Core Java (7 weeks)

Head first Java

- Notepad++, SublimeText, JCreator
- Eclipse,Netbeans (IDE-Integrated development environment)

- Basics of Java programming
- JVM,JRE,JDK
- OOP
- Arrays
- String
- Inheritance
- Exception Handling
- Multithreading
- Collection

Windows x86 (32-bit)
Windows x64 (64-bit)

Program Files => 64 bit software
Program Files (x86) => 32 bit software

javac - compilation
java - Run(execute)

Path variable :

oop
inheritance
exception handling
multithreading
collection

(primitive/in-built data types)
byte, short, int, long
float, double
char,
boolean

```
class Person {
    String name;
    Date birthDate;
    char gender;
```

```
    String address;
}

// Inheritance
class Student extends Person {
    int rollNo;
    String collegeName;
    String courseName;
}
int i = 10;
Student s1 = new
Student(1,”Fc”,”computer”);
Student s2 = new Student();
s2.rollNo = 2;
s2.collegeName = “Wadia”;
s2.courseName = “Arts”;
 class Institute {
        static String instituteName;
        static String address;
 }
```

Java is platform independent:

- Write once run anywhere(WORA)
- Program compiled on one machine can run on any machine
- There are 2 things which makes java platform independent:
    1. Bytecode (Platform independent)
    2. JVM (Platform dependent)
- In java size of each data type is fixed (e.g. “int” takes 4 bytes)

Program execution sequence:

Source code (.java) => javac(compiler) => ByteCode(.class) => JVM => m/c code

JIT - Just-in time compilation (It uses optimization and caching techniques)

Java is both “compiled and interpreted” language.

Java is pure object oriented language:

- Everything in java is about objects

- Even data types can be represented as objects using “Wrapper classes”
- In Java you can not write anything outside class.(In C++ we can write code outside class using :: scope resolution operator)

Garbage collection:
- Java has in-built mechanism of memory cleanup and it is called as Garbage Collection(GC)
- GC is one of the component of JVM. It runs periodically and checks for objects which are not in use from Heap memory.

Java is more secure:
- There are no pointers in Java, so no memory manipulations are allowed.

Data type size:
byte - 1 byte (-128 to 127) n=8 - 2^(n-1) to 2^(n-1) - 1

short - 2 bytes
int - 4 bytes
long - 8 bytes

float - 4 bytes
double - 8 bytes

char - 2 bytes (Java supports UTF and multilingual characters)
UTF- unicode text format (Java characters are unicode characters)
boolean(true-1/false-0) -> 1 bit

Wrapper class:
For each data type Java provides a class and it is called as wrapper class.
e.g. Integer for “int”, Float for “float” and so on.

AutoBoxing: Converting data type to wrapper object
e.g.

Integer i = 8; (int to Integer)

Unboxing: converting wrapper type to data type.

```
Integer obj = new Integer(8);
int i = obj;//wrapper type to data type
```

File Naming Rules:

- File name must be same as class name, only if that class is public.
- If we have class with “default” access specifier, then we can give any file name.
- In one file we can declare only one public class. We cannot have multiple public classes in one file.
- We use “fileName” during compilation and “class name” during execution.

Scenario 1. FileName and class name are different.

FileName:Demo.java

```
class A{
```

```
  public static void main(String[] args) {
    System.out.println(“Demo”);
  }
}
//Compile: javac Demo.java (no error)
//Run       : java A (output-Demo)
```

Scenario 2: FileName must be same in following case.

file name: Test.java

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

//Compile: javac Test.java (Compilation error- file name must be same as class name)

What is class?

- Class is encapsulation of related data members(properties and functions)
- Class consist of properties(states) and functions(behaviors)

- It’s a template using which we can create objects.
- Any real world entity or logical entity who has set of properties and functions can be represented as a class.

ex. of class:

```
class Student {
    //properties
   private int rollNo;
    private String name;
    private float marks;
    static int totalCount;
    //functions
    public int getRollNo( ) {  return this.rollNo; } // Getter method
    public void setRollNo(int r) { this.rollNo = r; }// Setter method

   public float calculatePercentage( ) {  return %; }
    static increaseCount() {
```

```
        Student.totalCount++;
    }
}
```

what is Object?

- Object is an instance of a class.
- How to create object in java?
  Ans: Using “new” operator.
  Ex.
  Student s = new Student( );
- All objects goes in Heap memory.
- Block of memory is allocated for object based on properties of class.
  e.g. Student s = new Student(11, “John”, 80.5f);

4bytes	8bytes	4bytes
int	4 char	float

Instance and static members:

- A class can consist of both instance and static members.

1) instance members:
    a) instance property
        - It is a property of an object.
        - Declared inside class outside method (without “static” keyword)
        - It is also called as “non-static” property.
        - We need an object to access this property.
    b) instance function
        - Function which refers instance property.
        - We must have an object to call instance function.
        - “this” reference variable is always available in instance function and it gives reference of current object.

2) static members:
    a) static property:
            - Memory is allocated only once.

- It should be accessed by class name

- We can also access static property using object(but we should avoid this)

- static property is property of a class.

- No matter how many objects we create, static property would be created only once in memory and all objects can share that.

b) static function:

- A function which uses static property

- We don’t need object to call static function. Static functions are called directly using class name.

- “this” reference variable is NOT available in static function.

<u>Reference v/s Object:</u>

- Reference is a variable which goes on stack and actual object gets created on heap memory.
- Reference points to an object (Reference holds memory address of an object)
- Reference is used to access an object.
Dig.

```
class Emp {
   int empId;
    String name;
    float salary;

    static int empCount;
    static String companyName;

    float calculateTax( ) {
        Sop(this.empId);//11
        return this.salary * 0.1;
   }
```

```
    static void increaseEmpCount() {
        Emp.empCount++;
    }
}
```

```
Emp e1 = new Emp(11,”Fred”,10000);
e1.calculateTax( );

Emp.increaseEmpCount( );
```

Constructor

- A special function whose name is same as class name.
- It gets executed automatically when object of class is created.
- It is used to initialize properties of an instance(object)
- Every class in Java has a constructor.
- If we don’t write a constructor then JVM provides a default constructor.
- There are 2 types of constructors in

Java.

