages in Java
  1. User defined package
     The package we create
  2. Built-in package
     Already defined package like java.io*, java.lang etc.

Advantages of using a package in Java:

- Reusability
- Better organization
- Name conflicts

Built-in packages in Java:

lang, io, awt, math, sql, util, time, rmi, security, net, nio, javax, text, applet

Example program

```
package letmecalculate;
public class Calculator
 {
   public int add (int a, int b)
    {
      return a+b;
    }
   public static void main (String args[])
    {
```

```
Calculator obj = new Calculator();
S.O.P (obj.add (10,20));
 }
}
```

```
import letmeCalculate.Calculator;
public class Demo{
   public static void main (String args[])
    {
     Calculator obj = new Calculator();
        S.O.P (obj.add (100,200));
    }
}
```

Ex 2: Creating a class inside package while importing another package

```
package anotherpackage;
import letmecalculate.Calculator;
public class Example {
   public static void main (String args[])
  {
   Calculator obj = new Calculator();
      S.O.P(obj.add (100,200));
   }
  }
```

Sub package : A package inside another package

Interface in Java

- Interface looks like a class, but it is not a class.
- An interface can have methods & variables just like a class, but the methods declared in interface are by default abstract (only method signature, no body).
- Also, the variables declared in an interface are public, static & final by default.

Syntax:

Interfaces are declared by specifying a keyword "interface".

Ex:
```
interface MyInterface
{
/* All the methods are public abstract by default.
 As you see they have no body */

  public void method1();
  public void method2();
}
```

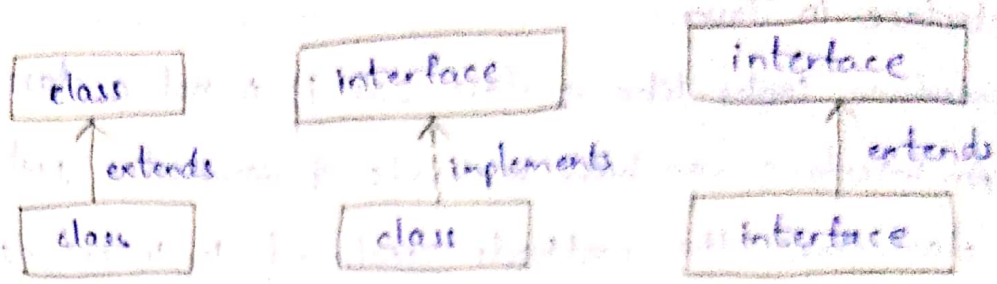
Why use java interface?

There are mainly 3 reasons to use interface.

They are:

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

Relationship b/w classes & interfaces:



Example of "interface"

①
```
interface printable {
  void print();
}
class AG implements printable {
  public void print()
  {
    S.O.P ("Hello");
  }
public static void main (String args[]) {
    AG obj = new AG();
    obj.print();
  }
}
```

O/P : Hello

②
```
interface Drawable {
  void draw();
}
// Implementation by second user
  class Rectangle implements Drawable
  {
```

```
public void draw()
{
  s.o.p ("drawing rectangle");
 }
}
class Circle implements Drawable {
   public void draw() {
    s.o.p ("drawing circle");
    }
   }
```

//Using interface by third user

```
    class TestInterface1 {
    public static void main (String args[]) {
      Drawable d= new Circle();  // In real scenario
      d.draw();
      }
     }
```

o/p: drawing circle

26/2/20

Multiple inheritance in Java by interface

```
  Interface Printable {
   void print();
   }
  interface Showable{
   void show();
  }
  class A7 implements Printable, Showable {
   public void print()
```

```
    S.O.P("Hello");
    }
  public void show()
  {
     S.O.P("Welcome");
    }
  public static void main (String args[]) {
    A7 obj = new A7();
    obj.print();
    obj.show();
    }
  }
```

O/P: Hello
Welcome

Differences b/w Abstract class & Interface

Abstract class	Interface
1. It cannot support multiple inheritances	
2. It contains both abstract & non-abstract method.	
3. Abstract class is partialy implemented class.	
4. It can have main method & constructor.	

5. It can static, non-static, final & non-final methods

What is CLASSPATH in Java (Setting classpath in Java)

- Classpath in Java is the path to directory or list of the directory which is used by ClassLoaders to find and load class in Java program.

1. Environmental variable
2. Command prompt

Setting in environmental variable

C drive

↓

Program files

↓

File

↓

JDK

↓

bin

↓

copy location

↓

This PC

↳

Setting in command prompt:

set CLASSPATH=%CLASSPATH%;JAVA_HOME\lib;

Differences b/w classes and interfaces:

CLASS	INTERFACE
1. Supports only multilevel & hierarchical inheritances but not multiple inheritance.	1. Supports all types of inheritance multilevel, hierarchical & multiple
2. "extends" keyword should be used to inherit.	2. "implements" keyword should be used to inherit.
3. Should contain only concrete methods (methods with body).	3. Should contain only abstract methods (methods without body).
4. The methods can be of any access specifier (all the 4 types)	4. The access specifier must be public only.
5. Methods can be final & static.	5. Methods should not be final & static.
6. Variables can be private	6. Variables should be public only.
7. Can have constructors.	7. Cannot have constructors.
8. Can have main() method.	8. Cannot have main() method as main() is a concrete method.

