(ii) By hiding class B within class A, A's members can be declared private and B can access them in hidden from outside world.

MST-2

Pakage and Interface

Package: A package in java is a namespace that groups related classes and interfaces. It helps in avoiding naming conflicts by providing unique namespaces and makes it easier to manage and structure a large codebase.

Types of packages:

(i) Built-in package: These pacakages consist of a large no of classes which are part of Java API:

(i) Java.lang (ii) Java.io (3) Java.util (4) Java.net

(ii) User defined packages: These are the packages that are defined by the user. first we create a directory mypackage, ~~then create file~~ directory name and package name should be same and then create a file in directory and create class inside file with the first statement → package name

```
package mypackage;
public class myclass
{  public void getName(String s)
   {
      SOP(s);
   }
}
```

Now we can use the package class myclass in our program. (CS-1)

```
import mypackage.myclass
public class printName
{ public static void main(String args[])
  { String name = "Raufham99";
    myclass obj = new myclass();
    obj.getName(name);
  }}
```

output : Raufham

Accessibility of Access Modifiers in Java

Access Modifiers	Accessible by classes in same packages	Accessible by classes in other the same package	Accessible by Sub classes in the Same package	Accessible by subclasses in other package
public	Yes	Yes	Yes	Yes
Protected	Yes	NO	Yes	Yes
default	Yes	NO	Yes	NO
private	NO	NO	NO	NO

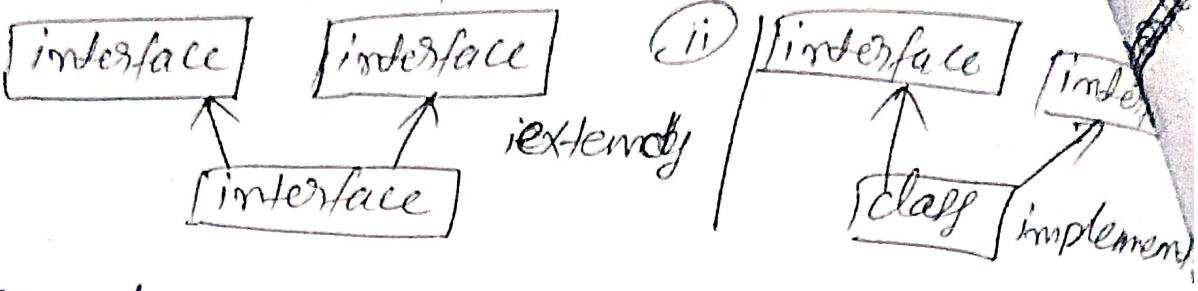
* Interface: Interface is just like a class, which contains only abstract method. To achieve interface, java provides a keyword called implements. Interface methods are by default public & abstract & interface variables are bydefault public + static + final. methods must be overriden inside implementing class.

Methods => public + abstract
variables => public + static + final

* Implementing an interface.

```
- Interface Animal
{  void sound(); // public + abstract
   void eat();
}
class Dog implements Animals
   @Override
   public void sound()
   { sop("Dog barking");
   }@Override
public void eat(){
   sop("Dog eats");
}}
public class Main
public static void main(String args[])
{  Dog dog = new dog();
      dog.sound();
      dog.eat();
}}      output: // dog barking
                // Dog eats
```

* Extending interface / Multiple inheritance




```
interface A
{
   void A();
}
interface B ~~extends A~~ {
   void B();
}
interface C extends A,B{
{  void C();
}
class D implements C{
   void A()
{  sop("A interface");
}void B()
{ sop("B interface");
}void C()
{ sop("C interface");
}
}
public class main{
public static void main(String[] args)
{
   D d = new D();
      d.A(); // A interface
      d.B(); // B   "
      d.C(); // C   "
}}
```

Nesting Interface:

```
interface outerinterface {
    interface innerInterface {
        void show();
    }
}
public class NestedClass implements
    outerinterface.innerInterface {
    public void show()
    { sop("Nested interface");
    }
    public static void main(String[] args) {
outerinterface.innerInterface obj = new NestedClass();
        obj.show();
    }
}
```

Output !! => Nested interface

** Default interface methods: Before JDK 1.8, interface can only have abstract methods and all the abstract methods of interfaces must be overriden in implementing class as well as methods are public & abstract by default.

```
interface A {
    void a();
}
class B imple A
{ pblic void a()
{ sop("B");
}
}
```

```
class C impl A
{ pblic void a()
{ sop("C");
}
}
```

```
class D imple A
{ public void a()
{ sop("D");
}}
```

if in interface A if a new abstract method is being added in future then all the implementing classes have to override this new method this is a big problem for all implementing class to override this problem.

From JDK 1.8 onwards interface can have default & static methods.

```
interface A
{ void fun(); //
  //+
  default void show(){ }
  sop("I am default interface method after jdk 1.8V99");
};
class B implements A{
  @override
  public void fun()
  { sop("abcd");
  }
}
class C implements A{
  @override
  public void fun(){
  sop("def");
}}
class D implements A{
  @override
  public void fun(){
  sop("ghi");
  }}
```

```
public class main{
public static void main(String[] args)
{   B b = new B();
    b.fun();
    D d = new D();
    d.fun();
    d.disp(); d.show();
}
```

output: abcd
ghd
i am default interface method after Jdk 1.8v.