1. Default constructor (No-arg constructor)
    - constructor without any argument
2. Parameterized constructor
    - constructor with arguments

- We can write multiple constructors for one class. That means “constructor overloading” is possible.
- If we write parameterized constructor, then JVM doesn’t provide any default constructor. That means JVM provides default constructor ONLY when if we don’t have any constructor.
- Constructor does not have return type.
- Access specifier of constructor should match with access specifier of class. So for public class we will add public constructor and for default class we will add constructor with default access specifier.

Ex 1. Class with default constructor(no-arg)

```
    public class Emp {
        public Emp( )
{ System.out.println("default constructor"); }
    }
```

Ex 2. Class with parameterized constructor

```
    class Emp {
      private int id;
        private String name;
        private float salary;
        Emp(int id, String name, float salary) {
            //initialization logic
            this.id = id;
            this.name = name;
            this.salary = salary;
        }
  }
```

- In Java we don't have copy constructor. But we can use "Object cloning" for creating new copy of an object with same data.

“toString( )” method:

- This is a method from “Object” class.
- This method is used to “convert object into string representation”
- Any class can override this method.
- Syntax of this method is:

```
public String toString( )  { }
```

- This method is written in class where you need string representation for an object.
- By default “toString( )” method from “Object” class returns “hashcode”(which is calculated from memory address).
- This method gets called automatically when string representation of an object is needed.

Example of using “toString( )” method.

“this” keyword:

- “this” keyword can be used only inside instance function.

- Usage:

1. “this” as a reference:

Inside every instance function reference “this” is available which points to current object on which that instance function is invoked.

Using “this” reference we can differentiate between instance and local properties.

2. “this” to call self constructor:

Using this( ) or this(<parameters>) we can call constructor of our own class.

So it is used to call constructor explicitly. There are few conditions for using it. a) You must be inside a constructor b) Explicit constructor call can be given only from first line of constructor.

3. Instance functions can also be called using “this” keyword.

e.g.

```
class A {
    public void f1( ) {
            …
            this.f2( );//Calling f2( ) on same object
            …
    }
    public void f2( ) { //some code }
    main() {
     A obj = new A( );
     obj.f1();
    }
}
```

Data type conversation:

```
int i = 8;
short s = i; //int to short ? Not directly allowed. Need explicit type casting

short s = 8;
int i = s;//short to int ? Possible
```

Bigger type can not be converted into smaller type.
But smaller type can be converted into bigger type.

Sequence:
byte->short->int->long->float->double
Left side data type in above sequence can be easily assigned to right side type.
Reverse (right side to left side) in above sequence needs explicit type casting.

No “unsigned” keyword in java (Because there is no pointers in Java)

main ( ) function in Java:
- This is entry point function in Java. Any java program execution starts from “main()” function.
- Who calls main( ) ? => JVM (using “java” command JVM is invoked)
- Syntax of main( ):

public static void main(String[ ] args) { } OR

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

public static void main(String… args) { } //Using var-arg (…)

- Why main() is public?

It is called from outside the program by JVM and if we don't keep it as public then JVM would not able to call it.

- Why main() is static?

Static functions can be called directly using class name. If main() is instance then JVM would need to create an object.

- What is "String[] args" => Command line Arguments

When we run java program we have an option of passing data from command line and data passed from command line internally gets converted into String array.

e.g. java StudentTest 11 Fred 78.5f

Here “StudentTest” is class name and “11 Fred 78” are command line arguments. So array of size 3 of type String [“11”, “Fred”, “78”] is created and passed to “main()” function.

- Return type “void”? -> main( ) doesn’t return anything and hence return type is “void”

Command Line Arguments:

- Syntax of running java program is as below.

“java <class-name> <para1> <para2> ….

e.g.

java StudentTest 11 Fred 89.4f

Here command line arg is: [“11”,”Fred”,”89.4f”]

- It is used to pass some data to program when execution starts.
- It is generally used to pass configuration parameters. (e.g. database IP, port number,

IP address etc)
- User level data can be passed. But other mechanisms like “BufferedReader/Scanner” are preferred for reading user input.
- All data passed with command gets converted into String array.

We need explicit type conversion inside main( ) function.

How to read data from user ?

a) Using BufferedReader
b) Using Scanner

Menu driven program for arithmetic operations using Scanner:

1. Add
2. Subtraction
3. Multiplication
4. Division
5. Exit

Enter choice ? 3

Enter two numbers : 5 8
5 * 8 = 40

1. Add
2. Subtraction
3. Multiplication
4. Division
5. Exit

Enter choice ?

Arrays:

- Array is collection of similar types of elements.
- Only specific type of data can be stored in array(Based on type of an array)
- Once array is created we cannot increase/decrease it's size.
- Array index always starts from 0.
- In Java array length can be accessed using "length" property.
- There is a utility class "java.util.Arrays" which helps for various array operations.

- We can create both 1-D and 2-D arrays in Java.

Syntax of creating 1-D array:

```
int[ ] arr = new int[5];//array of 5 integers
 int[ ] arr = {1,3,5,2};//Array with data
 String arr[ ] = new String[10];//Array symbol can be after type or variable
 String[ ] cities = {“Pune”,”nasik”,”jalgaon”, “goa”};

 int a[10];//INVALID -Error(Size must be specified on right side)
```

How to traverse 1-D array:

```
 e.g. int[ ] a = {2,4,6,8,10};
 Using normal for loop:
 for (int i=0; i < a.length; i++ ) {
  //System.out.println(a[i]);
  a[i] = a[i] + 5;
}
```

Using for-each loop:(Traversing without using index)

```
for(int element: a) {
    System.out.println(element);
}
```

Traverse String[ ] using for-each loop:

```
String[ ] cities = {“Pune”,”Nagpur”,”Nasik”,”Goa”};
for(String c : cities) {
    sop(c);
}
```

Traversing array in reverse order:(e.g. array name is “arr”)

```
for(int i=arr.length-1 ; i >=0 ; i—) {
    sop(arr[i]);
}
```

** Arrays are always created on Heap memory.**

** We can pass array to any function and

we can also return array from function**

Trace the output:

```
class A {
  public static void main(String[ ] args) {
        int arr[ ] = {1,2,3,4,5};
        fun(arr);
        for(int e: arr) { sop(e); }
  }
  public static void fun(int[ ] a) {
    for(int i=0; i<a.length; i++ ) {
        a[i] = a[i] * a[i];
   }
  }
}
```

https://www.geeksforgeeks.org/

Array of user-defined type :

```
class Student {
    int rollNo; String name; float marks;
}
```

Syntax:

Student[ ] arr = new Student[10]; //Array of 10 student references

Above syntax just creates an array.
Actual student object can be created as below.

arr[0] = new Student(11,"Fred",80);

Var-Arg List (…)

- Used when we have to pass same type of data and it varies in number.
- How to write add( ) function which can work for any number of elements.