Practice Question-1:

1. Write a java program to create abstract class Car, with instance variable regno, Car constructor, concrete

method fillTank(), abstract methods steering(int direction) braking(int force). Create two subclasses Maruthi and Santro to implements abstract methods in it. Create a AbstractDemo class to for testing.

```
abstract class Car {
    int regno;
```

Exception handling in Java

- Exception handling is one of the most important features of Java programming that allows us to handle the runtime errors caused by exceptions.

What is an exception?

- An exception is an unwanted event that interrupts the normal flow of the program.
- When an exception occurs program execution gets terminated.
- In such case we get a system generated error msg.
- The good thing about exceptions is that they can be handled in Java.
- By handling the exceptions we can provide a meaningful message to the user about the issue rather than a system generated msg, which may not be understandable to a user.

Difference b/w error and exception

- Errors indicate that stg severe enough has gone wrong, the app^n should crash rather than try to handle the error.
- Exceptions are events that occurs in the code. A programmer can handle such conditions and take necessary corrective actions.

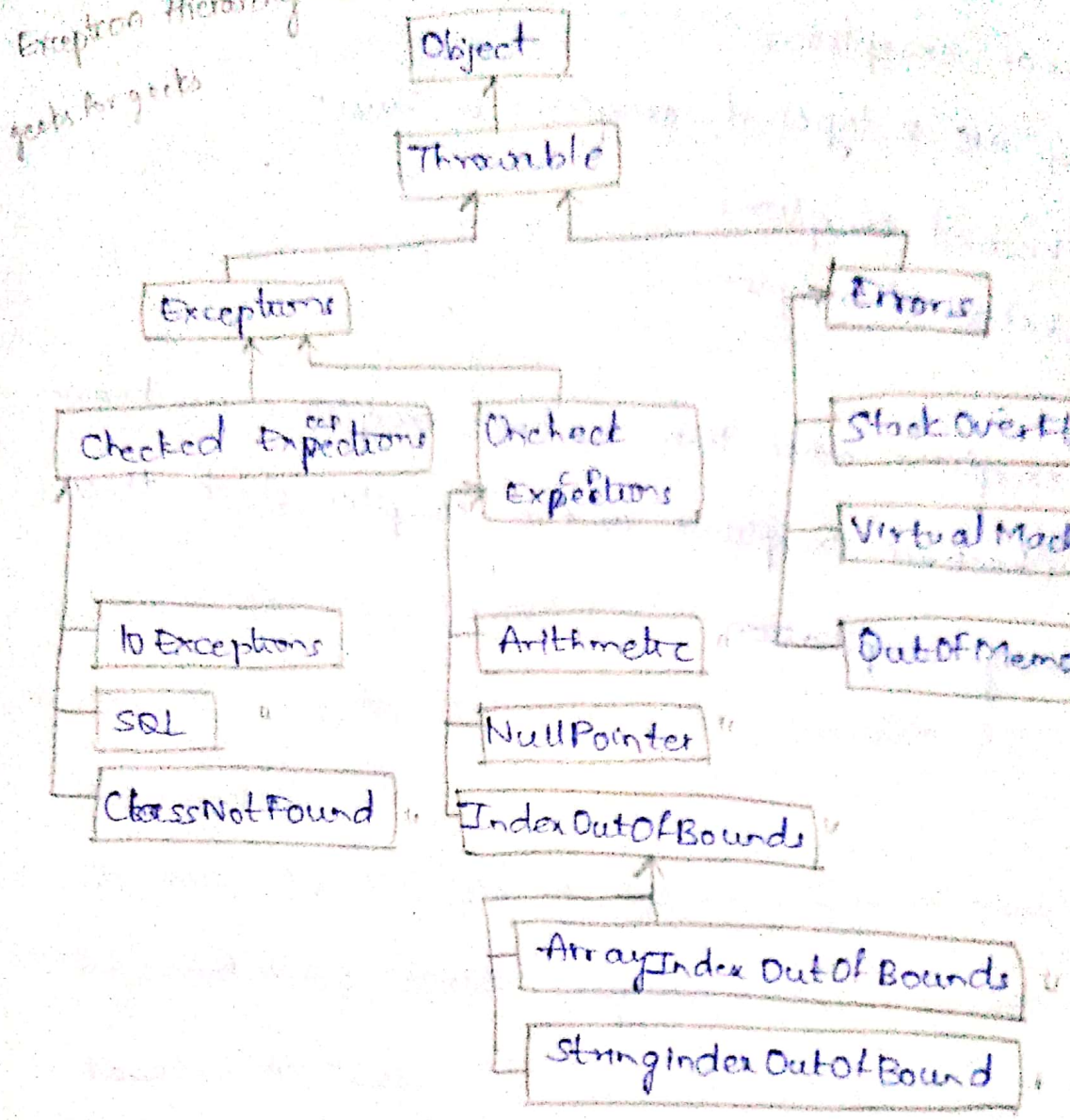
Types of exceptions:

There are 2 types of exceptions in Java:

