INDEX :

JAVA BASICS :

→ Code is written in .java file. And, it is compiled by java compiler.

∴ Command : javac filename

→ After compilation, compiler creates .class file which is understandable by Java Virtual Machine (JVM).

→ Now, to execute the program, the command is :

java classname.

→ If running your code from command prompt, don't forget giving the path of your project folder using cd to change the directory.

→ Always create packages under project. And create classes under packages.

Naming Conventions :

→ For classes, we use 'Pascal Convention'. First & subsequent characters from a word are uppercase letters.

Eg: Main, MyEmployee, TestBooking etc.

→ For methods & variables, we use 'Camel Case Convention'. First character is lowercased & the subsequent characters are uppercased.

Eg: myEmployee, availableSeats etc.

✱ OOPS concepts :

→ Object creation :

```
Student S1 = new Student();
```

→ Here, S1 is the reference to call properties from the Student class.

→ new is a Keyword which creates an object.
Operator

→ It instantiates a class by allocating (dynamically) memory for a new object and returning reference to that memory. That reference is stored in variable S1.

→ If properties like name, age, marks are set to private, then it cannot be accessed outside of class.

→ We can add getters and setters to access these properties.

→ These are used to protect our data & make our code more secure.

→ this is a reference variable that refers to the current object. It is used to call current class methods.

→ In java, Encapsulation is achieved by declaring the instance variables of a class as 'private', which means they can be accessed within the class itself.

→ To allow outside access, public methods called setters & getters are defined, which are used to retrieve & modify the instance variables.

STATIC vs NON-STATIC

→ When a variable is declared as static, then a single copy of the variable is created and shared among all objects at a class level.

→ If data is 'unique' to object then, maintain such data through 'instance variable'.

→ If data is common to all objects, then maintain such data through 'static variable'.

```
int age;
String name;
int rollNo;
```
} This changes for all objects.

```
static String university = "CMR";
```
} This stays same for all objects.

→ Whereas, a non-static variable is a instance variable which is accessible only within the block it is created using instance of the class.

→ A non-static method cannot be referenced from static content.

☆ METHOD OVERRIDING :

→ Redefining a super class method in sub class is called Method overriding.

→ If an object of a parent class is used to invoke the method, then the version in the parent class is executed. But if object of the subclass is used to invoke the method, then version in the child class is executed.

Example:

```
class parent {
    void show(){
        System.out.println("Parent's show()");
    }
}
```

```
class child extends parent {
    void show() {
        System.out.println ("child's show()");
    }
}

class Main {
    public static void main (String[] args) {
        parent obj1 = new Parent();
        obj1.show();
        parent obj2 = new child();
        obj2.show();
    }
}
```

→ If a parent type reference refers to a parent object, then parent's show is called.

→ If a parent type reference refers to a child object, then child's show is called.

→ Method overriding is one of the ways to achieve Runtime polymorphism.

→ Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

→ " At runtime, it depends on the type of the object being referred to, that determines which version of an overridden method will be executed.

→ Superclass obj = new Subclass();

A Superclass reference variable is referring to a Subclass object. This is called Upcasting.

★ RULES FOR OVERRIDING :

→ The access modifier for overriding method can allow more, but not less, access than the overridden method.

→ For eg: A protected instance method in the Super class can be made public, but not private, in the Sub class.

→ Weak to Strong access order :

private < default < protected < public

(Less access) (More access)

→ The sub class method access specifier should be same or stronger than superclass method access specifier.

→ If we don't want a method to be overridden, we declare it as final.

→ When you define a static method with the same signature as a static method in the base class, it is known as Method Hiding.

→ private methods cannot be overridden.

→ '@Override' is an annotation used to tell the compiler that method is being overridden. So that it will check if all the rules are followed or not.

☆ INHERITANCE :

→ It is the mechanism in java by which one class is allowed to inherit the features (fields & methods) of another class.

→ A class that inherits from another class can reuse the methods of that class. New fields & methods also can be added.

→ Useful features provided by inheritance:

Code Reusability
Method Overriding
Abstraction

→ The extends keyword is used for inheritance in java.

```
class derivedClass extends baseClass {
    // methods
}
```

JAVA INHERITANCE TYPES:

i) Single inheritance:

A subclass is derived only from one super class.


```
class One {
}
Class Two extends One{
}
```

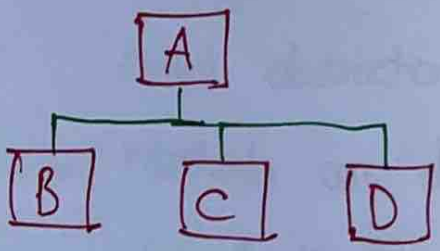
ii) Multilevel inheritance:

A derived class will be inheriting a base class, and as well as the derived class also acts as base class for other classes.


```
Class One {
}
Class Two extends One {
}
class Three extends Two{
}
```

iii) Hierarchical inheritance :

One class serves as Superclass for more than one Subclass.



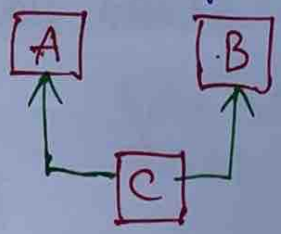

```
class A {
}
class B extends A {
}
class C extends A {
}
```

iv) Multiple inheritance :

One class can have more than one Superclass and inherit features from all parent classes.

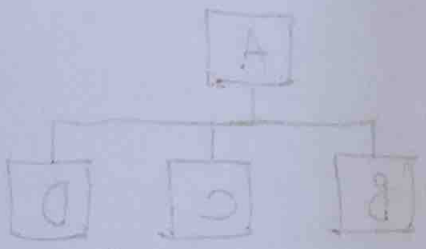
→ Java does not support multiple inheritances with classes.

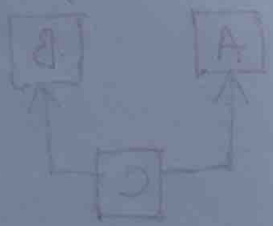
→ It can be achieved only through java interfaces.




```
interface One {
}
interface Two {
}
interface Three extends One, Two {
}
```

```
class child implements Three {
}
```





☆ ABSTRECTION :

→ The idea of hiding and providing Standardization in project is called abstraction.

→ Methods are declared in Superclass and defined in Subclasses.

→ abstract is the keyword used to create abstract methods or classes.

```
public abstract class Polygon {
        abstract void info();
}

public class Square extends Polygon {

        @override
        void info() {
                System.out.println("All Sides are Same");
        }
}
```

→ If any of the methods in a class is abstract, then the class should be defined abstract.

→ In java, abstract method means method declaration.

→ If class is abstract, object creation is not possible.

JAVA INTERFACES: Uses implements keyword to access interface methods.

→ Interfaces are another way to achieve abstraction in java. It is a completely abstract class used to group related methods with empty bodies.

Then what is the key difference between abstract classes & interfaces?

Consider below Snippet:

```
public abstract class Polygon {
        void
        abstract void info();
}
public abstract class Arithmetic {
        abstract void quotient (int x, int Y);
}

// Interfaces with same methods:
public interface Polygon {
        void info();
}
public interface Arithmetic {
        void quotient();
}
```

→ Now, with classes multiple inheritance isn't supported.

```
class calcuation extends Polygon, Arithmetic {
} // This is not possible.
```

→ But with interfaces, it is supported.

```
class calculation implements Polygon, Arithmetic {
} // Possible.
```

→ Other differences are :

i) public static final are implicit keywords for constant in interface.
public abstract interface are implicit keywords for methods.

ii) Variables & constants (final variables) can be created in abstract classes but not in interfaces.

iii) Abstract methods & non abstract methods can be written in abstract classes but non-abstract methods cannot be written in interfaces.

iv) Constructor can be created & called in abstract classes but not in interfaces.