So by applying default keyword before function in interface class, not need implementing class to override the all methods in interface but provide → default void disp(){}; ← this should be there

(5) How interface differ from inheritance?

ans ⇒

Q2. (a) Discuss keywords: Try-Catch, Finally, Throw, Throws

(b) Diff between Exception and Error

* Exception handling:

*Exception: An Exception is unexpected/unwanted/ abnormal situation that occured at runtime called exception.

Exception handling: Exception handling is a programming concept that allows developers to manage and respond to runtime errors or unusual conditions in a controlled manner, without crashing the entire application. when an error occures, the program generates an "exception" - a signal that something unexpected happend. It enables programmers to detect these issues, resolve or log them and continue executing if possible.

Key components of Exception handling

(i) try block: This block contains code that might throw an exception. when the program emlounters an error within try block, it transfers control to the corresponding Catch block.

(ii) Catch block: Catch block is used to define how to handle specific types of exceptions. Each catch block can be designed to catch a particular exception type, and multiple catch blocks can be used to handle various exceptions differently.

(iii) finally block: An optional ~~block~~ block that execues after the try and catch block, regardless of whether an exception was occure or not. This is typically used for cleanup the resources, such as closing resources (file or network connections).

Throw statement: Used to explicitly generate an exception or an exception when a certain condition occurs. This allows developers to signal errors manually.

(i) try and catch:

```
public class TryCatchExample {
public static void main(String[] args)
{
    try {
        int result = 10/0;
    }
    catch (ArithmeticException e)
    {
        Sop("caught an exception: cannot divided by zero");
    }
}}
```

(ii) Multiple catch:

```
public class MultipleCatcheExample{
    public static void main(String[] args)
    {
        try{
            int num[] = {1, 2, 3};
            Sop(num[5]); //
            int ref = 10/0;
        }
        catche (ArithmeticException e)
        { System.out.println("cannot divided by zero");
        }
        catch (ArrayIndexOutOfBoundException e)
    { Sop("Array index out of bound:")
    }
    catch (Exception e) {
        Sop("caught a general exception:" + e.getMessage());
    }}}
```

(ii) Nested try statement :

```
public class NestedTryExample {
  public static void main(String[] args)
  { try {
      int num[] = {1,2,5};
      System.out.println("outer try block");
    try { int res = 10/0;
    }
    catch( ArithematicException e)
    { Sop("Inner catch: cannot divided by zero :");
    }
    System.out.println("Inner catch: cannot divide by
        zero");
    System.out.println(num[5]);
    }catch (ArrayIndexOutOfBoundException e)
  { Sop("outer) catch : Invalid idx array");
  }
  }
```

* throw : used to throw an exception manually.

```
public class ThrowExample {
 public static void main(String[] args)
 { try { checkAge(15);
 }catch(IllegalArgumentException e){
   Sop("Exception caught" + e.getMessage());
 }
public static void chekAge(int age)
 { if(age < 18) {
```

throw new IllegalArgumentException("Age must be 18 or older");

}else{
sop("Age is valid");

⑤ finally: The finally block is used to clean up resources such as closing file or database connection:

```
import java.util.InputMismatchException;
import java.util.Scanner;  class A{
public static void main(String[] args)
{ Scanner sc = new Scanner(System.in);
  try{ sop("Enter an integer: ");
       int n = sc.nextInt();
       sop("You entered: "+n);
  }catch(InputMismatchException e)
  {sop("Invalid input. plz enter an integer;");
  }
  finally{
       sc.close();
       System.out.println("Scanner closed");
  }
}}
```

* Exception Types: (i) checked Exception (Compile-Time exception)
(ii) unchecked Exception
(iii) Errors:

Built-in-Exceptions

1. Checked Exception (Built-in-Exception): Checked exceptions are exceptions that must be either caught or declared in the method signature using 'throws'. The compile checks for these exceptions, and if they are not handled, it will result in a compile time error.

example: IOException (file handling)
SQLException (database access issue)
ClassNotFoundException

(2) Unchecked Exceptions (Runtime Exceptions): Unchecked exceptions are exceptions that occure at runtime and are not checked by the compiler. They are subclasses of RuntimeException, and they do not need to be declared in the method signature or catch in a try-catch block.

Example: NullPointerException
ArrayIndexOutOfBoundException
ArithmaticException

(3) Errors: Errors are serious problems that typically indicate a failure in the system's operations. They are usually beyond the control of the application.

Ex → OutOfMemoryError (when JVM runs out of memory)
StackOverflowError (infinite recursion)
VirtualMachineError (JVM issue)

* Uncaught Exception: An uncaught exception occurs when an exception is thrown but not handled by a try-catch block. Uncaught exceptions generally lead to program termination because java doesn't know how to handle them at runtime

common uncaught exceptions

(i) NullPointerException

(ii) ArrayIndexOutOfBoundException

(iii) ArithemeticException

* Handling uncaught Exceptions

using Try-catch Block

```
try {
    int ref = 10/0;
}
catch (ArithemeticException e)
{ SOP("Cannot divided by zero" + e.getMessage()); }
```