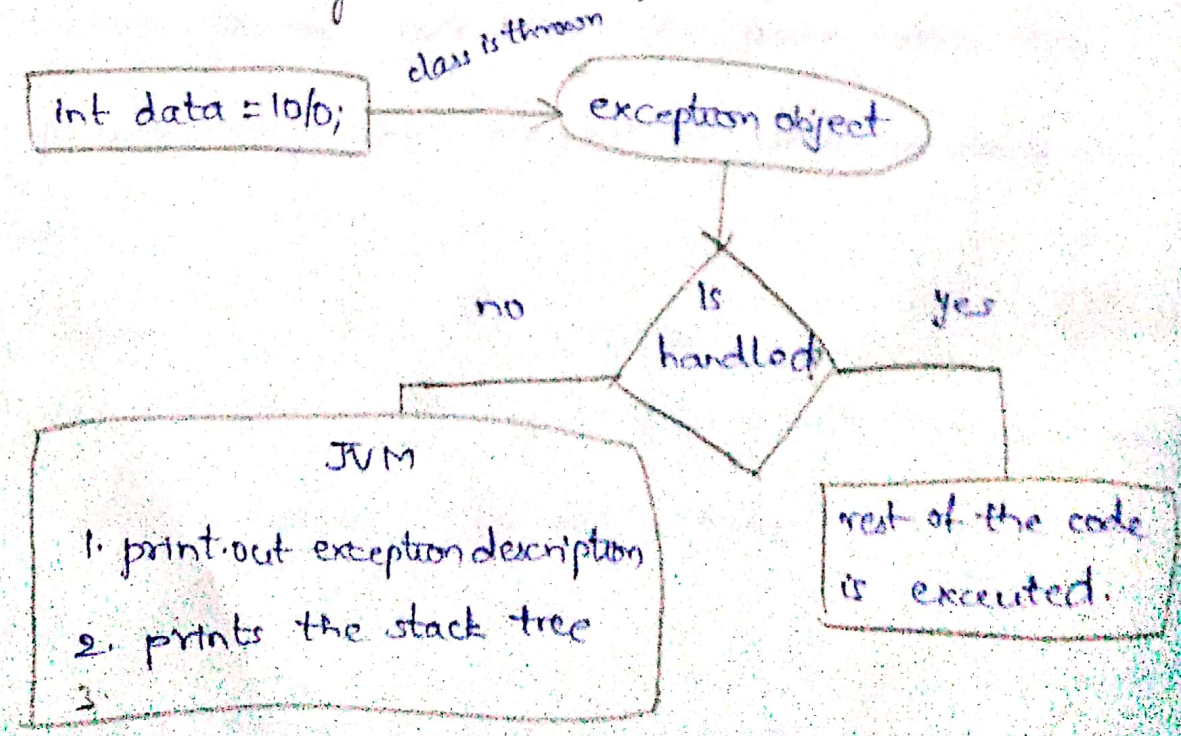
1. Checked exceptions
2. Unchecked exceptions

- All exceptions other than runtime exceptions are known as Checked exceptions as the compiler checks them during compilation

Exception Hierarchy



Internal working of java try-catch block



Java Exception Keywords

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.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Syntax for Java try-catch

```
try {
    //code that may throw an exception
}
catch (Exception_class_Name ref)
{

}
```

Syntax for try-finally block

```
try {
    //code that may throw an exception
}
Finally
{

}
```

Program without exception handling

```
public class TryCatchExample1
{
    public static void main (String[] args) {
        int data = 50/0;
    S.O.P("rest of the code");
    } }
```

o/p: Exception in thread "main" java.lang.ArithmeticException:/ by zero

Program with exception handling

```
public class TryCatchExample2 {
  public static void main (String[] args) {
    try
    {
    int data = 50/0;
    }
    catch (ArithmeticException e) {
      {
       s.o.p(e);
      }
      s.o.p ("rest of the code");
    }
  }
```

O/p: java.lang.ArithmeticException: / by zero
rest of the code

ALM-3 19/3/20

1. What is Exception in Java?
2. What are the Exception Handling keywords in Java?
3. Explain Java Exception Hierarchy?
4. What are the imp methods of Java Exception Class?
5. What is diff b/w checked and Unchecked Exception in Java?
6. How to write custom exception in Java?
7. What are different scenarios causing "Exception in thread main"?
8. What is difference b/w final, finally & finalize in Java?

10. Can we have an empty catch block?

11A-3

1. Write a java program to Demonstrate STACK using Interface Concept. You need to define an interface "StackInterface", abstract methods void push(int), int pop() and void dispStack(). Define a class MyStack that implements the interface and write a code to push(), pop() and dispStack() methods.