☆ RUNTIME POLYMORPHISM (IN DEPTH):

Let us look at an example code to under dynamic polymorphism.

Polygon.java

```
public interface Polygon {
        Void info();
}
```

Square.java

```
public class Square implements Polygon {
        @Override
        public void info() {
                System.out.println("All sides are same");
        }
}
```

Rectangle.java

```
public class Rectangle implements Polygon {
        @Override
        public void info() {
                System.out.println("Opposite sides are same");
        }
}
```

PolygonManger.java

```
public class PolygonManager {
    static Polygon getPolygon(int s1, int s2, int s3, int s4){
        if (s1 == s2 && s2==s3 && s3==s4) {
            return new Square();
        }
        else if (s1==s3 && s2==s4){
            return new Rectangle();
        }
        else {
            return null;
        }
    }
}
```

TestPolygon.java

```
public class TestPolygon {
    public static void main(String[] args){
        polygon ~~get~~ p1 = PolygonManager.getPolygon(10,10,10,10);
        polygon p2 = PolygonManager.getPolygon(10,20,10,20);
        p1.info();
        p2.info();
    }
}
```

→ Polymorphism allows objects to be treated as instances of their parent type. It can occur at compile time (method overloading) or runtime (method overriding).

→ The above code has an interface Polygon which defines single method info(); which is implemented differently by different classes Square and Rectangle.

→ The PolygonManager class has a static method getPolygon() that determines which object to create based on the dimensions passed as arguments.

→ TestPolygon class tests the polymorphism by calling getPolygon() method and invoking the info() method on the returned objects.

→ The variable p1 of type Polygon holds a reference to an object.

→ At runtime, the actual type of the object determines which info() implementation is executed.

→ When P1.info() is called, the runtime type of p1 is Square, So the info() method in Square is executed, printing:

All sides are Same.

→ Same happens with p2.info() method.

EXCEPTION HANDLING:

→ An exception is a runtime error. When an error occurs, java will normally stop and generate an error message. Which means, it is throwing an Exception.

→ With this, regular flow of code/application can be preserved.

→ Here comes try and catch statements.

→ Try block allows us to define a block of code to be tested for errors while it is being executed.

→ Catch block allows us to define block of code to be executed, if an error occurs in try block.

Consider the following code snippet:

```
public class Calculator {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        int num1 = 0;
        int num2 = 0;
        int result;

        try {
            System.out("Enter Number 1 : ");
            num1 = sc.nextInt();
```

```
        Sysout ("Enter Number 2 : ");
        num2 = sc.nextInt();
        sc.close();

        result = num1 / num2;
        Sysout ("Quotient : " + result);
    }
    catch (ArithmeticException e) {
        Sysout ("Enter valid numeric value");
    }
    catch (InputMismatchException e) {
        Sysout ("Unexpected input");
    }

    result = num1 + num2;
    Sysout ("Addition : " + result);

}
```

→ The doubtful part of the code which is prone to error is placed inside try block.

→ Here, the exceptions ArithmeticException and InputMismatchException are handled.

How does all this work ?

→ Whenever we are encountering an exception inside a method, the method creates an object called as Exception object and hands it off to the run-time system. (JVM).

→ Error vs Exception :

i) An error indicates a serious problem that a reasonable application should not try to catch.

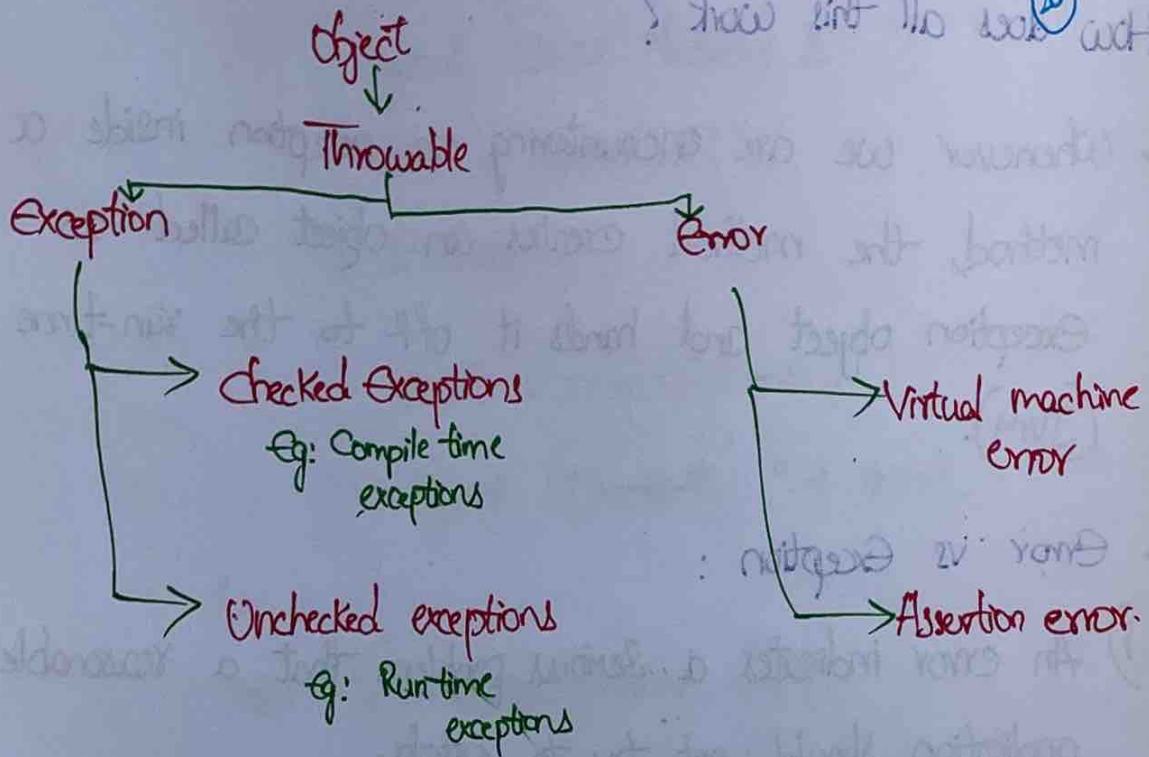
ii) An exception indicates conditions that a reasonable application might try to catch.

☆ EXCEPTION HIERARCHY :

→ All exceptions & errors types are subclasses of the class throwable, which is the base class of the hierarchy.

→ Exception class is used for exceptional conditions that user programs should catch.

→ Error is used by the Java runtime system (JVM) to indicate errors having to do with the run-time environment itself.



→ Examples of checked exceptions :

ClassNotFoundException
InterruptedException
IOException
FileNotFoundException

→ Examples of unchecked exceptions :

ArithmeticException
NullPointerException
ArrayIndexOutOfBoundsException

→ Sometimes, built-in exception are not able to describe certain situation. In such cases, users can custom create exceptions, called User-defined Exceptions.