* Creating your own Exception:

```
class InvalidAgeException extends Exception
{ InvalidAgeException(String msg)
  { System.out.println(msg); }
}
class test
{ public static void main(String[] args)
  { try {
        vote(20)
    }
    catch(Exception e)
    { System.out.println(e); }
  }
  public static void vote(int age) throws InvalidAgeException
  { if(age<18)
    { throw new InvalidAgeException("Not eligible for vote"); }
    else { System.out.println("Eligible for vote"); }
  }
}
```

Difference between Error and Exception

Error	Exception
(i) An errors are not caused by Program.	An exception is caused by Program.
(ii) A lack of system resources typically causes errors in a program.	An exception occurs mainly due to issues in program.
(iii) Example : OutOfMemoryError	SQLException
(iv) Recovery from Error is not possible ;	we can recover from exceptions by using try catch block
(v) All errors in Java are unchecked type	Exceptions include both checked as well as unchecked type. unchecked exceptions occurs at runtime. and checked exception occure at compile-time
(vi) Errors can occure at compile time.	It is defined in java.lang.Exception
(vii) It is defined in java.lang.Error	caused by application logic, such as invalid input or file not found, array out of bound.
(viii) cause : typically caused by the environmental like : hardware failure or JVM issues	

Difference between throw and throws

	throw	throws
Use	① used to throw exception explicitly.	used in method signature to declare that a method can throw exceptions.
Usage	② used in inside a method	used in the method signature
Followed by	③ throw is followed by an object.	throws is followed by class
throw	④ we can throw only one exception at a time	we can handle multiple exceptions using throws keyword at a time
Example	⑤ throw new NullPointerException();	public void myM() throws IOException { ... }
propagation	checked exception cannot be propagated using throw only	checked exception can be propagated with throws.

Interface differ from inheritance

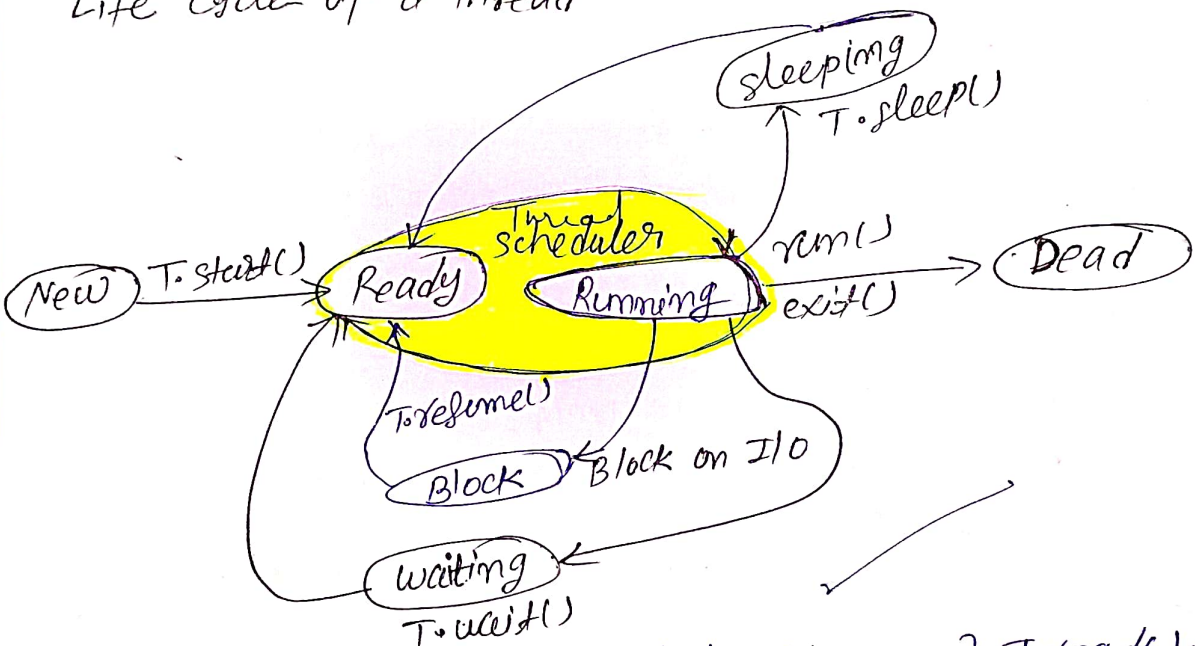
ire	inheritance	interface
urpose	Creates a hierarchy of classes and promotes code reusability	Difines a contract for classes to implement specific methods
No of superclasses	A class can inherit from only one superclass (single inheritance)	A class can implement multiple ~~and~~ interface.
Method Implementation.	Methods can have implementations in the superclass.	Methods are declared without implementation, implementing class provides implementation.
Relationship	Establishes an "is-a" relationship (e.g, car is a vehicle.)	Establishes a "can-do" relationship (eg, a class can "do" what the interface specifies).
Polymorphism	Supports polymorphism through superclass.	Support polymorphism by allowing different classes to implement the same interface.
Use cases	Used when creating a new class that shares common features with an existing class	Used for ensuring multiple classes adhere to a common contract or when multiple inheritance is needed.

1. 3.

What are the different states in life cycle of thread java? List the common methods used in thread agement?

⇒ Threads: In Java, a thread is a lightweight process or small piece of process that represents a seprate path of execution within a program as a single flow of control that can run concurrently with other threads. This allows for multitasking, where multiple tasks can be executed simultaneously.