Solution: Var-arg list

```
static int add(int ...a) {
    int sum = 0;
    for(int i=0;i<a.length;i++) {
        sum = sum + a[i];
    }
    return sum;
```

}

- Var-arg list must be the last parameter in arguments.

fun(int i, int… a) //Valid

fun(8,9,10)

fun(int…a, int i) //invalid

- We can not have multiple var-arg parameters in one function

fun(int… a, int …b) //Invalid

fun(String …a, int …b) //invalid

- WE can pass 0 to n number of arguments to var-arg list.
- If function with specific number of argument matches then preference is given to more specific function over var-arg.

2-D array

Pass By mechanism

STRING

2-D array syntax:(Array of arrays)

- int[ ][ ] a = new int[3][4];//3 rows, 4 columns
- int[ ][ ] a = {{1,2,3},{4,5,6},{7,8,9}}

Dig.

```
//Jagged/Ragged array(Array with variable number of columns)
int [ ] [ ] arr = {{1,2}, {3,4,5}, {6,7,8,9}};
    for(int i=0; i < arr.length; i++) {
        for(int j=0; j<arr[i].length; j++) {
        }
    }
```

Row size is mandatory and column is optional.

```
int [ ] [ ] a = new int [5][ ];//skipping column size : Valid
int [ ] [ ] a = new int [][2 ];//skipping row size : Invalid
```

Pass By Mechanism in Java:
Java uses Pass By Value mechanism for

parameter passing.

int i = 10;//10 is integer literal
String d =“abc”;//abc is string literal

String:
- It is used when we have to store group of characters.
- “String” is in-built class from Java. (java.lang.String)
- It is immutable (We can not change string content directly. Changes are possible but all changes happens in new memory)
- There are two way of creating string in Java.
    1. Direct assignent
       String s = “java”;//
    2. Using “new” operator
       String s = new String(“java”);
- When we assign string literal directly, string gets stored in special area called as

“String Literal Pool” and

When we create string using “new” operator, string gets memory in Heap area.

String storage mechanism

a. String Literal pool (String created by assigning literal directly)

b. Heap (String created using “new”)

Ex.

String s1 = “Java”;//new entry in pool

String s2 = new String(“Java”);//new memory in Heap

String s3 = “Java”;//No new entry, existing entry is re-used

String s4 = new String(“Java”);//new memory in Heap

Dig.

<u>String functions:</u>

1. How to get length of string?

int length( )

e.x. String s = “java”;

```
        int len = s.length( );//4
```

2. How to get character from string?

    char charAt(int index)

ex. String s = “Today”;

```
        char ch = s.charAt(2);//‘d’
```

This function is useful in string traversing.

e.g.

```
        for (int i=0; i<s.length( );i++) {
            char ch = s.charAt(i);
            sop(ch);
        }
```

3. Convert string into character array:

    char[ ] toCharArray( )

ex. String s = “Java”;

```
        char[ ] arr = s.toCharArray( );//
```

[‘J’,’a’,’v’,’a’]

4. Convert char[ ] to String

    Use String constructor as below

ex. char[ ] arr = {‘a’,’b’,’c’};

```
        String s = new String(arr);//“abc”
```

5. Find position of character/sub-string in string:

int indexOf(char ch)

int indexOf(String substr)

int lastIndexOf(char ch)//Returns last occurance

ex. String s = “String is immutable”;

int i = s.indexOf(‘i’);//return position of first ‘i’ => 3(count from 0)

int j = s.lastIndexOf(‘i’);//checks for last occurrence of ‘i’ => 10

int k = s.indexOf(“ring”);//2

int m = s.indexOf(“try”);//-1

int n = s.indexOf(‘x’);//-1

6. Check if string starts/ends with particular pattern

boolean startsWith(String pattern)

boolean endsWith(String pattern)

ex. String s = “String is immutable”;

boolean b1 = s.startsWith(“ring”);//false

boolean b2 = s.endsWith(“table”);//true

7.
boolean contains(String str) - check if string contains substring
e.g. String s = “Java is oop language”;
boolean b1 = s.contains(“is”);//true
boolean b2 = s.contains(“prime”);//false

8. substring( )
syntax:
String substring(int start)//return substring from index ‘start’
String substring(int start , int end)//returns substring between start and end.
ex.
String s = “Macbook Pro”;
String s1 = s.substring(3);//“book Pro”
String s2 = s.substring(3, 7);//7-3=4 characters from index 3 => “book”

9. String comparison:
Never use “==“ for string comparison use one of the following functions.

equals(String s2)//case sensitive

equalsIgnoreCase(String s2)//case insensitive

ex.

```
String s = “Java”;
boolean b1 = s.equals(“java”);//false
boolean b2 = s.equalsIgnoreCase(“java”);//true
boolean b3 = s.equals(“Java”);//true
```

Note: “equals( )” is function of “Object” class which is overridden in String class. and “equalsIngoreCase( ) is function of “String” class itself.

10.Case conversion:

String toUpperCase( )

String toLowerCase( )

ex.

```
String s = “MacBook”;
```

String lower = s.toLowerCase( );//“macbook”
String upper = s.toUpperCase();// MACBOOK

11. Replace
syntax:
String replace(char old, char new)
String replace(String s1, String s2)// replaces all occurances
String replaceFirst(String s1, String s2)// replaces only 1st occurrence
ex.
String s = “windows OS and windows phone”;
String s1 = s.replace(‘w’,’*’);//*indo*s OS and *indo*s phone
String s2 = s.replace(“windows”, “linux”);//linux OS and linux phone
String s3 = s.replaceFirst(“windows”,”mac”);//mac OS and windows phone

12. Splitting string into multiple strings(array of strings)

syntax:

String[ ] split(String pattern)

e.x.

String s = “white is a color of peace”;
String[ ] words = s.split(“ “);//split using space(“ “)
//output =>
[“white”,”is”,”a”,”color”,”of”,”peace”]

String s = “String is immutable”;
String[ ] arr = s.split(“t”);//[“S”,”ring is immu”,”able”]

13. trim( ) - Remove extra whitespace from both ends.

ex.