METHODS TO PRINT THE EXCEPTION INFORMATION:

i) printStackTrace()
ii) toString()
iii) getMessage()

→ Sometimes, built-in exception are not able to describe certain situation. In such cases, users can custom create exceptions, called User-defined Exceptions.

METHODS TO PRINT THE EXCEPTION INFORMATION:

i) printStackTrace()
ii) toString()
iii) getMessage()

FILE HANDLING :

→ File handling is integral part as it enables us to store the output of any particular program in a file & allows us to perform certain operations on it.

→ In java, a sequence of data is known as Stream. This concept is used to perform I/O operations on a file.

→ There are two types of streams :

i) Input Stream :

→ The java 'InputStream' class is the superclass of all input streams.

→ InputStream is an abstract class, & it is not useful by itself. However, it's subclasses are used to read data.

→ Subclasses of InputStream class are as follows :

a) FileInputStream
b) StringBufferInputStream
c) ObjectInputStream

ii) Output Stream :

→ The output stream is used to write data to numerous output devices.

→ OutputStream is an abstract superclass that represents an output stream.

→ As it is an abstract class, it is not useful by itself, however, it's subclasses are used to write data.

→ Subclasses of OutputStream are as follows:

a) FileOutputStream
b) StringBufferOutputStream
c) ObjectOutputStream.

→ Based on the data type, there are two types of streams:

i) Byte Stream : read & write byte data.
ii) Character Stream: read & write character data.

☆ FileInputStream class in java:

→ FileInputStream is useful to read data from a file in the form of sequence of bytes.

☆ FileOutputStream class in java:

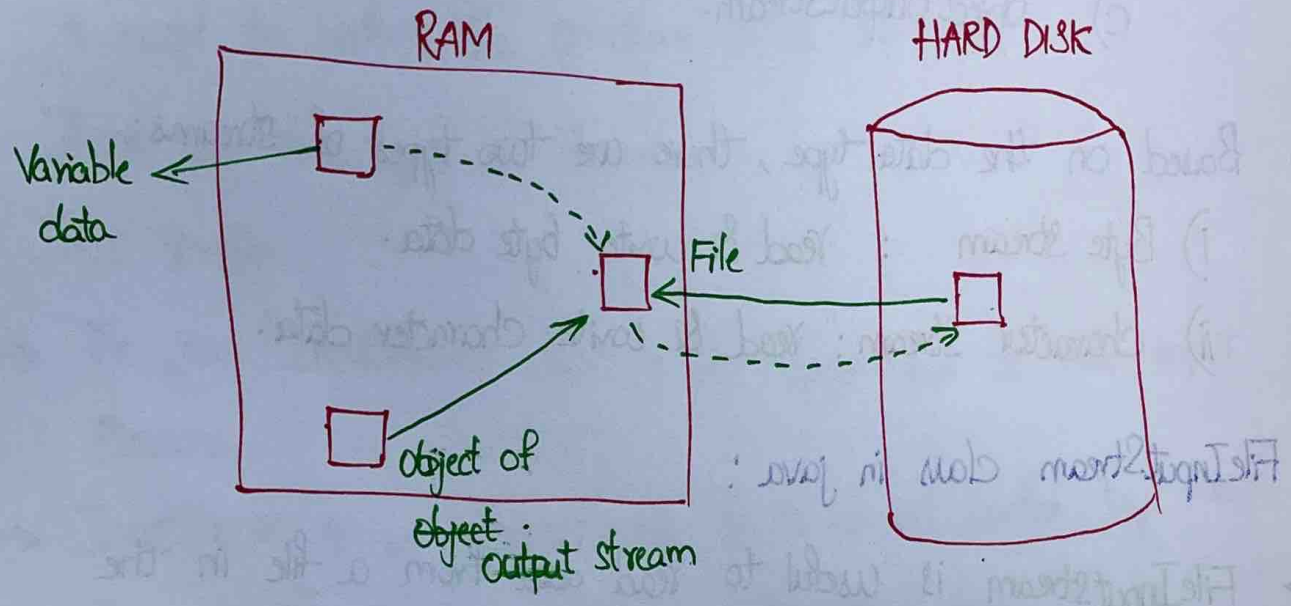
→ FileOutputStream is used to write data of raw bytes to file.

→ When a java program is made to run, the data is stored in the RAM.

→ Now, Suppose the variable data stored in RAM, we want to access that data & bring it to a file in our hard disk.

→ So, we will create an object of ObjectStream in the RAM and that will point to a file referencing to hard disk.

→ The data from the variable data file in the RAM will go to the referring file (object of outputstream) & from there, it will be transferred/stored in the file of the hard disk.



→ Small code snippet to understand File I/O Streams:

StudentFile.java:

```
import java.io.Serializable;
public class StudentFile implements Serializable {

    String name;
    int age;
    int marks;
    int rollno;

    // Generating setters & getters
```

```
public String getName() {
    return name;
}
Public void setName(String name) {
    this.name = name;
}
public int getAge() {
    return age;
}
public void setAge(int age) {
    this.age = age;
}
public int getMarks() {
    return marks;
}
public void setMarks(int marks) {
    this.marks = marks;
}
public int getRollno() {
    return rollno;
}
public void setRollno(int rollno) {
    this.rollno = rollno;
}
```

```
public String toString() {
    return "Name : " + name + " Age : " + age +
        " Marks : " + marks + " Roll no : " +
            rollno ;
```

FileTransactions.java :

```
import java.io.FileInputStream;
public class FileTransactions {
    void readFile() throws IOException {
        FileInputStream fis = new FileInputStream
                ("src/package_name/abc.txt");
        int data = fis.read();
        while (data != -1) {
            System.out.println ((char) data);
            data = fis.read();
        }
        fis.close();
    }
```

```
    Void writeFile() throws IOException {
        FileOutputStream fos = new FileOutputStream
                ("src/package_name/xyz.txt");
        fos.write('B');
        fos.close();
    }
}
```

→ When writing readFile() method, a text file should be created by the user upfront if using FileInputStream.

→ However, when writing into file using FileOutputStream, the file is created by itself, as we describe in the code.

TestFileTransactions.java :

```
import java.io.IOException;{
public static class TestTransactions {
        public static void main (String[] args) {
            FileTransactions fileTransactions =
                    new FileTransactions();
            fileTransactions.readFile();
            fileTransactions.writeFile();
        }
}
```

→ The data from the text file which is fetched by FileInputStream is of int datatype.