Life cycle of a thread



1. New: when a thread is created using new Thread(), it is in the new state, The thread is not yet started.

2. Ready (Runable): After calling the start() method, the thread moves to the runnable state, It is ready to run but may be waiting for the CPU to allocate time for its execution.

3. Running: In this state, the thread is executing its task. The thread scheduler allocates CPU time to the runable thread moving it to the running state.

④ Blocked: A thread enters the blocked state if it is waiting for a monitor lock (e.g when a method is used) that is held by another thread.

⑤ Waiting: A thread enters the waiting state when it waits indefinitely for another thread to perform a specific action (e.g using wait(), join() without a timeout). It moves back to ready state when another thread notifies it.

⑥ Timed waiting: This is a variant of the waiting state. A thread enters timed waiting when it waits for a specific period (using sleep(long millis), wait(long timeout), join(long millis) etc).

⑦ Terminated (Dead): ~~When a thread completes its execution~~

A thread terminates because of either of the following reasons.

(i) it exits normally: This is happens when the code of the thread has been entirely executed by the program.

(ii) Because there occured some unusual error even, like a segmentation fault or an unhandled exception.

* Common methods for Thread Management

(i) Start(): Starts the thread, moving it from new state to ready state / runnable state

(ii) run(): Defines the code that executes the task of the thread. It is usually overriden in a Thread subclass or when implementing Runnable.

(iii) sleep(long millis): puts the thread to sleep for the specified time (in milliseconds) moving it to the timed waiting state

wait(): Causes the current thread to wait until another thread calls notify().

(v) notify() and notifyAll(): Used to wake up threads that are waiting on an objects monitor. notify() wakes up one random waiting thread, while notifyAll() wakes up all waiting threads.

(vi) join(): waits for the specified thread to finish executing before proceeding. it can be used with or without a timeout.

(vii) yield(): A hint to the thread scheduler that the current thread is willing to yield its current use of a processor and let the another thread execute first.

(viii) interrupt(): Interrupts a thread, which may stop it or change its state depending on how it handles the interrupt tion.

(ix) isAlive(): checks if the thread is alive.

Q4. Explain why Applets were once popular for web-based applications and how their disadvantages have led to their decline?

Ans ⇒ Applets, small Java programs embedded in web pages, were once popular in the late 1990s and early 2000s for adding interactive and dynamic features to web applications. They allowed developers to create rich, interactive content, such as animations, games, data visualizations, and other complex functionalities that weren't possible with early HTML and Javascript capabilities. Applets were attractive because Java's platform independently allowed them to run on various operating systems

as long as a Java virtual Machine (JVM) was available in the user's browser.
However, several disadvantages led to the decline of Applets:

(1) Security concerns: Applets introduced significant security vulnerabilities. They required extensive permissions to access system resources, which posed risks to users, as Applets could be exploited for malicious purposes.

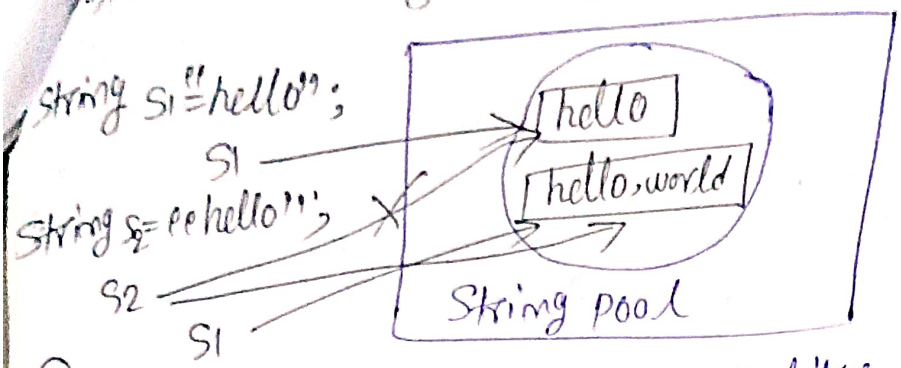
(2) Compatibility and Maintenance: Running applets required a compatible JVM in the browser. This dependency made compatibility inconsistent across different browsers and operating systems.

(3) Performance Issues: Applets could be slow to load and consume significant resources, especially on systems with limited processing power.

(4) Advancements in Web Technologies: The rise of HTML5, CSS3, JavaScript and frameworks like React and Angular introduced new, secure and efficient ways to achieve interactivity and multimedia functionality directly in the browser.

(5) Mobile Device Limitations: Java Applets were largely incompatible with mobile devices especially as smartphones and tablets gained popularity.

why String objects are immutable? what are string buffer and string builder in java



immutable → cannot change

① When we create a string in java like s1 = 'hello'; then an object will be created in string pool (hello) and s1 will be pointing to hello; Now if again we do String s2 = 'hello' then another object will not be created, but s2 will point to 'hello' because JVM will first check if the same object is present in string pool or not, if not present, then only a new one is created else not.
Now if suppose Java allow string mutable then if we change s1 to 'hello world' then s2 value will also be 'hello world' unintentionally, leading to unpredictable bugs and violations of data integrity.

② Thread safety: String immutability makes it inherently thread-safe because there is no risk of strings being modified by one thread while being read by another.

③ Security: String is widely used in Java for handling sensitive information like passwords, usernames, database URLs.