String s = “ Java “;
s = s.trim( );//“java”

14. concat

String s1 = “java”;
String s2 = “cpp”;
String s3 = s1 + s2;//“javacpp”
String s4 = s1.concat(s2);//“javacpp”

StringBuffer
StringBuidler

“equals( )” method:
- This is a method from “Object” class.
- Override this method when we need object comparison based on data/content.
- equals( ) method of Object class uses “==“ for comparison. (== operator does not check data it checks memory address)
- Syntax: public boolean equals(Object obj) { }

Example of overriding equals method:

```
class Emp {
```

```
    int id;
    String name;
    float salary;
    Emp() {}
    Emp(int id, String name, float salary) {
     this.id = id; this.name = name;
this.salary = salary;
    }
    //How to override equals() method of
Object class
    public boolean equals(Object obj) {
       // check if both are pointing to same
location
        if(this == obj) { return true; }
        // confirm that second object is of
correct type
        if(obj instanceof Emp) {
            Emp e1 = this;
            Emp e2 = (Emp)obj;
            return e1.id==e2.id &&
e1.name.equals(e2.name) &&
                    e1.salary == e2.salary;
```

```
        }
        return false;
    }
}
public class EmpEquals {
    public static void main(String[] args) {
        Emp e1 = new Emp(11,"Fred",
80000);
        Emp e3 = e1;
        Emp e2 = new Emp(11,"Fred",
80000);
        System.out.println(e1==e2);//false
        System.out.println(e1.equals(e2));
    }
}
```

clone( ) method:

- Method of “Object” class
- This is needed when we want to create new copy of an object with same data.
- Java allows cloning of an object through clone( ) method of Object class.

- Syntax of clone( ) from Object class:

```
protected native Object clone() throws CloneNotSupportedException;
```

- It is native method whose code is written in C/C++(Because it required memory manipulations and Java can not do that)

Steps of cloning:

1. A class must inform JVM about cloning by implementing “Cloneable” interfaces.

```
e.g. class Emp implements Cloneable { }
```

2. A class must override clone( ) method as below:

```
protected Object clone() throws CloneNotSupportedException {
    return super.clone();
}
```

3. From place where we call clone( ) method:

- We must have try-catch block with “CloneNotSupportedException”

- We must typecast object into appropriate type as below

```
try {
    e2 = (Emp)e1.clone();//Object
}catch(CloneNotSupportedException e) { }
```

Note: clone( ) method uses “Shallow copy”.

Object class methods:

- toString( ) : Object to String conversion
- equals ( ) : Checking equality of objects based on data
- clone( ) : Used for cloning objects(creating new copy with same data)
- hashcode( ) : Used in hash based collection
- finalize( ) : Gets called when object is about to destroy. We can write clean up logic. (Works like a destructor, but not guaranteed to be called always)

- wait(),notify(),notifyAll() - Method used in Mulit-Threading

Inheritance:
- Types of inheritance
- Polymorphism (Compile time v/s Runtime polymorphism)
Method overloading v/s Method overriding)
- super keyword
- Abstract class
- interface

Inheritance:
- OOP concept
- Inheritance promotes “Reusability”
- Inheritance occurs between two types(class, interface)
- “extends” and/or “implements” keywords are used for inheritance.
- It’s an IS-A relationship between child and parent class.

- A child class can re-use properties and functionality from parent class.
(Note A child can not access private property/function of parent class)
- Whenever we create object of child class, first object of parent is created.
- In Java, “Multiple inheritance” is not possible with classes but it is possible between class and interfaces.

Types of inheritance:
1. Single Level
2. Multi-level
3. Hierarchical

1. Single Level

```
class A { }
class B extends A { }
```

Dig.

2. Multi-level

```
class A { }
class B extends A { }
class C extends B { }
```

Dig

3. Hierarchical

```
class A { }
class B extends A { }
class C extends A { }
class D extends B { }
class E extends C { }
```

"Object" class:

- A top level class.
- It is also called as "Cosmic Super class".
- Any class in Java can inherit methods of "Object" class.
- If we don't specify parent for a class then default parent is "Object" class.

e.g.

```
class  A { }
```

In this case parent of "A" is "Object".

dig.

Constructor chaining:

- Whenever we create object of child class, first constructor of parent classes is called and this is called as constructor chaining.

First line of constructor is reserved for super( ) / this( ) constructor calls.

You can call your own constructor using this( ) / this(<para>) OR

You can call parent class constructor using super( ) / super(<para>

A default first line of constructor is “super( )”

- A call to default constructor of parent class.

static block and instance block

- A block written inside class, outside methods.
- Syntax:

```
static {
```

```
        //some code
    }
```

- A static block gets called only once during class-loading
- If class is not loaded then static block will not be called.
- Generally initialization code/code to be executed only once is written inside static block.
- One class can have any number of static blocks. Order of execution depends on sequence in which there are written.
- Any code which we write in static method can be written inside static block.
- Class loading:
    - JVM uses lazy loading mechanism that means JVM loads a class in memory only when it is needed.
    - One class gets loaded only once.
    - Few scenarios for which class loading will happen
        a. class with “main( )”

function(JVM load main class for starting execution)

b. A constructor call

c. “Class.forName(<class-name>)” - This is programmatic way of asking JVM to load specified class explicitly.

- Using static block we can also change entry point function. Instead of main() we can call some other static method. But main( ) will be called after static block.

Example1 :

```
class A {
        static { System.out.println(“static”); }
        public static void main(String[ ] args) {
                System.out.println(“main”);
    }
}
```

java A => output is :

static

main

“instance” block:

- A block written inside class outside method
- Syntax:

```
class A {
        …
        {
                //instance block
        }
    …
}
```

- It gets called whenever object is created
- If both “constructor & instance block” are present in class then first instance block and then constructor is invoked.
- Whatever code we write in instance function can be written in this block.
- This block is useful for performing some common task which is needed in all constructors.

- Instance block & constructor are always executed in pair.

- In case of inheritance, first “instance block + constructor” of parent is called and then “instance block+constructor” of child class will be called.

- Instance block can not be parametrized, whereas constructor can have parameters.

“super” keyword

- “super” is always related with immediate parent.