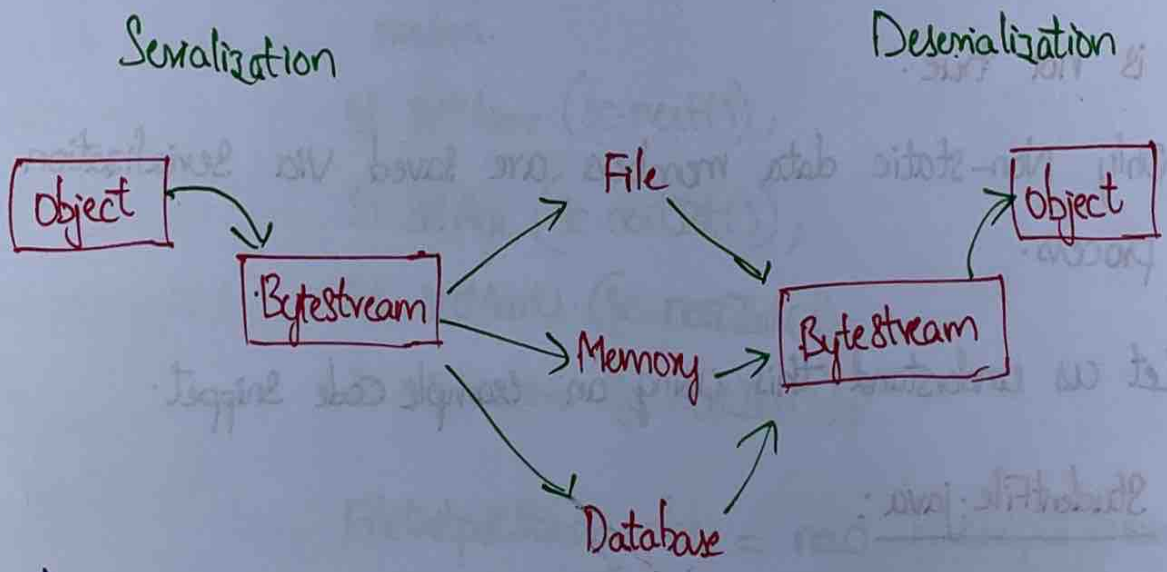
→ That's why we have ^(type) casted the i data to char.

→ If file doesn't contain anymore data, then it returns -1.

☆ OBJECT SERIALIZATION & DESERIALIZATION :

→ Serialization is a mechanism of converting the state of an object into a byte stream.

→ Whereas, deserialization is the reverse process, where the byte stream is used to recreate the actual java object in the memory.



→ Serialization & Deserialization are crucial for saving and restoring the state of objects in java.

→ To make java object Serializable, we implement the java.io.Serializable interface.

→ The ObjectOutputStream class contains writeObject() method for Serializing an object.

→ The ObjectInputStream class contains readObject() method for deserializing an object.

→ Only the objects of those classes can be Serialized which are implementing java.io.Serializable interface.

→ Serializable is a marker interface. (Has no data member & method.

→ If a parent class has implemented Serializable interface, then child class doesn't need to implement it but vice-versa is not true.

→ Only Non-static data members are saved via Serialization process.

→ Let us understand this using an example code snippet.

StudentFile.java:

```
import java.io.Serializable;
public class StudentFile implements Serializable {
        String name;
        int age;
        int rollNo;
        int marks;

        // Generate setters & getters for controlled
            access & toString() method.
```

ObjectSerialEx.java :

```
import java.util.Scanner;
public class ObjectSerialEx {
    public static void main (String[] args) throws IOException {
        Scanner sc = new Scanner(System.in);
        StudentFile s1 = new StudentFile();
        // write print statements to prompt input if
           needed.
        s1.setName (sc.next());
        s1.SetAge (sc.nextInt());
        s1.setMarks (sc.nextInt());
        s1.setrollNo (sc.nextInt());

        FileOutputStream fos = new FileOutputStream
                ("src/package_name/stu.txt");
        ObjectOutputStream oos = new ObjectOutputStream
                (fos);
        oos.writeObject (s1);
```

ObjectDeserialEx.java :

```
import java.io.*;
public class ObjectDeSerialEx {
        public static void main (String[] args) throws
                        IOException, ClassNotFoundException {

            FileInputStream fis = new FileInputStream
                        ("src/package_name/ stu.txt");
            ObjectInputStream ois = new ObjectInputStream
                        (fis);
            StudentFile s = (StudentFile) ois.readObject
                        ();
            System.out.println (s);
        }
}
```

→ In the above code, StudentFile.java has the required properties and getters & setters for controlled access.

→ ObjectSerialEx.java program Serializes StudentFile object into a file named stu.txt.

→ FileOutputStream is used to create a file (text) for writing binary data.

→ ObjectOutputStream is used to Serialize s1 object into the file. (stu.txt)

→ ObjectDeserialEx.java program deserializes StudentFile object from the file stu.txt.

→ FileInputStream is used to read binary data from 'stu.txt' file.

→ ObjectInputStream is used to read the serialized object from the file & prepares it for deserialization.

→ While reconstructing object from the bytestream, typecasting of the object ois is necessary.

→ Without typecasting, the compiler cannot determine the type of the returned object & throws compilation error if we try to assign it directly to a variable of type StudentFile.

→ StudentFile s = (StudentFile) ois.readObject();

In this line, by adding (StudentFile) before the ois object, we inform the compiler that the object being deserialized is of type StudentFile.

→ In Summary,

StudentFile properties

written ↳ FileOutputStream file (stu.txt)

↳ ObjectOutputStream (oos)

Serialized into stu.txt

Serialization

stu.txt

reads ↳ FileInputStream fis

↳ ObjectInputStream ois

deserialized into object

Deserialization

EXCEPTION HANDLING (AGAIN):

FINALLY KEYWORD:

→ finally is a reserved keyword in java. It is used in association with a try/catch block.

→ It guarantees that a section of code will be executed, even if an exception is thrown.

→ Finally block is executed even if try block has a return statement.

→ It is used when a file needs to be closed even if an exception is caused, to avoid resource leak.

```
try {
        // Creating file object fos
        // Writing into file.
}
catch (FileNotFoundException e) {
        e.printStackTrace();
}
finally {
        fos.close();
}
```

TRY WITH RESOURCE :

→ Try-with-resource is a try statement that declares one or more resources in it.

→ A resource is an object that must be closed once the program is done using it.

→ This is an alternative to write finally block.

```
try (declare resources here) {
    .....
}
catch (FileNotFoundException e) {
    .....
}
```

THROW & THROWS IN JAVA :

→ The throw keyword in java is used to explicitly throw an exception from a method or any block of code.

→ Checked & Unchecked exceptions can be thrown.

→ Mainly used to throw custom exceptions.

→ User-defined (custom) exceptions apparently extend the exception class.

→ throws is a keyword that is used in the signature of a method to indicate that this method might throw one of the listed type exceptions.

→ The caller to these methods has to handle the exception using a try-catch block.

```
public class ThrowEx {
    public static void main(String[] args) throws
                                NegativeNumberException {
        Scanner sc = new Scanner(System.in);
        int a;
        a = sc.nextInt();
        sc.close();
        if (a <= 0) {
            throw new NegativeNumberException();
        }
        if (a % 2 == 0) {
            print("Even");
        }
        else {
            print("Odd");
        }
```

→ User defined Exceptions must have a class with same name and should extend exception class.

```
public class NegativeNumberException extends Exception {

}
```

→ Throws keyword is required only for checked exceptions.

ANONYMOUS INNER CLASS IN JAVA :

→ It is an inner class without a name and for which only a single object is created.

→ It is useful when making an instance of an object to use it for overriding methods of a class or interface, without having to subclass a class.

Polygon.java :

```
public interface Polygon {
        Void info();
}
```

Square.java :

```
public class Square implements Polygon {
        @Override
        public void info() {
                print ("Sides Same");
        }
}
```

TestPolygon.java :

```
public class TestPolygon {
    psvm() {
        polygon p1 = new polygon() {
            @Override
            public void info() {
                print("opposites same");
            }
        };
        p1.info();
    }
}
```

→ The method inside the super class or interface must be implemented inside the anonymous inner class.

☆ LAMBDA EXPRESSIONS :

→ Lambda expressions were introduced in Java 8 to provide a concise way of representing anonymous functions.