④ Efficiency in Hashing: Immutability makes String suitable as a key in hash-based collections like HashMap or HashSet.

StringBuffer and StringBuilder are mutable classes in Java that allow you to modify strings without creating new objects.

StringBuffer: It is a thread-safe, mutable sequence of characters. StringBuffer is synchronized, which makes it slower than StringBuilder in single-threaded scenarios but ensures safe access in multi-threaded environments.

Ex → StringBuffer sb = new StringBuffer("Hello");
sb.append("world");
Sop(sb) // Hello world

StringBuilder: Introduced in Java 5, StringBuilder is similar to StringBuffer but is not synchronized meaning it's faster in single-threaded scenarios, However, it is not thread-safe.

Ex ⇒ StringBuilder sb = new StringBuilder("Hello");
sb.append("world");
Sop(sb); // op: Hello world

Q6. what is event handling in Java? List the key components in event handling.

ans ⇒ Event handling in java is a programming mechanism that allows a program to respond to user actions or system generated events, such as mouse clicks, key press or window resizing. This is a key aspect of Java's GUI programming, enabling interactive applications

List components of Event handling in java

Event source: This is the component that generates the event. for examples, a button, a ~~field~~ field, or a window can acts of an event source.

(2) Event object: when an event occurs, an event object is created, which contains information about the event, such as the type of event, the ~~source~~ source of the event, and any relevant data like the coordinates of a mouse click.

(3) Event Listener: This is an interface that defines the methods for handling specific types of events. The class that implements this interface is responsible for processing the event when it occurs. common event listeners

(i) ActionListener (ii) MouseListener (iii) KeyListener

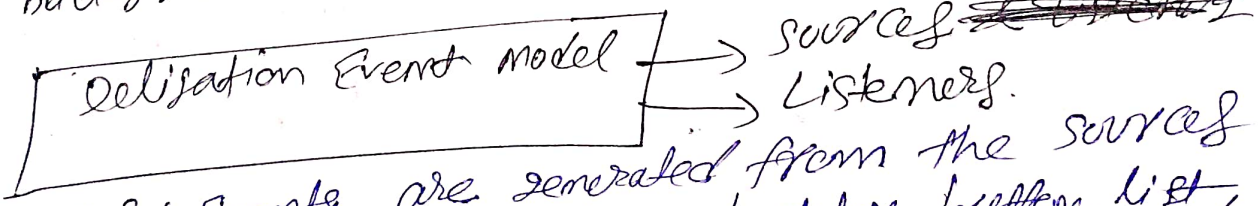
(4) Event Registration: The process of registering an event listener with an event source. This involves associating the listener with the source so that when the event occurs, the listener method is invoked. Eg → addActionListener()

(5) Event Dispatching: This is the process by which the JCVA runtime system involves the appropriate event listener methods when an event occurs.

Event handling : change in the state of object is known as event, events are generated as a result of user interacting with the gui component Eg: (i) clicking a button
(ii) Moving the mouse
(iii) Entering a character through keyboard
(iv) Selecting an item from
(v) scrolling the page
These are the activities causes to event occure

Event handling : It is a mechanism that control the event and decide what should happen if an event occurs. This mechanism has a code which is known as event handler. Java uses delivating event model to handle the events this model defines the standard mechanism to generate and handle the event.

Types of Event
(i) foreground Events
(ii) background Events



sources : Events are generated from the sources
Ex → various sources like checkbox, button, list, button etc

Listeners : Listeners are use to handling the events generated from the source each of these listeners represents interfaces that are responsible for event

handling

* Event classes in Java :

Event class	Listener interface
~~Active Event~~	
ActionEvent	ActionListener
AdjustmentEvent	AdjustmentListener
ComponentEvent	ComponentListener
ContainerEvent	Container
FocusEvent	FocusListener
ItemEvent	ItemListener
KeyEvent	KeyListener
MouseEvent	MouseMotionListener MouseListener
TextEvent	TextListeners
WindowEvent	WindowListener

Abstract window Toolkit : package used

→ import java.AWT. *

→ java.awt.event.* → Action Event
Mouse "
Key "
EventListener
Specific listener Interface :
(i) Action Listener
(ii) Mouse Listener
(iii) Key "

* The main Thread in java: In java, the main thread is the primary thread that the java virtual machine (JVM) creates when a java program starts. It is the first thread to be executed and serves as the entry point for a java application. The main thread is created by the JVM and begins with the main method.

```
public class MainThreadExample
{ public static void main(String[] args)
  { System.out.println("This is the main thread.");
  }
}
```

Characteristics of the Main Thread

(i) Entry point: This main thread runs the main method, making it the starting point of every standalone java app.

(ii) Thread class: The main thread is a Thread object, it can be controlled like any other thread using methods from the Thread class, such as getName(), setPriority(), sleep() etc

(iii) Thread management: The main thread can create other threads and manage them. The program terminates only when the main thread and all other non-daemon threads finish executing.