10/3/20

Program using exception handling

```
public class TryCatchExample6 {
    public static void main (String args[]){
    int i=50;
        j=0;
        data;
    try
    {
     data = i/j ;
    }
// handling exception
     catch (Exception e)
      {
       // resolving the exception in catch block
       System.out.println(i/(j+2));
      }
    }
}
```

// may throw exception

reference will be stored in ref. class

object of exception class

prints error msg along with 25

O/P : 25

Program using ArrayIndexOutOfBoundsException

```
public class TryCatchExample9{
  public static void main(String[] args){
    try
    {
      int arr[]= {13,5,7};
      System.out.println(arr[10]); //Array throws exception
    } //Handling the array exception
    catch(ArrayIndexOutOfBoundsException e))
      {
        S.O.P(e);
      }
        S.O.P("No data available in the specified location");
      }
}
```

O/P: 10

Multi-catch block

- A try block can be followed by one or more catch blocks. Each of catch block must contain a diff exception handler. So, if you have to perform diff tasks at the occurrence of diff exceptions, use java multi-catch block.

Note:

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general i.e, catch for ArithmeticException must come before catch for Exception.

Exception handling using multiple catch block

```
public class MultipleCatchBlock1 {
    public static void main(String args[])
    try {
        int a[] = new int[5];
        a[5] = 30/0;
        }
        catch (ArithmeticException e)
        {
            S.O.P("Arithmetic Exception occurs");
        }
        catch (ArrayIndexOutOfBoundsException e)
        {
            S.O.P ("ArrayIndexOutOfBounds Exception occurs");
        }
        catch (Exception e)
        {
            S.O.P("Parent Exception occurs");
        }
}
```

op: Arithmetic exception occurs

Throw Vs Throws In Java

THROW	THROWS
1. A throw is used to throw an exception explicitly.	1. A throws to declare one or more exceptions, separated by ','.
2. Can throw a single exception using throw.	2. Multiple can thrown using throws
3. This keyword is used in the method	3 signature method is used with keyword throws.
4. Only unchecked exceptions propagated using throw keyword.	4. To raise an exception throws keyword followed by the class name & checked exception can be propagated.
5. Throw keyword is followed by the instance variable.	5. Throws keyword is followed by the exception class.

Program using throw keyword:

```
public class ThrowExample{
  void Votingage (int age){
      if (age<18)
        throw new ArithmeticException("you can't vote as
          not Eligible to vote");
      else
        S.O.P("Eligible for voting");
  }
  Public static void main (String args[]){
    ThrowExample obj = new ThrowExample();
    obj.Votingage(13);
    S.O.P("End of Program");
```

```
    }
  }
```

o/p: End of program

Program using throws keyword:

```
public class ThrowsExample{
  int division(int a, int b) throws ArithmeticException {
    int intet = a/b;
    return intet;
  }
  public static void main (String args[]){
    ThrowsExample obj = new ThrowsExample();
    try{
      S.O.P(obj.division (15,0));
    }
    catch(ArithmeticException e){
      S.O.P("Division cannot be done using ZERO");
    }
  }
}
```

o/p: Division cannot be done using ZERO

Nested try blocks:

- Nested try blocks is try block written inside another try block.
- The nested try-catch block is used when a particular catch block is unable to handle an Exception, then

exception is re-thrown to outer catch block & this exception would be handled by the outer set of try catch block.

Example program using Nested try statement.

```
import java.util.InputMismatchException;
class Nested
 {
  public static void main(String[] args)
   {
    try
     {
      S.O.P("Outer try block starts");
      try
       {
        S.O.P("Inner try block starts");
        int res = 5/0;
       }
      catch(InputMismatchException e)
       {
        S.O.P("InputMismatchException caught");
       }
      finally
       {
        S.O.P("Inner final");
       }
     }
    catch(ArithmeticException e)
     {
      S.O.P("Arithmetic Exception caught");
     }
    finally
     {
      S.O.P("Outer finally");
     } } }
```

o/p: Outer try block starts

Inner try block starts

Arithmetic Exception starts

Outer finally

Multi-threaded programming

- Multithreading refers to two or more tasks executing concurrently within a single program.
- A thread is an independent path of execution within a program.
- Every thread in Java is created and controlled by the `java.lang.Thread` class.
- A Java program can have many threads & these threads can run concurrently, either asynchronously or synchronously.

What are Java Threads?

A thread is a:

- Facility to allow multiple activities within a single process
- Referred as lightweight process.
- A thread is a series of executed statements.
- Each thread has its own program counter, stack & local variables.
- A thred is a nested sequence of method calls.
- Its shares memory files

What is the need of a thread or why we use Threads?

To perform asynchronous or background processing.

- ↑ the responsiveness of GUI app^ns
- Take advantage of multiprocessor systems
- Simplify program logic