- Following are usage of “super” keyword:

a) Calling parent constructor from child constructor.

e.g. super( ); // call to default constructor of parent

super(1,3);//call to parameterized constructor of parent

b) Accessing parent class property. Very useful if you have same property name in parent and child.

super.<property_name>;

e.g.

```
class A {
    String abc = “A”;
}
class B extends A {
    String name = “B”;
    void fun( ) {

sop(this.name);//B

sop(super.name);//A
    }
}
```

c) Invoking parent class function from child class.

super.fun( );

Whenever we add parameterized constructor make sure we add default constructor as well.

Otherwise following problem can come.

```
class OS {
    String name = "OS";
    OS(String name) { this.name = name;}
}

class Linux extends OS { }
//error in Linux class- Call to super( ) fails.
//JVM adds default constructor in Linux class as below:
        Linux( ) {
            super( );
        }
```

- WE can not use “super” inside static function.

“final” keyword:

- “final” keyword can be used with variable, method and class.
- final variable
  - A constant variable whose value can not be changed.

    e.g.

    final int i = 19;

    i = 30;//error (“i” is constant)
  - initialization is mandatory for final variables

- final method :

  A method which can not be overridden

  e.g.

```
class A {
  final void fun( ) { }
}
class B extends A {
  void fun( ) { }//ERROR:
```

fun( ) is final method
}
- final class
A class which can not be inherited.
e.g. final class A { }
class B extends A { } //
Error: A is final class
Many classes in Java are final -
e.g. String, All wrapper classes (Integer,
Float, Character etc)

Polymorphism:
- One name many forms
- One of the piller of OOP concept.
- There are 2 types of polymorphism.
1. Compile-time/Static
polymorphism (static binding)
2. Run-time/dynamic
polymorphism (dynamic binding)

1. Compile time polymorphism:
- Decision of which method to be

called is taken at compile time.

- Method overloading is an example of compile time polymorphism.

2. Run time polymorphism:

- A decision of which method to be called is take at run-time.
- "Method Overriding" is an example of run-time polymorphism.

Method overloading:

- Same function name in one class with different in parameters.
- Parameters can differ in a) type b) count c) sequence
- Method overloading does not depend on return type.

e.x.

```
class Addition {
    int add(int a, int b) { return a+b; }
    String add(String a, String)
```

```
{ return a.concat(b); } // Differs in type
                        String add(int a, String b)
{ return b + a; }//Differs in type
                        String add(String a, int b)
{ return b+a; }//Differs in sequence
                        int add(int a, int b, int c)
{ return a+b+c;}//Differs in count
                    }
```

- Method overloading is compile time polymorphism.

Method overriding:

- It is run-time polymorphism
- When child class write method with same signature as of parent class method then it is called as method overriding.
- Decision of which method to be called (parent/child) is taken at runtime.
- Following methods can NOT be overridden.

1. static method 2. final method 3. private method

- While overriding scope/access specifier of method can be changed -

Rule is: A child class can increase the scope but can not decrease it.

- If parent class method has “throws” in signature then while overriding in child class there are 3 choices

a) Keep signature as it is with ‘throws’

b) Remove ‘throws’

c) You can change exception type to a child class exception type.

public -> protected -> default -> private

Example of Runtime polymorpihism:

```
class A {
    void fun() { sop(“A”); }
}
class B extends A {
    void fun() { sop(“B”); }
}
    A obj1 = new A( );
```

```
B obj2 = new B( );
obj1.fun( );//A
obj2.fun( );//B

A obj3 =new B( );//object of child class is assigned to parent reference
obj3.fun( );//B
```

Package:

- Used for modularity.
- Instead of keeping all classes at one place, you can divide them in various categories and store them under different packages.
- Package is used for better organization of classes.
- Package help for logical grouping of classes.
- Package is also useful in controlling access (scope).

Class with package scope(default) can be accessible only within same package

Class with public scope can be accessible everywhere

- Java also has it’s own packages:
    - java.lang - All commonly used classes(String, System, all wrapper classes, etc)
    - java.util - Utility classes (All collection related classed, Date, Calendar etc)
    - java.sql - Classes to implement JDBC(Database connectivity)
    - java.net- Networking related classes (Socket programming)
    - java.io - All input/output related classes
    - java.awt - AWT - UI classes used for desktop UI development
- Package also helps to achieve encapsulation (expose limited classes to

outside world by keeping them public, other classes can be kept hidden within the package)

- “package” keyword: Used to specify package name for a class.

This package declaration has to be on 1st line of java source file.

e.g. If I want to put “class A” under package “com.pga” then following is the syntax:

```
package com.pga;
public class A {  }
```

- “import” keyword : Class in one package can access public class of another package by importing it.

e.g. class B in “pkg1” want to access public class from package “com.pga” then following is the syntax:

```
package pkg1;
import com.pga.A;
class B {
    //using A for some reason
```

}

- Package name can consist of any number of words (‘dot’ . in between)

e.g.

package pga;
package com.pga;
package com.pga.maths;

- When we say class A belongs to package “com.pga” then .class file of class A (A.class) must present in folder structure “<project_dir>/com/pga/A.class”.

JAR file

```
package com.pga.maths;
public class Addition {
    public int add(int a, int b) { return a + b; }
}

package  com.pga.stats;
import com.pga.maths.Addition;
public class Average {
```

```
        public int avg(int a, int b) {
            Addition obj = new Addition( );
            return obj.add(a,b)/2;
        }
    }
```

Test.java

```
    import com.pga.stats.Average;

class Test {
    public static void main(String args[]) {
        int a =10, b = 20;
        Average obj = new Average();
        System.out.println(obj.avg(a,b));
    }
}
```

```
    javac -d . Addition.java
    javac -d . Average.java
    javac -d . Test.java
```

Run: java Test (Test belongs to default package)

If “Test” class belongs to other package “com.test” then

#java com.test.Test

Access specifiers :

- private: Accessible only within same class
- default(package level) : Accessible within same package. Not accessible outside package.
- protected - Accessible in same package. It is also accessible outside package only in sub-class(inheritance).
- public- Accessible everywhere

Ex.

```
package pkg1;
public class A {
    private int a = 1;
    int b = 2;
    protected int c = 3;
    public int d = 4;
    public static void main (String[ ]
```

```
args ) {
            A obj = new A( );
            //print all 4 properties
        }
    }
```