→ These are well-suited for working with functional interfaces.

```
(parameter 1, parameter 2, ...) -> {

}
```

- Parameters are input values to lambda expressions.
- (→) Arrow token seperates parameters from the body.
- Code inside block gets executed when lambda expression is invoked.

```
interface Adder {
    int add (int a, int b);
}
public class TestAdder {
    public static void main (String[] args) {
        Adder adder = (a,b) -> a+b;
        int sum = adder.add (4,5);
        System.out.print (sum);
    }
}
```

☆ DEFAULT METHODS :

→ Before JDK8, interface could only have abstract methods. The implementation of these methods has to be provided in a separate class.

→ If a new method is to be added in an interface, then it's implementation code has to be provided in the class implementing the same interface.

→ So, default methods were introduced to allow interfaces to have methods with implementation without affecting subclasses.

→ After JDK8, interface allowed non abstract methods. The only condition is they should be default or static.

```
public interface Polygon {
        Void info();
        default Void Show() {
                print ("Default method executed");
        }
}
```

→ Now, the info(); method must be implemented in another class which implements this interface.

→ However, the show() method is default, & is executed without implementation code in other class.

MULTITHREADING :

→ Multi-threading allows concurrent execution of two or more parts of a program for maximum utilization of CPU.

→ Each part of such program is called Thread.

→ So, threads are light-weight processes within a process.

→ Threads can be created using 2 mechanisms :

i) Extending the thread class.

ii) Implementing runnable interface.

i) Thread

→ To implement multi-threading, a class is created, that extends java.lang.Thread class.

→ This class overrides the run() method available in the Thread class.

→ The life cycle of a thread begins inside run() method.

→ start() invokes the run() method on the Thread object.

```
class MultithreadingDemo extends Thread {
    public void run() {   //overridden
        print ("Thread " + Thread.currentThread().getId() +
                " is running.");
    }
}
```

```
public class Main {
    public static void main (String[] args) {
        int n=8;
        for (int i=0; i<n; i++) {
            MultithreadingDemo obj = new
                MultithreadingDemo();
            obj.start();
        }
    }
}
```

ii) Runnable

→ We create a new class which implements java.lang.Runnable interface and overrides run() method.

→ Then we instantiate a Thread object and call start() method on this object.

```
class MultiThreadingDemo implements Runnable {
    public void run() {
        .....
    }
}
```

Then, what is the difference between Thread & Runnable:

→ If we extend thread class, our class cannot extend any other classes because java doesn't support multiple inheritence. But, if we implement Runnable interface, our class can still extend other base classes.

LIFE CYCLE AND STATES OF A THREAD IN JAVA:

→ A thread in java exists in any one of these states at any point of time.

i) New
ii) Runnable
iii) Blocked
iv) waiting
v) Timed waiting
vi) Terminated.

i) New: Instance of a thread is created, which is not yet started by invoking start().

ii) Runnable: After invokation of start() and before selected to run by thread scheduler.

iii) Running: After thread scheduler has selected it, then thread is in running state.

iv) Non-Runnable : Thread alive, not eligible to run.

v) Terminated : Thread is dead.

Difference between Thread.start() and Thread.run() :

→ When a program calls the 'start()' method, a new thread is created and then the 'run()' method is executed.

→ But if we directly call the 'run()' method, then no new thread will be created and 'run()' method will be executed as a normal method call on the current calling thread itself & no multithreading will take place.

Thread.sleep() method in java :

→ There are two overloaded methods of sleep() method in Thread class, one is with one argument & another is with two arguments.

→ The sleep() method is used to stop the execution of the current thread for a specific duration of time & after that time, it starts getting executed again.

Native →
```
public static void sleep(long millis) {
} // Implemented in another programming language.
```

Non-Native →
```
public static void sleep(long millis, int nanos) {
} // Implemented in java.
```

Deadlocking in java MultiThreading:

→ Synchronized keyword is used to make the class or method thread-safe which means only one thread can have the lock of synchronized method and use it.

→ Other threads have to wait till the lock releases.

Suppose there is a method which is used by multiple threads at the same time.

```
Void bookTickets() {
        tickets--;
}
```

When multiple threads try to execute the above method simultaneously, they try to modify tickets variable at the same time.

Example Scenario with Synchronization:

- Initial tickets = 10.
- Thread A & B both read tickets = 10.
- Thread A decrements tickets to 9.
- Thread B which read tickets in the same time reads 10 & decrements to 9.
- Output shows 9 available tickets even though two tickets are booked.

```
Synchronized void bookTickets() {
        tickets--;
}
```

Now, only single thread utilizes the resource tickets at any given point of time and output is consistent.

Daemon Threads in java:

→ Daemon threads are 'low-priority threads' in java that run in the background and perform tasks such as 'garbage collection'.

→ These threads provide service to user threads. As soon as the user thread completes the execution, the JVM exits, also terminating all daemon threads.

→ By default, the main thread is always a non-daemon thread.

Java Strings :

→ String is the type of objects that can store the sequence of characters enclosed by double quotes.

→ Simply said, String is a class that can also be used as datatype.

```
String str1 = "Pudhvi";
```

When creating a string, we are actually creating an object.

→ The above string can be done in another way.

```
String str1 = new String("Pudhvi");
```

→ There are two ways of creating a string. First one is a String literal and second one is using new keyword.

→ Lets first understand String pool before knowing actual difference.

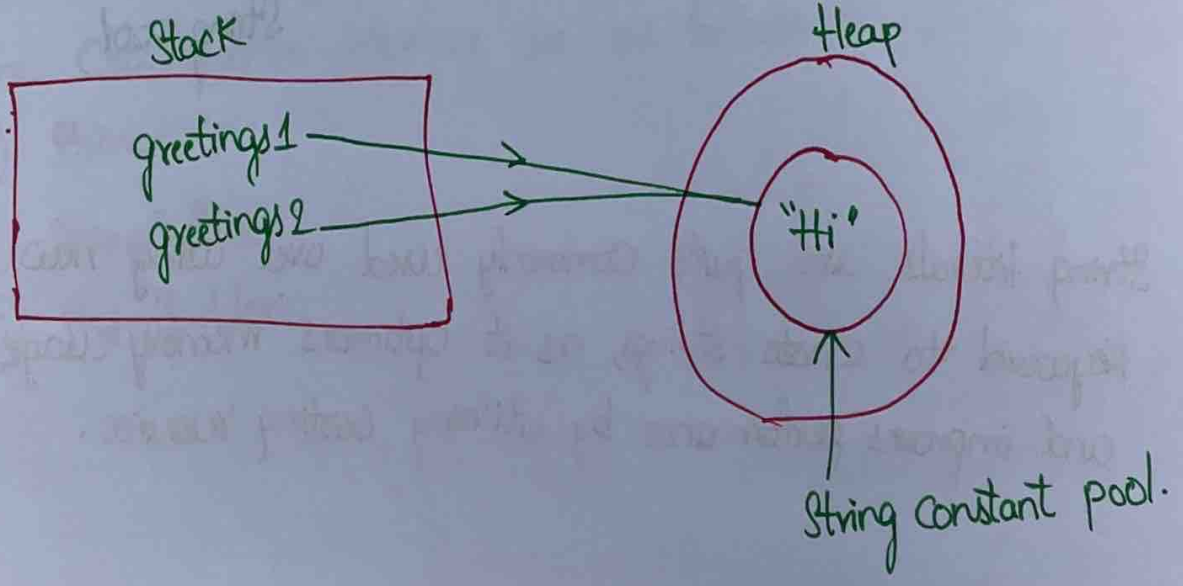
String pool (also called String intern pool) is a special memory in java's heap memory. It is designed specifically for storing unique strings.

```
1  String greeting1 = "Hi";
2  String greeting2 = "Hi";
3  String greeting3 = new String("Hi");
```

i)
→ When using string literals (line 1 & 2), JVM first looks in the string pool to see if an identical string exists. If it's found, java reuses that same string and returns a reference to it. If not, it creates a new string in the pool.