Thread creation in Java

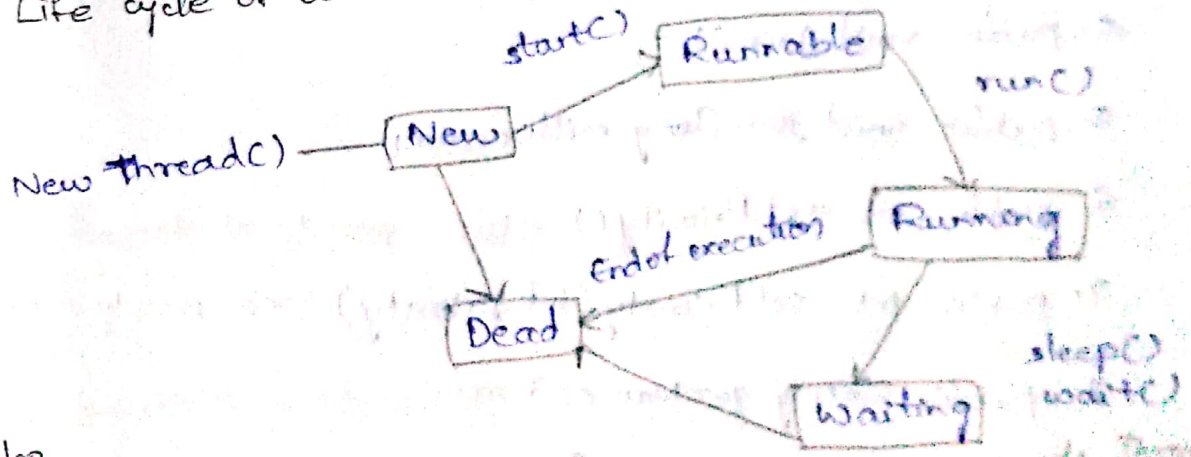
Thread implementation in Java can be achieved in 2 ways:

- Extending the java.lang.Thread class
- Implementing the java.lang.Runnable Interface

Note:

Thread & Runnable are available in the java.lang.* package

Life cycle of a thread



12/3/20

Commonly used Constructors of Thread class:

Thread()

Thread (String name)

Thread (Runnable r)

Thread (Runnable r, String name)

Basic structure of how Thread runs:

```
class Class_Name extends Thread
{
```

```
public void run()
{
}
Class_Name obj = new Class_Name();
Thread t = new Thread(obj);
t.start();
```

Commonly used methods of Thread class:

1. public void run() : used to perform action for a thread
2. public void start() starts the execution of thread JVM calls run()
3. public void sleep(long milliseconds)
4. public void join()
5. public void join(long milliseconds)
6. public int getPriority() returns priority of thread
7. public int setPriority(int priority) changes priority of thread
8. public String getName() returns name of thread
9. public void setName(String name) changes name of thread
10. public Thread currentThread() returns reference of currently executing thread
11. public int getId() returns id of the thread
12. public Thread.State getState() returns state of thread
13. public boolean isAlive() tests if thread is alive
14. public void yield() causes currently executing thread to yield
15. public void suspend()
16. public void resume()

17. public void stop()

18. public void interrupt()

19. public boolean isInterrupted()

20. public static boolean interrupted()

13/3/20

The procedure for creating threads based on Runnable interface is as follows:

1. A class implements the Runnable interface, providing the run() method that will be executed by thread. An object of this class is a Runnable object.
2. An object of Thread class is created by passing a Runnable object as argument to the Thread constructor. The Thread object now has a Runnable object that implements the run() method.
3. The start() method is invoked on Thread object

The procedure for creating threads based on extending enterface.

Program using thread class

```
class MultithreadingDemo extends
 Thread{
   public void run()
     {
       S.O.P("My thread is in running state");
     }
     public static void main (String args[])
     {
      MultithreadingDemo obj=new MultithreadingDemo();
      obj.start();
      }}
```

O/P: My thread is in running state

Program using Runnable Interface

```
class MultithreadingDemo
 implements Runnable
  {
   public void run()
    {
     S.O.P (" My thread is in running state");
     }
     public static void main(String args[])
     {
     MultithreadingDemo Obj= new MultithreadingDemo();
     Thread tobj =new Thread(obj);
        tobj.start();
        }}
```

Op: My thread is in running state

(1) Program using for multithreading using thread class

```
class MultithreadingExample extends Thread
 public void run()
    {
     for (int i=1; i<=3;i++)
     {
      S.O.P(Thread.currentThread().getName()+"
          is running: "+ i+" ");
       try{
          Thread.sleep(1000);
        }
       catch (InterruptedException e) {}
```

```
    }
  }
}

class Main
{
  public static void main (String args[])
  {
  MultithreadingExample obj1 = new MultithreadingExample();

  MultithreadingExample obj2 = new MultithreadingExample();

  obj1.setName("Thread-1");
  obj2.setName("Thread-2");
  obj1.start();
  obj2.start();
  }
}
```

O/p: Thread-1 is running: 1

2	1
2	2
1	2
1	3
2	3

(2)

```
class A extends Thread{
  public void run() {
  S.O.P ("Class A method...");
  for(int i=0; i<=10; i++){
```

```
        S.O.P(i);
        }}}

class B extends Thread{
    public void run(){
        S.O.P("Class B method...");
        for(int i=0;i<10;i++)
        {
        S.O.P(i);
        }}}

public class ThreadTutorial {
    public static void main(String args[]){
        A a=new A();
        B b=new B();
        a.setPriority(Thread.MAX_PRIORITY);
        b.setPriority(Thread.MIN_PRIORITY);
    }
}
```