Scenario 1 : Same class (A)

Scenario 2 : Class in same package B

```
        package pkg1;
        class B {
            main( ) {
                //create object of A and try to
access all properties
            }
        }
```

Scenario 3: Sub-class of A in same package

```
        package pkg1;
        class C extends A {
            main( ) { //Create object of “A”
and try to access all properties }
        }
```

Scenario 4: Sub-class of A in different

package

```
package pkg2;
class D extends A {
    main( ) { //Create object of “A” and try to access all properties }
}
```

Scenario 5: Class in different package accessing A

```
package pkg2;
class E {
    main( ) { //create object of A and try to access properties }
}
```

Abstract class :

- A class with combination of abstract and non-abstract methods.
- “abstract method”: A Method which doesn’t have definition and it ends with semicolon.

e.g.

abstract void fun( );

- Definition of abstract method will be provided by child class.
- Abstract method mandates child class to override abstract method.
- Abstract class can consist of 0 or any number of abstract methods.
- We can not instantiate abstract class(Object of abstract class can not be created)
- Abstract class can have a constructor and it will be called during child class object creation.
- When a sub-class(child class) inherits abstract class, child class must override all abstract methods from parent class. If child class do not override any single abstract method, then child class also need to be declared as “abstract” class.
- We use abstract class when we want some methods to be written by child class compulsorily.

e.g.

```
abstract class Vehicle  {
    float calculateAvg( ) {
        //logic of average calculation is same across all vehicle, so it can be kept in parent class
    }
    //changeGear() {}
    abstract void changeGear( ) ; //abstract - because all vehicles(2wheeler,4wheeler) have    different mechanic of gear changing
}
class FourWheeler extends Vehicle  {
}

class Electronics { }
```

interface:

- It's a type in Java like a class. (Total 3 types in Java- class, interface, enum)
- Syntax of declaring interface:

```
interface <interface-name>{
    //constants
    //abstract methods
}
```

- All properties of interface are by default public static final. (constant)
- All methods of interface are by default abstract. (public abstract)
- Interface can have only abstract methods. Hence it is also called as pure abstract.
- One class can inherit an interface using “implements” keyword.

e.x.

```
interface A{
  void f( );
}
class B implements A{
    public void f( ) { //code }
}
```

- Multiple inheritance is possible with interface

```
interface A {  void f1( ); }
interface B {  void f2( ); }
class C implements A, B {
    public void f1( ) { }
    public void f2( ) { }
}
```

- Interface can not be instantiated (We can not create object of interface)
- Interface can not have constructor.
- One interface can inherit another interface using “extends” keyword.

```
interface A {
        void f1( );
}
interface B extends A {
    void f2( );
}
class C implements B {
        // C has to write both f1( ) and f2( )
}
```

- SAM interface (Single abstract method

interface)

An interface with only one abstract method is called as “SAM”/Functional interface(this concept is introduce in Java 8)

e.g. Runnable

```
interface Runnable {
    public abstract void run( );
}
```

- Marker interface :

An interface without any abstract method is called as “marker” interface.

This interface is used to give some special instruction to JVM.

e.g. Cloneable, Serializable are marker interface.

class Emp implements Cloneable { } => This tells JVM that allow cloning on employee objects

class Student implements Serializable { } => This tells JVM that allows serialization on student objects

- When class inherits interface and fails to override any of abstract method from interface, then that class needs to be declared as “abstract” class.

Ex of using multiple interfaces:

```
interface Arithmetic {
        int add(int a, int b);
        int sub(int a, int b);
        int mult(int a, int b);
        int div(int a, int b);
    }
    interface Trignomatory {
        int sin(int x) { }
        int cos(int x) { }
        int tan(int x) { }
    }
    class BasicCalculator implements Arithmetic {  }
    class ScientificCalculator implements Arithmetic, Trignomatory { }
```

Example of multiple inheritance:

```
interface A { }
interface B { }
class C { }
class D extends C implements A, B { }
```

class class => extends
interface class => implements
interface interface => extends
class interface = > NA

Exception Handling :

What is exception?
Exception is runtime error which causes program to terminate abnormally.

If exception occurs and not handled in program, then JVM terminates the program execution.

How to handle an exception?
- Use "try-catch" block.

"try" block:
- A block which consist of lines of code

where exception can occur.

- syntax:

```
try {
    //code
}
```

- Every "try" block must be followed by "catch" and/or "finally" block.
- Whenever exception occurs in code written inside "try" block, JVM searches for "catch" block and if matching catch block is found it takes the program control inside "catch" block.

- It is good practice to have minimum code inside "try" block. We can always write multiple "try" blocks in code.
- Consider following scenario :

```
try {
        1-4 line (no error)
        5 line <exception occurs>
        6-10 lines
```

}catch(ArrayIndexOutOfBoundsException e) {

}

- In above code if exception occurs on line 5, JVM jumps to catch block and then 6-10 lines inside "try" block are skipped.

"catch" block:
- An exception handler block.
- Syntax:
  catch(<Exception-class> obj) { // exception handler }
- It has to be written after try/catch block.
- We can have multiple catch blocks to one try block.

Exception handling doesn't solve any exception, it just helps to continue program execution.

"finally" block:
 - If you want to write some code which needs to be executed in both success(exception doesn't occur) and failure(exception occurs) scenarios then "finally" block can be used.
 - Generally cleanup code(closing a file, closing db connection, closing socket connection etc) is written inside "finally" block, because this code is required in both succes/failed cases.
 - "finally" block is executed in both scenarion(success/failure).
 - "finally" is executed only corresponding "try" block is hit.
 - "finally" block will not be executed if "System.exit(0)" comes before hitting the finally block. (exit takes priority and program will be terminated.)
 - We can have at the max one finally block. (one finally per try block)
 - If we have catch blocks for try, then finally

should be written at the end(after last catch block)
 - syntax:
    finally { //cleanup code }

3 possible cases :
1) Success (no execption at all)
    try - complete
    finally - complete
    remaining code - complete
2) Exception occurs and matching catch block is found
    try - half try (code before exception)
    catch - matching catch block
    finally - complete
    remainig code - complete
3) Exception occured but no matching catch block found
    try - half try (code before exception)
    finally - complete

remainig code - NO. Program terminates

```
start:Division
Some exception occurred
java.lang.ArithmeticException: / by zero
    at
AdditionDemo.main(AdditionDemo.java:9)
0
end:division
```

Exception classes Hierarchy:

Ex. of checked exceptions:

- IOException
- ClassNotFoundException
- CloneNotSupportedException
- ServletException
- FileNotFoundException

Ex. of unchecked exceptions:

ArrayIndexOutOfBoundsException
NullPointerException
ArithmeticException
NumberFormatException
IllegalArgumentException

Types of exceptions:

1. Checked Exceptions
2. Unchecked exceptions

Checked Exceptions :

- Exceptions whose parent is “Exception” class
- If compiler detects the possibility of checked exception in program, then compiler would mandate to handle the checked exception.

e.g. when we call “readLine( )” method of “BufferedReader” there is possibility of IOException. No IOException is checked exception and hence compiler

would enforce to handle the IOException. (public String readLine() throws IOException)

- Compiler strictly checks for checked exceptions.
- These are more critical then unchecked exceptions.