* Controlling the Main thread

```
public class MainThreadControl {
  public static void main(String[] args)
  { Thread mainThread = Thread.currentThread();
    System.out.println("Main thread name:" + mainThread.getName());
    System.out.println("Main thread priority:" + mainThread.getPriority());
    mainThread.setName("primary Thread");
    mainThread.setPriority(Thread.MAX_PRIORITY);
```

```
System.out.println("Updated main thread name:" +
                    mainThread.getName());
System.out.println("updated main thread priority:" +
                    mainThread.getPriority());
}
}
```

Joining and sleeping: you can put the main thread to sleep or use the join() method on it, like any other thread.

```
public class MainThreadSleep {
  public static void main(String[] args){
  System.out.println("Main Thread starting.");
  try {
  Thread.sleep(2000); // pause the main thread for 2sec.
    Sop("Main thread woke up after 2 sec.");
} catch (InterruptedException e)
  {
   Sop("Main thread interrupted.");
  }
   Sop("Main thread ending.");
}
}
```

* There is two ways to create Thread
  (i) Thread (class) (java.lang)
  (ii) Runnable (interface)

* creating Thread : In java, there are two primary ways to create a thread. (i) Thread class (ii) using runnable interface.

(i) Thread class → Create a class that extends Thread. override and call start().

```
class Test extends Thread {
    @override
    public void run() {
    Sop("Thread is running");
} }
public class Main {
  public static void main(String[] args)
{   Test t1 = new Test();
      t1.start();     Op: Thread is running.
} }
```

(ii) using Runnable interface:

```
class Test implements Runnable {
   @override
   public void run()
{  Sop("Thread is running via Runnable:");
} }
public class main {
  public static void main(String[] args)
{  Test t1 = new Test();
   Thread th = new Thread(t1);
} } th.start();   // Op: Thread is running via Runnable.
```

* creating multiple thread via Thread class.

```
class Test1 extends Thread
{
  public void run()
  {
    Sop("thread 1:");
  }
}
class Test2 extends Thread
{
  public void run()
  {
    Sop("thread 2:");
  }
}
class Test3 extends Thread
{
  public void run()
  {
    Sop("thread 3:");
  }
}
public class Main
{
  public static void main(String[] args)
  {
    Test1 t1 = new Test1();
    Test2 t2 = new Test2();
    Test3 t3 = new Test3();
    t1.start();
    t2.start();
    t3.start();
  }
}
```

output : thread 2
thread 3
thread 1

Thread Methods

(1) Basic methods ⇒ run(), start(), currentThread(), isAlive()
(2) Naming methods ⇒ getName(), setName(String name)
(3) Daemon Methods ⇒ isDaemon(), setDaemon(boolean b)
(4) priority methods ⇒ getPriority(), setPriority(int p)
(5) Preventing Methods ⇒ sleep(), yeild(), join()
(6) interrupting Methods ⇒ interrupt(), isInterrupted(), interrupted()

Depreciated methods ⇒ suspend(), resume(), stop(), destroy()

* Inter thread communication ⇒ wait(), notify(), notifyAll()

* sleep(long millisecond) : pause execution

```
public class Test extends Thread{
    public void run(){
     try{
        for(int i=1; i<=5; i++)
       { sop(i);
         Thread.sleep(1000) // pause for 1 Second
       }}
       catche(InterruptedException e)
         { sop(e Thread Interrupted"");
       }}
       public static void main(String[] args)
      {   Test t1 = new Test();
          t1.start();
    }}
```

* getId() ⇒ Returns the ID of the thread

```
Public class Test extends Thread {
    public void run()
    { sop("Thread ID:" + Thread.currentThread().getId());
    }
    public static void main(String[] args)
    { Test t1 = new Test();
      t1.start();
    }}
```

* getName() and setName(String name):

```
public class Test extends Thread
{ public void run(){
  sop("Thread Name" + Thread.currentThread().getName());
}}
public static void main(String[] args)
  { Test t1 = new Test();
    t1.setName("Raghav");
    t1.start();
    sop("main thread name:" + Thread.currentThread().getName());
  }}
```

* getPriority() and setPriority(int p): MAX_PRIORITY → 10

```
public class Test extends Thread{
public void run(){
 sop("Thre prio:" + Thread.currentThread().getPriority());
}
public static void main(String[] args){
 Test t1 = new Test();
 t1.setPriority(Thread.MAX_PRIORITY);
 t1.start();
}}
```

MIN_PRIORITY ⇒ 1
NORM_PR " → 5
by default main → 5
o child thread priority of their main method if not provided

0 ≤ priority ≤ 10

t1.setPriority(3) // 4, 5, 6, 7 till 10

* isAlive(): public class Test extends Thread {

```
public void run(){
sop("running...");
}
```

```
public static void main(String[] args)
{ Test t1 = new Test();
  Sop("Before Starting:" + t1.isAlive());
    t1.start();
    Sop("After starting:" + t1.isAlive());
} }
```

* join: waits for a thread to complete and then resumes.

```
public class Test extends Thread
{ public void run(){
  for(int i=1; i<=5; i++)
  { Sop("Thread " + i);
    Thread.sleep(1000);
} public static void main(String[] args)
  { Test t1 = new Test();
    Test t2 = new Test();
    t1.start();
    try{ t1.join();
    }catch (InterruptedException e){
      Sop(e);
    }
    t2.start();
  }
```