Program using of multiple threading using runnable interface

```
class MyThread implements Runnable
  {
    String message;
    MyThread(String msg)
      {
        message=msg;
      }
    public void run()
```

```
{
for(int count=0; count<=5; count++)
{
  try
  {
    s.o.p("Run method:" + message);
    Thread.sleep(100);
  }
 catch (InterruptedException ie)
  {
    s.o.p(" Exception in thread : " + ie.getMessage());
  }}
  }}
 public class MainThread
 {
 public static void main (String args[])
 {
  MyThread obj1 = new MyThread ("MyThread obj1");
  MyThread obj2 = new MyThread ("MyThread obj2");
  Thread t1 = new Thread(obj1);
  Thread t2 = new Thread(obj2);
    t1.start();
    t2.start();
  }
 }
```

16/3/20

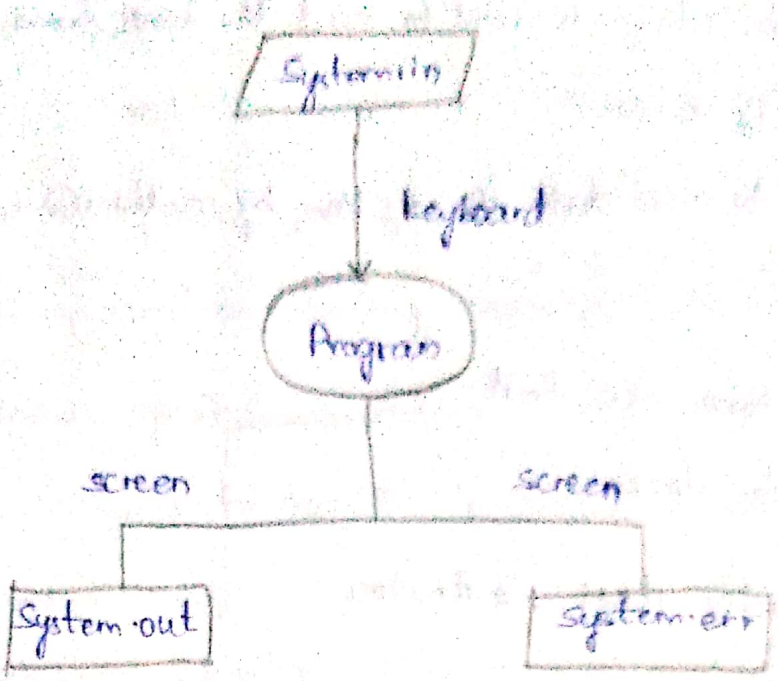
Java IO Stream

- Java performs I/O through Streams
- A stream is linked to a physical layer by java I/O system to make i/p and o/p operation in java.
- In general, a stream means continuous flow of data. Streams are clean way to deal with i/p/o/p without having every part of your code understand the physical.
- Java encapsulates Stream under java.io package. Java defines two types of streams. They are:
- Byte stream: It provides a convenient means for handling i/p and o/p of byte
- Character Stream: It provides a convenient means for handling i/p & o/p of characters.
- Character stream uses Unicode and therefore can be internationalized.

Java IO: Input - Output in Java

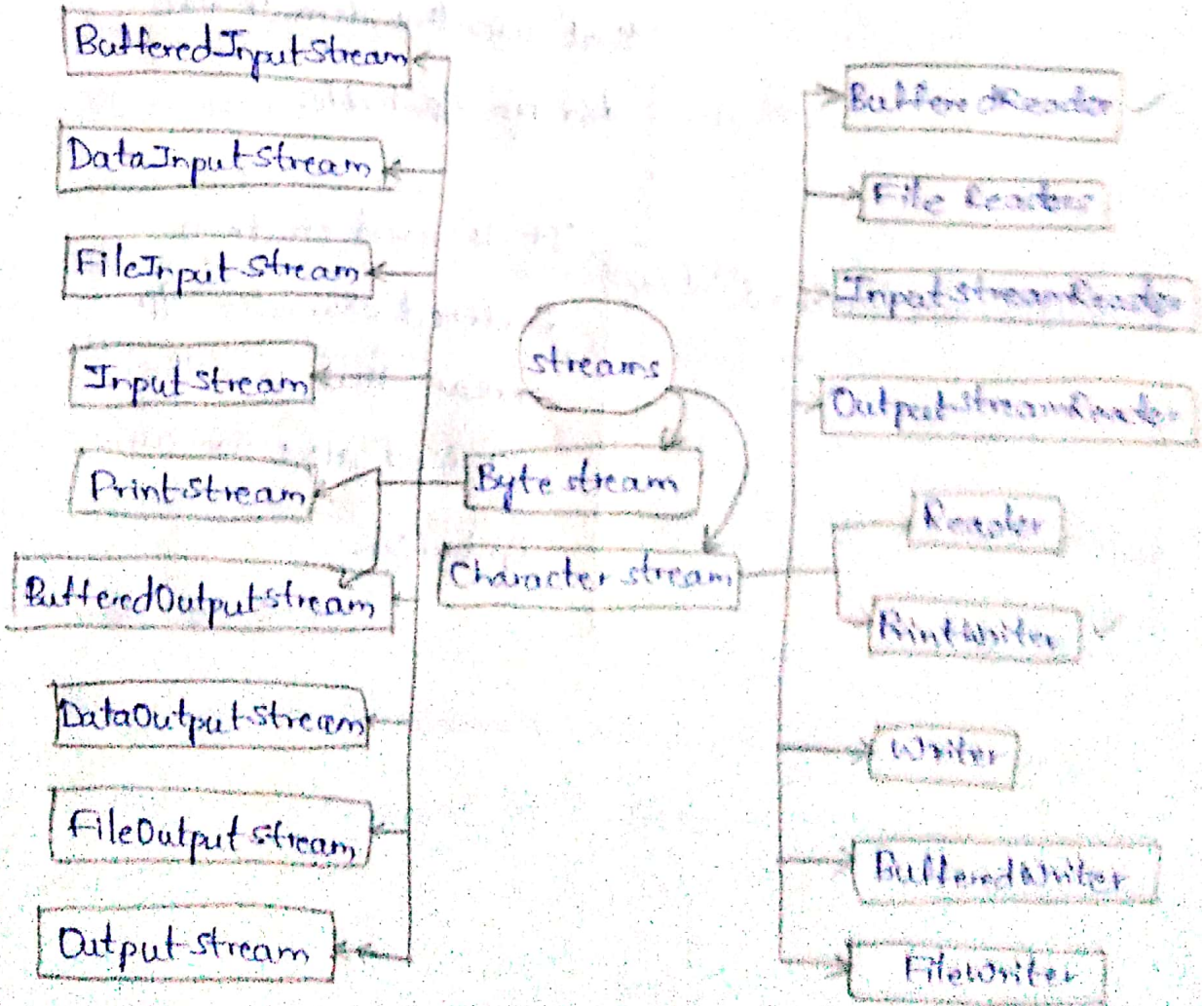
- Java brings various Streams with its I/O package that helps the user to perform all the i/p o/p operations.
- These streams support all the types of objects, data types, characters, files etc to fully execute