Ex. IOException, ClassNotFoundException, CloneNotSupportedException, ServletException etc

UnChecked exceptions:

- Exceptions whose parent is “RuntimeException”
- Compiler wouldn’t check for these exceptions.
- Not mandatory to handle.
- Less critical

ex. ArrayIndexOutOfBoundsException, NullPointerException, ArithmeticException, NumberFormatException etc.

“throws” keyword:

- This keyword is written in method signature.
- syntax is

    method( ) throws <Exception1>,<exception2>…

  e.g.

    public String readLine( ) throws IOException { }

    protected void doGet() throws ServletException, IOException { }

- Generally checked exceptions are written after “throws” keyword
- Writing “throws” in method signature indicates outside world about possibility of mentioned exceptions from that method.
- When we call a method having throws with checked exception, caller has to handle/throw the checked exception written with throws.
- “throws” is also useful to bypass handling of an exception to the caller.

“throw” keyword :

- It is used to throw an exception explicitly.
- Java allows programmer to throw any exception using “throw” keyword.
- Syntax:

  throw new <Exception-class> ( );
- It is generally used to throw user-defined exceptions.
- JVM also uses “throw” keyword. Whenever exception occurs, JVM create an object of appropriate exception class and throws that exception using “throw” keyword.

Example of using “throw” :

We want to throw “IllegalArgumentException” if argument count is not 3.

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

```
        if(args.length != 3) {
            System.out.println("Invalid
number of arugments: need 3");
            throw new
IllegalArgumentException();
        } else {
            System.out.println("correct
argumetns are passed");
        }
        }
}
```

Exception Propogation:

- Whenever exception occurs JVM checks if that exception is handled in current function.
- If it is not handle in current function then that exception is bypassed to the caller

e.g.

JVM -> main( )-> f1( ) -> f2( ) -> f3( )

Now if exception comes in f3( ),

1. JVM checks if it is handled in f3( ), if not then it will be propagated to f2( )
2. If f2( ) also do not have handling of that exception, then it will be propagated to f1( )
3. Now if f1( ) handles that , then program will continue .

MultiThreading:-

What is program?

Set of instructions. Program is stored in secondary storage device(HDD)

What is process?

Running instance of a program is called as process. Processes are stored in primary memory(RAM).

Context switch?

When CPU switches from one process to other process, it is called as context switch.

How CPU executes process?

- One CPU can execute one instruction of one process at a time.
- Execution speed of CPU is so fast and hence we feel that multiple processes are running in parallel.
- CPU tries to give equal chance to all running processes and for that it uses scheduling algorithm(e.g. Round Robin)

What is Thread?

- Thread is created from process.
- It is also called as “lightweight process”. (Since it doesn’t require any extra memory, it uses memory from process)
- Thread is independent path of execution.
- Every thread should have specific responsibility to perform.

Multithreading:

- When multiple threads of same process running in parallel then it called as “multi-threading”.
- Two threads can run in parallel if they are independent.

Examples of multi-threaded application:

Browser application threads
- Browsing
- Printing
- Downloading
- Uploading

Possible threads in Email application :
- Compose email thread
- Attachment thread
- Send-Email thread
- Search-Email thread
- Email-Sync thread (syncing with server)

JVM thread scheduler:
- Manages scheduling of Java threads

- If there are 4 threads of one java process, then JVM thread scheduler decides sequence of execution of 4 threads using scheduling algorithm.

P1 P2 P3 P4 : CPU scheduling

P2 -> T1,T2,T3 (JVM thread scheduler)

How to implement threading in Java?

- There are 2 ways you can define threads in java.
    1. Write a class which “extends Thread” class
    2. Write a class which “implements Runnable” interface

In both mechanisms we have to define thread logic inside “public void run( )”

method.

1. Thread using “extends Thread” mechanism:

```
class DownloadThread extends Thread {
    public void run( ) {
        //logic of downloading
    }
}
```

2. Thread using “implements Runnable” mechanism:

```
class PrintingThread implements Runnable {
    public void run( ) {
        //logic of printing
    }
}
```

Above are template where we write logic of threads. Now how to run these threads?

1. How to start a thread which is created

using “extends Thread”?

```
DownloadThread t1 = new DownloadThread( );
t1.start( );
```

2. How to start a thread which is created using “implements Runnable”?

```
Thread t2 = new Thread(new PrintingThread());
t2.start( );
```

Relation between “Thread” class & “Runnable” interface:

- Thread class internally implements Runnable interface.

dig.

**public void run( ):

- Thread logic is writing inside this method.
- This method actually belongs to “Runnable” interface.

**public void start( ):

- Thread creation requires lot of low level tasks like memory allocation, stack creation etc. All of these low level tasks will be performed by “start( )” method.
- start( ) method internally calls run( ) method.
- We should always call start( ) method to start the thread.

We write logic of thread inside run( ) method, but to start the thread we call start( ) method and start( ) method internally calls run( ) method.

join( ) method:

-Current thread waits for complete execution of a thread on which join( ) is called.

Inter-thread communication :

- In multithreading if there is a dependency then one of the following mechanism can be used.

1. join( )
2. wait( )-notify( )/notifyAll( )

wait( )-notify( )/notifyAll( )

- These are method from “Object” class.
- Waiting thread calls “wait( )” on object.
- Notifying thread has to call “notify( )” on same object on which wait( ) was called.
- Waiting thread do not have to wait for complete execution, waiting thread can resume the execution immediately once it gets notified.
- In wait-notify, waiting time is less and both threads run in parallel.