* setDaemon(boolean on) and isDaemon(): set the thread as a daemon thread or checks if it's a daemon. Daemon thread run in the background and are terminated when all user threads finish.

```
Public class Test extends Thread{
  Public void run(){
  Sop("Daemon status:" + isDaemon());
```

```
if(Thread.currentThread().isDaemon())
{ sop("daemon thread");
}
else{ sop("child thread");
    }
}
Public static void main(String[] args)
{
    Test t = new Test();
    t.setDaemon(true);
    t.start();
}
}
```

* yield(): pauses the current thread to give other threads an opportunity to execute

```
public class Test extends Thread {
  public void run()
  {
    for(int i=1; i<=5; i++)
    {
      sop(Thread.currentThread().getName()+"_"+i);
      Thread.yield(); // Hints that this thread is willing to
                      to stop their execution to give
                        chance to another thread;
}}
Public static void main(String[] args)
{ Test t1=new Test();
  Test t2= new Test();
    t1.start();
  t2.start();
}
```

* interrupt(), isInterrupted() and interrupted():

interrupt(): This method is used to interrupt a thread. if the thread is in a sleep or wait.

isInterrupted(): This method return true if the thread has been interrupted, but it does not clear the interrupted status flag.

interrupted(): This static method checks if the current thread has been interrupted, and it clears the interrupted status flag.

Thread Interrupt only work when thread were to sleep or wait

```
class Test extends Thread
{ public void run()
  {// System.out.println(Thread.interrupted()); // true → false  ①
   // System.out.println(Thread.currentThread.isInterrupted()); // false  ②
      try { for(int i=1; i<=5; i++)
            { Sop(" "+i);
              Thread.sleep(1000); // 1sec
            }
          }
      catch (Exception e)
      { Sop("Thread interrupted: "+e); }
  }
public static void main(String[] args)
{  Test t = new Test();
     t.start();
   // t.interrupt();
}
```

① false → true

② true → false

File Handling : File handling in Java is done through the java.io package, which provides classes to create, read, write, and manipulate files. Here are some common classes and methods used in file handling.
for file handling we use 2 streams : (i) Byte (ii) Character.
* Stream is a sequence of data.
All the classes divided into 2 streams
* methods used in file handling are

(1) canRead() → return boolean value
(2) canWrite()
(3) createNewFile()
(4) exists()
(5) length()
(6) getName()
(7) getAbsolutePath()
(8) read()
(9) write()
(10) RenameTo()
(11) Delete()
(12) MkDir()
(13) List()

* file handling classes :
(1) File (2) File_Reader (3) File_Writer
(4) File_Input_Stream (5) File_Output_Stream
(6) Buffer_Input_Stream
(7) Buffer_Output_Stream.

```
import java.io.*;
class create_file {
    psvm {
    File f = new file("C:\Users\HP\Desktop\learn.txt");
    try{
        if (f.createNewFile()){
            Sop("File successfully created");
        }
```

```
else { sop("file already exists"); }
}
catch (IOException i)
{ sop("Exception Handling"); }
}}
```

// write file

⇒
```
import java.io.*;
class Filewriter {
psvm { try {
FileWriter f=new FileWriter("path");
try{ f.write("Java is programming lang");
}
finally{
f.close();
}
sop("succesfully data write in file");
}
catch(IoException i){
sop(i);
} finally { f.close(); }
}}
```

// Rename file

```
import java.io.*;
class Renamefile
{ psvm {
File f=new file("path of existing file with filename");
File g= new file("path of new file with filename");
if(f.exists())
{ sop(f.renameTo(g));
}else { sop("file doesn't exist"); }}}
```

Synchronization: It is a process that allows multiple threads to work together in a synchronized manner ensuring that they do not interfere with each other while accessing shared resources concurrently, when multiple threads access shared resources concurrently, there is a risk of data inconsistency or race condition, Synchronization helps to prevent these issues by controlling thread access. It is categorized in to two categories.

1. Method Level Synchronization.
2. Block Level Synchronization.
3. static synchronized method.

(i) Method Level synchronization: By making a method synchronized. you ensure that only one thread can execute at time for a given instance

(ii) Block Level synchronization: By making a block of code by synchronization keyword, only one thread can execute that block at a time for a given object

(iii) Static synchronization: If a static Method is synchronized the lock applies to the class not to the instances of the class.

Q diff String vs String buffer & String builder

parameters	String	String buffer	String builder
→ Storage: Heap obj, → memory		thread	use
		performance	

Synchronization problem

```
class BookTicket
{
    int total_Seat = 10;
    void bookSeat(int seats)
    {
        if(total_Seats >= seats)
        {
            sop("seats booked successfully");
            total_Seats = total_Seats - seats;
            sop("seats left" + total_Seats);
        }
        else { sop("seats cannot booked");
               sop("seats left" + total_Seats);
        }
    }
class MovieBookApp extends Thread {
    static BookTicket b;
        int seats;
        public void run()
        {
            b.bookSeat(seats);
        }
        public static void main(String[] args)
        {
            b = new BookTicket();
```

```
MovieBookApp U1 = new MovieBookApp();
    U1.seats = 7;
    U1.start();
MovieBookApp U2 = new MovieBookApp();
    U2.seats = 6;
    U2.start();
```

```
}
}
```

output is: 7 seats booked successfully
seats left: 3
seats booked successfully
seats left: -3