the I/O operations.



Classes, methods available in streams

Classification of byte stream and character stream



Java BufferedReader Class

- Java BufferedReader class is used to read the text from a character-based i/p stream.
- It can be used to read data line by line by readLine() method.
- It makes the performance fast.
- It inherits Reader class.

Java BufferedReader class constructors

Constructor	Description
BufferedReader(Reader rd)	It is used to create a buffered character i/p stream that uses the default size for an i/p buffer
BufferedReader(Reader rd, int size)	It is used create a buffered character i/p stream that uses the specified size for an i/p buffer.

Java BufferedReader class methods

int read()

It is used for reading a single character

int read(char[], int off, int len)

It is used for reading characters into a portion of an array.

boolean markSupported()

It is used to test the i/p stream support for the mark & reset method.

String readLine()

It is used for reading a line of text.

boolean ready()

It is used to test whether the i/p stream is ready to be read

long skip(long n)

It is used for skipping the characters.

void reset()

It is responsible to repositions the stream at a position the mark method was last called on this i/p stream.

void mark(int readAhead. in

It is used for marking the present position in a stream

void close()

It closes the i/p stream & releases any of the system resources associated with stream.

3/20

Program for BufferReader Class.

```
import java.io.*;
public class BufferedReaderExample {
    public static void main(String args[]) throws Exception{
        FileReader fr = new FileReader("D:\\testout.txt");
        BufferedReader br = new BufferedReader(fr);
        int i;
        while((i=br.read())!=-1){
            System.out.println((char)i);
        }
        br.close();
        fr.close();
    }
}
```

O/p: whatever the content in D:\testout.txt is printed

CO-4

Java Collections Framework

- Java Collections Framework is a collection of interfaces and classes which helps in storing and processing the data efficiently.
- This framework has several useful classes which have tons of useful APIs which makes a programmer task super easy.
- The collection in Java is a framework that provides an architecture to store & manipulate group of objects.
- Java Collections can achieve all the operation that you perform on a data such as searching, sorting, insertion, manipulation and deletion.
- Java Collection means a single unit of objects.
- Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