In join() waiting time is more and both threads can not run in parellel.

- If multiple threads are waiting on same

object and if some thread
calls notify( ) , then only one of the waiting threads will be notified.
In order to notify all waiting thread on one objected, you have to
call notifyAll ( ).
- wait(),notify()/notifyAll() methods have to be called inside
"synchronized" block.
- "IllegalMonitorStateException" comes if wait( )/notify() are not written inside synchronized block.

Synchronization:
- Mechanism of controlling concurrency.
- When multiple threads do "write" operation on
same object at a time, then object state will not be
consistent. In such case only one thread should
allowed to access an object.

- Synchronization is used to achieve above goal.
- When synchronization is applied:
a. No two threads can go in same function on
same object.
b. Multiple threads can execute same code on
different objects.
- There are 2 ways of applying synchronization
a) At method level (By making method as “synchronized”)
e.g. public synchronized void withdrawl(float amt) {}
b) Synchronized block.

```
public void withdrawl(float amt) {
…
synchronized(this) {
this.bal = this.bal - art;
}
…
```

}
- Synchronization block is more faster than method, because waiting would be less in case of
block.
- We can make both instance and static methods
as synchronized.
- Whenever we write synchronized methods/block
in class then that class is called as “Thread-safe”
class.
- Synchronization is not required for “read only
operations”(methods which returns some data).
- Synchronization is based on object/class level
locking.
Object level locking:
- To enter into instance synchronized

method(Or
inside synchronized block of instance
method) thread has
to acquire lock on an object on which
method is called.
e.g.
class Account {
public synchronized void withdrawl(float
amt) {
//code
}
}
Account act1 = new Account(101,10000);
Account act2 = new Account(201, 4000);
If T1 calls withdrawl() on act1 =>
- JVM checks type of method and figures
out
that it is synchronized.
- JVM checks if any lock is already applied
on
“act1” object?
- If no, then JVM would apply lock of

thread T1 on object “act1” and allows T1 to execute
withdrawl() method.
- If lock is already applied, then JVM put T1 in waiting queue and allowed to enter in
method only when previous lock is released.
- On one object only one lock can be applied. -
Lock is not required to execute non-synchronized
methods.
Class Level Locking:
- Static methods can also be synchronized.
- When any thread want to execute static synchronized method or synchronized block inside
static method, then thread has to acquire lock at class
level.
ex. class Abc {

static synchronized void f1( ) {}
}
Now thread T1 wants to call Abc.f1() =>
- JVM checks type of method and figures out
that it is static synchronized.
- JVM checks class level lock
a. If lock is not applied at class, then JVM applies lock of thread T1 on class and allow thread to
execute f1( )
b. If lock is already applied at class, then JVM would put T1 in waiting queue.
There is no relation between class level lock and
object level lock. They both are indpendent.
class A{
public synchronized static void f1 ( ) { }
public synchronized void f2() {}
}
T1 => obj1 => f2( ) — Lock is applied on object “obj1”

T2 => A = > f1( ) — Lock is applied on class “A”

Example of object level and class level locking:(IMP for interview)

Given following

```
class A {
synchronized void f1 ( ) {} //instance synchronized
synchronized void f2 ( ) {} //instance synchronized
void f3 ( ) {} //instance
synchronized static void f4 ( ) {} //static synchronized
synchronized static void f5 ( ) {} //static synchronized
static void f6 ( ) {} //static
}
class B {
static synchronized void f1( ) {}
}
```

1) Current: T1 -> obj1 -> f1 - lock is applied

on object
“obj1”
T2 -> obj1 -> f2 () => (No)
T3 -> obj1 -> f3( ) => (yes)
T4 -> obj2 -> f2( ) => (yes)
T5 -> A -> f4( ) => (yes)
T6 -> obj1 -> f1( ) => (no)
2) Current: T1 -> A -> f4 (A.f4()) - lock is applied on

Collection:
- Collections is group of objects.
- Array has limitation of size, once created we can
not change size of an array.
- All collections are dynamic
- Major categories under Collection - Set, List,
Map
- All collections interfaces/collections comes under “java.util” package.
- For each collection we have to see

following
operations.
Operations :
- Add
- Delete
- Retrive(possible with List)
- Search
- Modify
- Traversing
- Sorting

Set - Group of unique objects(Doesn’t allow duplicates)
- HashSet - Unordered
- LinkedHashSet - Ordered
- TreeSet - Sorted

List- Group of objects where repetition is allowed. It
is index based and sequential collections. All classes
under this are “ordered”.
- Vector : Synchronized
- ArrayList : Not synchronized

- LinkedList : Used where frequent modifications(delete, insert, update) are required.
Collections.sort(<list>)
Map: Used to store data in pair(key-value pair)
- Hashtable - UnOrdered, synchronized, doesn’t
allow null
- HashMap - UnOrdered, Not synchronized, Allows null
- LinkedHashMap - Ordered
- TreeMap - Sorted
Syntax for creating collection:

```
Set<Integer> set1 = new HashSet<Integer>();
List<Student> list = new ArrayList<Student>();
Map<Integer, Float> map = new HashMap<Integer,Float>();

List list = new Vector();//no type specified,
```

so
anything can be added
- Duplicate got allowed
- Not able to use TreeSet
- Search is giving wrong result

hascode( ) & equals( ) contract:

When we use Hash based collections (HashSet, LinkedHashSet,HashMap, Hashtable, LinkedHashMap)
it is important to add equals( ) and hashCode( )
methods in user defined class used with collections.
e.g. If we are creating HashSet of Employee
objects then we must override equals() & hashCode()
methods of Object class.
equals() & hashcode() methods:
- These both methods are from Object class

- By default both returns result based on memory
address.
- We can override these both by considering data
of object(and not memory)
- These methods are always written in pair. (so if
we write hashcode() then add equals OR vice versa)

1. If hashcode(obj1) == haschode(obj2) then
   obj1.equals(obj2) may or may not be true

   If hashcode of two objects is same then they may or may not be equals.

2. If obj1.equals(obj2) is true then
   hashcode(obj1) == hashcode(obj2) .

If equals on two objects is same then they hashcode of both objects MUST be same.

Hash function :

- A function which should not have any random behavior.
- For same value hashcode should result same.

In generate