→ When line 1 is executed, greeting1 object is created in the stack; and the value "Hi" is created and stored in string constant pool.

→ When line 2 is executed, greeting2 object is created in the stack, but another instance of value "Hi" is not created. Instead, previous instance of "Hi" is reused.

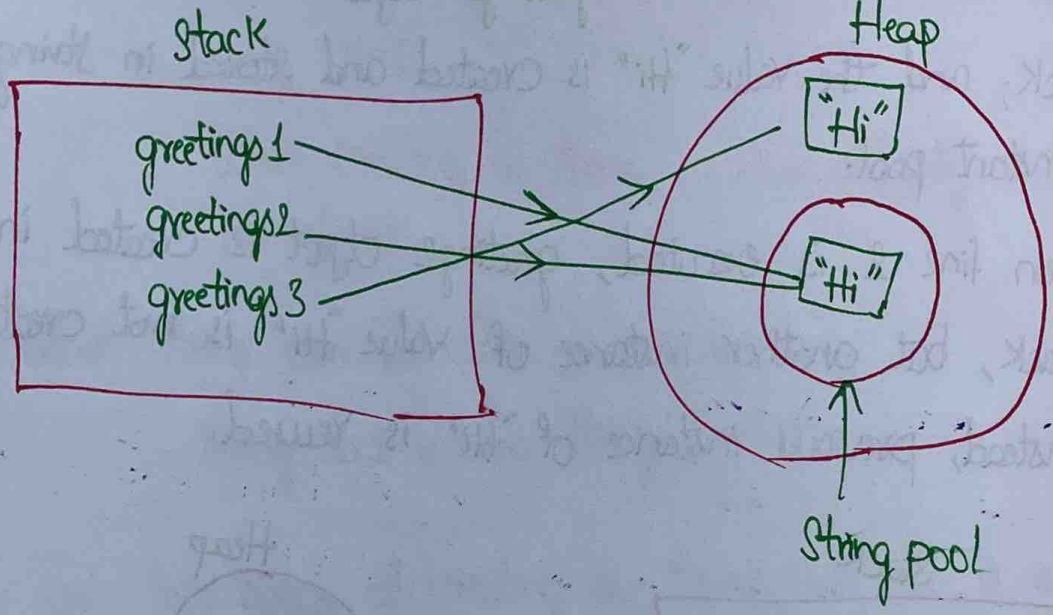


→ This way, if a similar value needs to be accessed again, a new string object created in the stack can reference it directly with the help of a pointer.

ii)

→ when using the new keyword, java creates a new object in the regular heap memory. It doesn't check the string pool at all.

→ A seperate copy is created, even if an identical string exists in the pool.



→ String literals are quite commonly used over using new keyword to create strings, as it optimises memory usage and improves performance by utilising existing resource.

Interfaces and classes in Strings :

String : It is a sequence of characters. In java, objects of String are immutable which means a constant and cannot be changed once created.

CharBuffer : This class implements the CharSequence interface. This class is used to allow character buffers to be used in place of CharSequences.

CharSequence Interface :

→ CharSequence interface is used for representing the sequence of characters in java. Classes that are implemented using the Charsequence interface are as follows:

i) String
ii) StringBuffer
iii) StringBuilder

1. String :-

→ String is an immutable class which means a constant and cannot be changed once created.

→ If we wish to change, we should create a new object. It is automatically thread safe.

→ Even the string methods like toUpperCase(), toLowerCase(), concat(), all these return a new object & doesn't modify original object.

"WHY ARE JAVA STRINGS IMMUTABLE ?

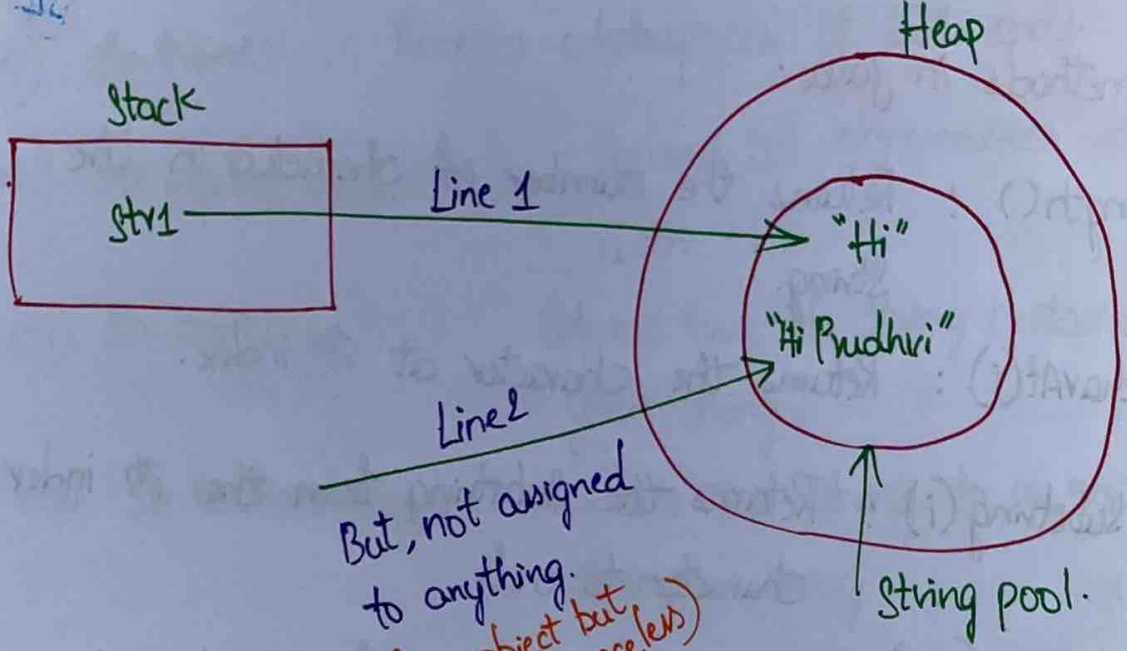
→ Java stores string literals in a pool to save memory. Immutability ensures one reference doesn't change the value for others pointing to the same string.