What is Collection in Java

- A collection represents a single unit of objects i.e; a group

Framework in Java

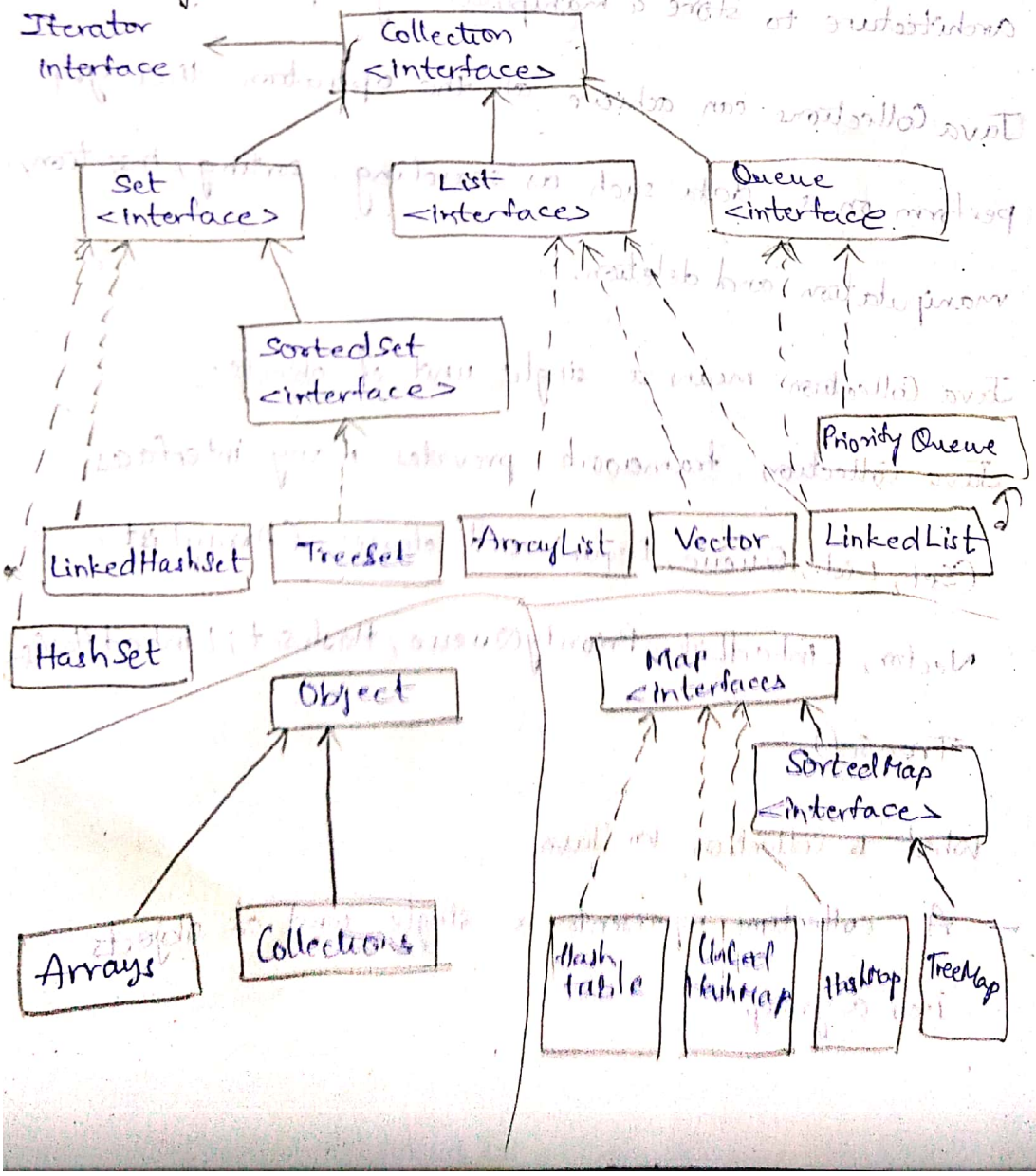
- It provides readymade architecture
- It represents a set of classes & interfaces.
- It is optional

Collection framework in Java

The Collection framework represents a unified architecture for storing & manipulating a group of objects. It has:

1. Interfaces & its implementations
2. Algorithm

Summary about Collections



Methods declared inside collection interface.

There are many methods declared in the Collection interface. They are as follows:

Methods:

1. public boolean add(E e)

   It is used to insert an element in this collection

2. public boolean addAll(Collection<? extends E> c)

   It is used to insert the specified collection elements in the invoking collection

3. public boolean remove(Object element)

   Used to declare an element from the collection.

4. public boolean removeAll(Collection<?> c)

   Used to delete all the elements of specified collection from invoking collection.

5. default boolean remove

   Used to delete all the elements of the collection that satisfy the specified predicate.

Used to delete all the elements of invoking collection except the specified collection.

Returns the total no of elements in collection

Removes total no of elements from collection

Used to search an element

It is used to search the specified collection in the collection

Returns an iterator

Converts collection into array

Converts collection into array. Here, the runtime type of returned array is that of specified array.

Checks if collection is empty

Returns a possibly ||el stream with collection as its source.

Returns a sequential stream with collection as its source.

It generates a Spliterator over the specified elements in the collection.

It matches two collections

Returns the hash code no. of collection.

Iterator Interface

Iterator interface provides the facility of iterating the elements in a forward direction only.

Methods of Iterator interface

There are only 3 methods in the Iterator interface.

They are:

1. public boolean hasNext()

   Returns true if the iterator has more elements otherwise it returns false.

2. public Object next()

   Returns the element and moves the cursor pointer to the next element.

3. public void remove()

   Removes the last elements returned by the iterator.

   It is less used.

Iterable Interface:

The Iterable interface is the root interface for all the collection classes.

The Collection interface extends the Iterable interface and therefore all the subclasses of Collection.

It contains only one abstract method.

1. Iterator<T> iterator()

Ex program using ArrayList in collections

Method Collection interface (Using ArrayList)

```
import java.util.*;

class TestJavaCollection1 {

  public static void main(String args[]){

ArrayList<String>list=new ArrayList<string>();
                                  //creating arraylist
 list.add("Sai"); //Adding object in arraylist
  list.add("Sree");
   list.add("Sai");
   list.add("Ancha"); //Traversing list through Iterator
    Iterator itr=list.iterator();
    while(itr.hasNext()){
     s.o.p(itr.next());
       }
    }
  }
```

o/p: Sai
Sree
Sai
Ancha