Solution of Synchronization issue

Method level Synchronization:

```
class BookTicket
{
  int total_Seats = 10;
Synchronized void bookSeat(int seats)
  {
    if(total_seats >= seats)
    { Sop("Seats booked succesfully");
      total_seats = total_seats - seats;
      Sop("seats left" + total_seats);
    }else{
      Sop("Seats cannot be booked");
      Sop("seats left" + total_seats);
    }
  }
class MovieBook extends Thread
{
  static BookTicket b;
    int seats;
    public void run()
    { b.bookSeat(seats);
    }
    Publis static void main(String[] args) b=new BookTicket();
    {
      MovieBook u1 = new MovieBook();
      u1.seats = 7;
      u1.Start();
      MovieBook u2 = new MovieBook();
      u2.seats = 6;
      u2.start();
    }
}
```

output: Seats Booked succesfully
Seats left : 3
Seats cannot be booked
Seats left : 3

Solution of synchronization issue

② Block level synchronization: same program as before but instead of making entire bookSeat() function synchronized we did important or only main source synchronized that may lead to confliction.

```
int total_Seat = 10;

void bookSeat(int Seats)
{ Sop("Thread is running");

  synchronized (this)
  { if( total_Seats >= Seats)
    { Sop("Seats booked successfully");
      total_Seats = total_Seat - Seats;
      Sop ("Seats left" + total_Seats);
    }else {
      Sop("Seat cannot be booked");
      Sop("Seats left" + total_Seats);
  } }
  Sop("Thread is running");
}
```

③ Static method synchronized:

```
class BookTicket
{ static int total_Seats = 10;
static synchronized void bookSeat(int Seats)
{ if(total_Seats ≥ Seats)
  { Sop("Seats booked successfully");
    total_Seats = total_Seats - Seats;
    Sop("Seats left" + total_Seats);
  } else {
```

same as before

| join()

Inter thread communication @ : Inter-thread communication used when threads need to cooperate with each other. This is commonly achieved using wait(), notify() and notifyAll() methods. These methods are used to synchronize threads. allowing one thread to wait until another thread performs a particular action.

```
class TotalEarnings extends Thread
{   int total=10;
    public void run()
    {  synchronized(this)
       {  for(int i=0;i<=10;i++)
          {  total = total +100;
          }
            this.notify();
            //this.notifyAll();
    }}
    public class MovieBook
{   public static void main(String[] args)
    {  TotalEarnings te = new TotalEarnings();
       te.start();
       synchronized(te)
       {  try{ te.wait();
            System.out("Total earining : "+te.total);
          }
          catch(InterruptedException e)
          {  System.out("main Thread Interrupted while waiting");
          }
    }
}
```

	String	StringBuffer	StringBuilder
Storage	Heap area, string constant pool.	Heap area	Heap area
objects	immutable object	mutable object	mutable object
Memory	If we change the value of string a lot of times, it will allocate more memory	consumes less memory	occupy less memory
Thread safe	Not thread safe	all methods are synchronized & thus it is thread safe	non-synchronized methods not thread safe
Performance	slow	fast as compared to String	faster than StringBuffer
Use	if data is not changing frequently	if data is changing frequently	if data is changing frequently.

By	Sleep()	Yield()	join()
purpose	if any thread does not want to perform any operation for perticular time	It stops the current executing thread & provide chance to another thread of same or heigher priority to execute	If a thread wants to wait for another thread to complete it's task
examples	① Times, PPT, blinking bulbs	shopping	Licence Dept
thread invokes	① automatically after provides time period ② if thread is interrupted.	① automatically invoked by thread-scheduler	① Automatically invokes after completion of another thread task. ② After completion of time period ② If thread is interrupted.
Methods	① sleep(long ms) ② sleep(long ms, int ns)	yield()	① join() ② join(long ms) ③ join(long ms, int ns)
Exception	Yes	NO	Yes
is method final	NO	No	yes
is method static	Yes	Yes	NO
is method native	① native ② Non-native	Yes	NO

* with the help of program demonstrate the use of four methods of StringBuffer class and StringBuilder.

```
Public class Test
{
  Public static void main(String[] args)
  {
    StringBuffer sb = new StringBuffer("Hello");
    // 1 Method → append("ABC");
    sb.append(" world");
    Sop(sb); // Hello world
    // 2 Method → reverse()
    sb.reverse();
    Sop(sb) // dlrow olleH
    // 3Meth → capacity()
    Sop(sb.capacity()); // 27
    // 4th → length()
    Sop(sb.length()); // 11
    // charAt()
    Sop(sb.charAt(1)); // l
    Sop("String Builder class Demo ----");
    StringBuilder SB = new StringBuilder("Hello");
    // 1 append()
    SB.append(" world");
    Sop(SB);
    SB.reverse(); // 2 M
    Sop(SB);
    Sop(SB.capacity()); // 3 M
    Sop(SB.length()); Sop(SB.charAt()); // 4 & 5 method.
```

Q. Implementing Runnable interface or extending Thread which method will you prefer for multithreading and why?

ans → prefer : Implementing the Runnable interface

(i) Flexibility : Java does not support multiple inheritance. So if you extend the Thread class, you cannot extend any other class. By implementing the Runnable interface, you can extend another class and achieve multiple inheritance.

(ii) separation of concerns : Implementing Runnable interface separates the task of what the thread should do from the actual Thread class.

(iii) Reusability : The same Runnable interface can be executed by another multiple threads, enhancing reusability.