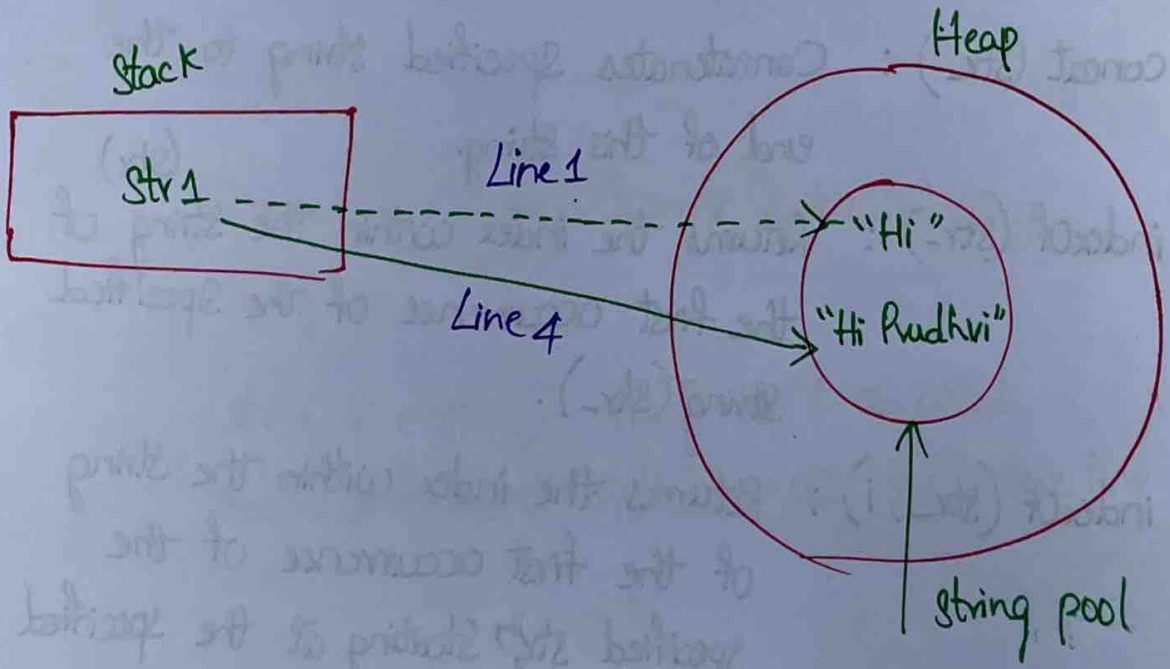
```
1  String str1 = "Hi";
2  str1.concat (" Pudhvi");
3  System.out.println (str1);
4  str1 = str1.concat (" Pudhvi");
5  System.out.println (str1);
```

Output:

Hi

Hi Pudhvi





→ After the execution of line 4, the object Str1 no longer points to "Hi" and points to "Hi Pudhvi".

→ This is the working of String immutability.

☆ String methods in java:

i) str.length() : Returns the number of characters in the String.

ii) str.charAt(i) : Returns the character at i^{th} index.

iii) str.substring(i) : Returns the substring from the i^{th} index character to end.

iv) str.substring(i,j) : Returns the substring from i to j-1 index.

v) str.concat(str_) : Concatenates specified string to the end of this string. (str)

vi) str.indexOf(str_) : Returns the index within the string of the first occurrence of the specified string (str_).

vii) str indexOf(str_, i) : Returns the index within the string of the first occurrence of the specified str, starting at the specified index i.

viii) str.equals(str_) : Compares this string to the specified object.

ix) str.toUpperCase() : Converts all characters in the string to uppercase.

x) str.toLowerCase() : Converts all characters to lowercase.

xi) str.trim() : Removes whitespaces at both ends.

xii) str.replace('ch1','ch2') : Replaces all occurrences of ch1 with ch2.

xiii) str.contains(str_) : Returns true if string contains the given string.

xiv) str.toCharArray() : Converts this string to a new character array.

∴

```
Scanner sc = new Scanner(System.in);
String str;
str = sc.next();     // To input a word.
str = sc.nextLine(); // To input the entire line.
```

2) StringBuffer :-

→ StringBuffer is a class in java that represents a mutable sequence of characters. It provides an alternative to the immutable string class, allowing you to modify the contents of a string without creating a new object every time.

```
StringBuffer s = new StringBuffer();
s.append("Hello ");
s.append("Pudhvi");
System.out.println(s); // implicitly calls toString() method on s.
```

→ StringBuffer is synchronized, making it thread safe.

→ StringBuffer is a peer class of string which provides more functionality than Strings.

→ As it is thread safe, it is mainly used for multi threaded programs.

3) StringBuilder :-

→ StringBuilder in java represents an alternative to String and StringBuffer class, as it creates a mutable sequence of characters & not thread safe.

→ Mainly used for single threaded programs.

→ StringBuilder is faster than StringBuffer in most implementations. Also, it performs better than StringBuffer.

→ As it doesn't guarantee synchronization, it is limited to single threaded programs.

String vs StringBuilder vs StringBuffer

Feature	String	StringBuilder	StringBuffer
Introduction	Introduced in JDK 1.0	JDK 1.5	JDK 1.0
Mutability	Immutable	Mutable	Mutable
Thread Safety	Thread Safe	Not Thread Safe	Thread Safe
Memory Efficiency	High	Efficient	Less efficient
Performance	High (No-Synchronization)	High	Low (Due to Synchronization)
Usage	When immutability is required	When Thread Safety is not required	When thread Safety is required

Collections Framework :

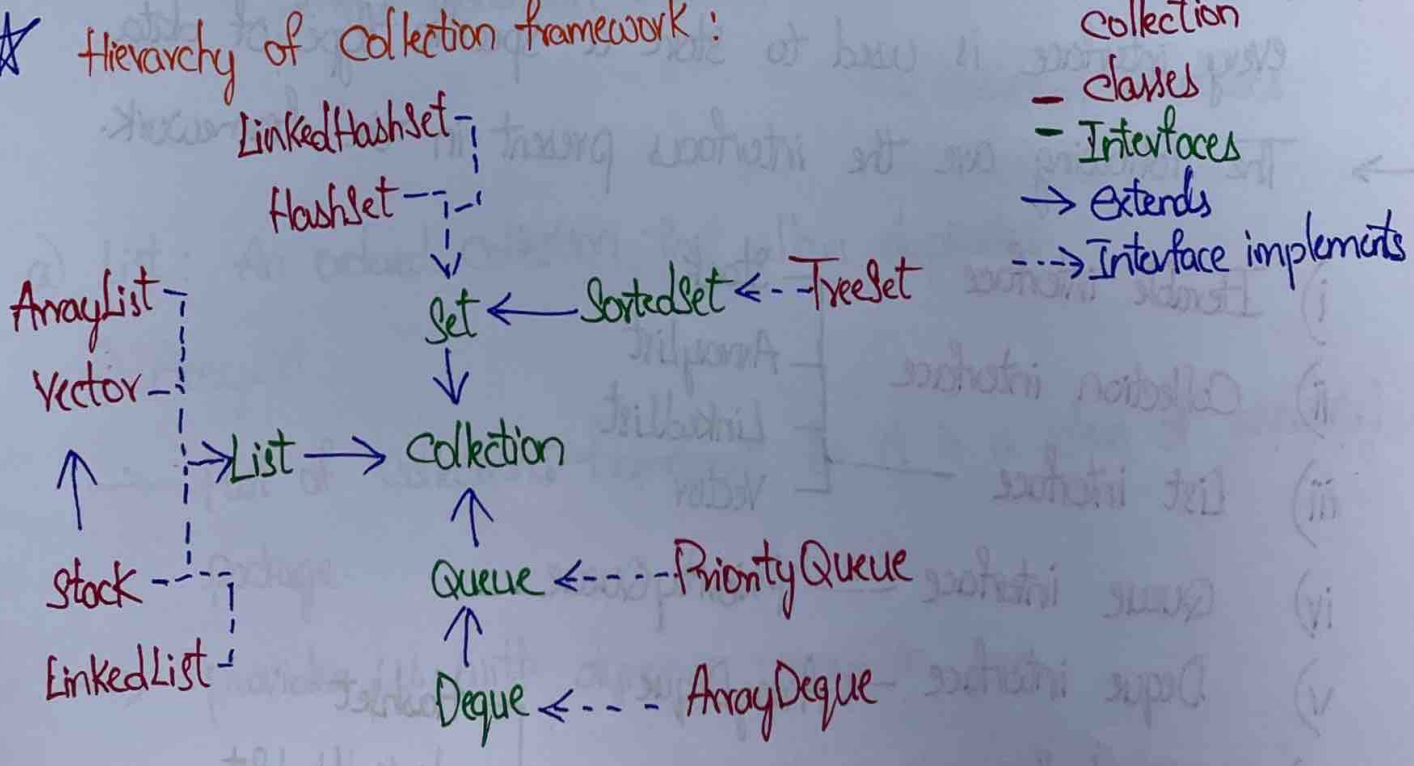
→ The java collections framework is a set of classes and interfaces that provide a standardized way to work with group of objects.

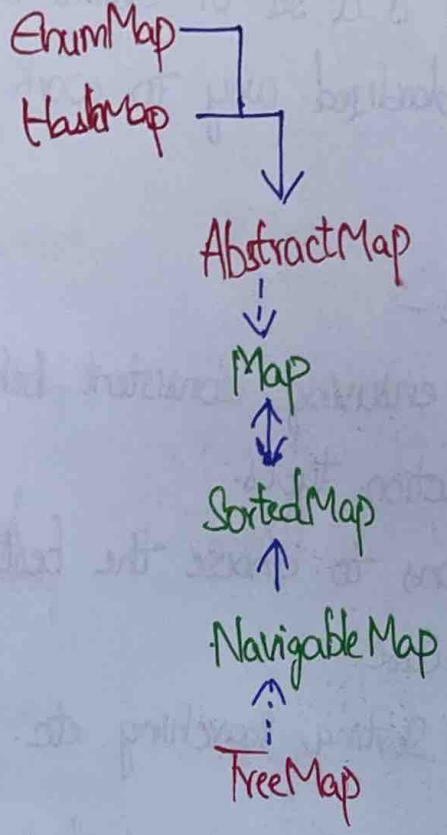
→ The collection framework provides -

- Standardized interfaces ensuring consistent behaviour across different collection types.
- Variety of implementations to choose the best data structure for the use case.
- Useful algorithms like Sorting, Searching etc.

→ The utility package contains all the classes & interfaces that are required by the collection framework. (java.util)

Hierarchy of Collection framework :





Interfaces that extend the Collection interface:

→ The collection framework contains multiple interfaces where every interface is used to store a specific type of data.

→ The following are the interfaces present in the framework.

i) Iterable interface
ii) Collection interface
iii) List interface — Stack, ArrayList, LinkedList, Vector
iv) Queue interface — PriorityQueue
v) Deque interface — ArrayDeque
vi) Set interface — HashSet, LinkedHashSet
vii) SortedSet interface — TreeSet
viii) Map interface — HashMap, TreeMap

✱ Collections class in java:

→ Collections is a utility class from java.util package. It defines several utility methods like Sorting & Searching; which are used to operate on collection.

→ It has all static methods.

✱ Collections Interface in java:

→ It is one of the root interfaces of the java Collection hierarchy.

→ It is not directly implemented by any class. Instead, it is implemented indirectly through it's Sub-interfaces. (List, Queue, Set)

→ For eg: ArrayList class implements List interface, a Sub-interface of the collection interface.

- Collection interfaces
- Collections classes

a) List: An ordered collection that allows duplicates.

i) ArrayList:

→ Part of Collections framework & it is a class of java.util package.

→ provides us with dynamic arrays in java.

```
ArrayList<Integer> arr = new ArrayList<Integer>();
```

→ ArrayList cannot be used for primitive types like int, char etc. We need a wrapper class for such cases.

→ It can store null values, which is useful to represent absence of a value.

→ preserves the order of elements.

```
import java.util.ArrayList;
public class ArrayList1 {
        psvm(String[] args) {
                ArrayList<String> arr = new ArrayList<>();
                arr.add("Prudhvi");
                arr.add("Komal");
                arr.add("Varnam");
                arr.add("Pavan");
                for (String str : arr) {
                        print(str);
                }
        }
}
```

ii) LinkedList :

→ Implementation of LinkedList data structure which is a linear data structure where the elements are not stored in contigous locations.

→ Every element is seperate object with a data part and address part.

→ The elements are linked using pointers & addreuses and each element is known as Node.

→ Internally, it is implemented using the doubly linked list data structure.

```
LinkedList<String> list1 = new LinkedList<>();
```

b) Set : An unordered collection that doesn't allow duplicate elements.

i) HashSet :

→ It is used to store the unique elements and it doesn't maintain any specific order of elements.

→ Uses Hashmap (implementation of hash table data structure) internally.

```
import java.util.HashSet;
public class HashSet1 {
    public static void main (String[] args) {
        HashSet <String> colors = new HashSet<>();
        Colors.add("Red");
        colors.add ("Blue");
        colors.add ("Black");
        colors.set (1, "Green");
        print (colors); // order might vary everytime.
    }
}
```

ii) TreeSet :

→ One of the most important implementations of the SortedSet interface. (Red-Black tree implementation)

→ Doesn't allow duplicate elements.

→ Doesn't allow null values and throws NullPointerException.

→ Ordering of elements is maintained by a set.

```
TreeSet<Integer> t = new TreeSet<>();
```

→ Implements NavigableSet which extends SortedSet which extends Set.

```
import java.util.TreeSet;
public class TreeSet1 {
    public static void main (String[] args) {
        TreeSet<Integer> t = new TreeSet<>();
        t.add(1);
        t.add(3);
        t.add(2);
        print(t);
    }
}
```

prints [1, 2, 3]

iii) LinkedHashSet :

→ It combines the functionality of a HashSet with a LinkedList to maintain the insertion order of elements.

→ stores unique elements only, maintains insertion order, provides faster iteration compared to HashSet and allows Null elements.

```
LinkedHashSet<String> set = new LinkedHashSet<>();
```

c) Queue : follows First-in, First-out principle.

i) PriorityQueue :

→ When elements of the queue are needed to be processed according to priority, then PriorityQueue is used.

→ PriorityQueue is based on the Priority Heap.

```
priorityQueue <Integer> q = new PriorityQueue<>();
```

d) Map : Represents a mapping between a key and a value.

i) HashMap :

→ provides basic implementation of Map interface.

→ Stores data in (Key, Value) pairs.

→ Access value by using corresponding key.

→ Duplicate elements are not allowed.

→ Order of insertion isn't preserved.

```
HashMap<Integer, String> map = new HashMap<>();
```

```
import java.util.HashMap;
public class HashMap1 {
    public static void main (String[] args) {
        HashMap<Integer, String> map =
            new HashMap<>();
```

```
        map.put (1, "one");
        map.put (2, "Two");
        print (map);  // Order might vary.
    }
}
```

ii) LinkedHashMap :

→ Same as HashMap, additional feature is that it maintains insertion order.

```
LinkedHashMap<Integer, String> map = new LinkedHashMap<>();
```

e) Deque : Double ended queue.

i) ArrayDeque :

→ Allows users to add or remove an element from both sides of the queue.

```
ArrayDeque<Integer> arr = new ArrayDeque<>();
```