Ex:- Application Rto

```
import java.util.Scanner;
Class Applicant
{
```

Example on Custome exception.

```
→ import java.util.Scanner;
  Class UnderAgeException extends Exception
  {
    public UnderAgeException(String message)
    {
      super(message);
    }
  }
class OverAgeException Extends Exception.
{
  public OverAgeException(String message)
  {
    super(message);
  }
}
class Applicant {
    int age;
    public void input()
    {
      Scanner scan = new Scanner(System.in);
      System.out.println("please enter your age");
      age = scan.nextInt();
    }
```

```
Void Verify() throws UnderAgeExp, OverAgeExp
{
 if (age <18)
  {
   UnderAge Exception uae = new UnderAgeException ("ohh du
    System.out.println (uae.getMessage());
    throw uae;
  }
 else if (age >60) {
   OverAge Exception oae = new OverageException
    System.out.println(oae.getMessage());
    throw oae;
  }
 else {
    System.out.println ("you are eligible");
  }
 }
}
class Rto {
  public void initiate () {
   Applicant a= new Applicant();
   try {
   }
   a.input();
   a.verify();
  }
 Catch (UnderAge Exception | OverAgeException e)
{
try {
 a.input();
 a.verify();
Catch ( UnderAgeExp | overAge Exp e)
{
```

```
{
  System.out.println("Don't ever try again");
  System.exit(0);
}
public class main()
    public static void main(String[] args){
      Rto r = new Rto();
        r.initiate();
    }
```

Day 4 :- Exception Handling

1. try with resource
2. try with multiple-catch block
3. Rules of Overriding associated with Exception

→ Remaining topics to be discussed.

1. Instanceof Vs isInstanceOf(object obj)
2. How to create a user defined package and in realtime project how it is used?

Resource - a inbuilt class for which we create object using a object you made a Calle to those method to perform some operation. is called resource bufferReader is a resource.

1.7 Version Enchancements

1. try with resource
2. try with multicatch block

until jdk.1.6, it is compulsory required to write finally block to close all the resources which are open as a part of try block.

eg:

```
BufferReader br=null
try{
 br =new BufferReader(new FileReader("abc.txt"));
}
Catch (IOException ie){
    ie.printStackTrace();
}
finally {
 try {
    if (br!=null){
        br.close();
    }
 }
 catch (IOException ie){
    ie.printStackTrace();
}}
```

problems in the above approach

1. Compulsory the programmer is required to close all opened resources which increases the complexity of the program.
2. Compulsory we should write finally block explicitly, which increases the length of the code and reviews readability. To overcome this problem sun ms introducted try with resources in "1.7" version of jdk.

try with resources

In this approach, the resources which are opened as a part of try block will be closed automatically once the control reaches to the end of try block normally or abnormally, so it is not required to close explicitly so the complexity of the program would be reduced.
It is not required to write finally block explicitly, so, length of the code would be reduced and readability is improved.

```
try ( BufferReader br = new BufferReader (new FileReader
      //use br and perform the ("abc.txt")){
    uncessary operation.
  //once the control reaches the end of try
  automatically br will be closed.
}catch (IOException ie){
```

// handling code
}

Rules of using try with resource

(1.) We can declare any no of resources, but all these resources should be seperated with eg:

```
try (R1; R2; R3;){
    //USE the resources
}
```

(2) All resources are said to be Auto closeable resourc if the class implements an interface called "java.lang.Auto closeable" either directly or indirectly

eg:: java.io package classes, java.sql.package classes.

```
public interface java.lang.Auto closable {
    public abstract void close() throws java.lang.
                                        Exception;
}
```

Note: which ever class has implemented this interface those classes objects are reffered as "resources".

(3) All resources reference by default are treated as implicity final and hence we can't perform reassignment with in try block.

Ex:-

```
try (BufferReader br = new BufferReader(new FileWriter
                                        ("abc.txt"))
    br = new BufferReader(new File
                          writer("abc.txt"));
}
```

Output :: (E: can't reassign a value.

(4) ~~Until~~ Until 1.6 Version try should compulsorily be followed by either catch or finally, but from 1.7. Version we can take only take try with resources without catch or finally

```
try(R) {
    //Valid
}
```

(5) Advantage of try with resources concept is finally block with become dummy because we are not required to close resources explicitly.

(6) try with resource nesting is also possible.

```
try (R1) {
    try (R2) {
        try (R3 {
        }
    }
}
```

Multi Catch block

Till Jdk 1.6 eventhough we have multiple exception having same handling code we have to write a seprate catch.
block for every exceptions, it increases the length of the code and reviews readability

logic

```
try {
  =
  =
}
catch (Arithmetic Exception ae){
      ae.printStackTrace();
  }
Catch (Null pointer Exception ne){
     ne.printStackTrace();
}
```

to overcome this problem SUN MS has introduced "Multi catch block" concept in 1.7 version

```
try {
  =
  =
}
catch (ArthimeticException | Null pointer Exception e){
     e.printStackTrace();
```

↓

In multiple catch, there should not be any relation blw exception types (either child to parent or parent to child or same type) it would result in compile time error.

eg:: try{
}
catch (ArthimeticException | Exception e){
e.printStackTrace();

Output: compileTime error

---

throw ⇒ handle the exception using catch block and throw it back the exception object to the caller

throws ⇒ method signature and commonly used if the exception is "CheckException".

CheckedException ⇒ Compiler will check for the handling code only then compilation is successful.

eg: IOException, SQLException, ... are all Checked Exceptions.

UncheckedExceptions ⇒ Compiler will not check for the handling code, but jvm come into picture. and possiblity of "Successful" or "abnormal" termination.

eg: RuntimeException and its child classes. error and its child classes are all "UnCheckedExpections".

Rules of overridding when exception is involved

While overridding if the child class method throws any Checked exception.

Compulsory the parent class method should throw the same Checked execption or its parent otherwise we will get compile Time error. There are no restrictions on unchecked Exception.

Eg:

```
Class parent {
    public Void methodone();
}
Class Child method extends parent {
    public Void methodone(){ throws exception { }
}
```

error: methodOne() in child cannot override method one() in parent public Void methodOne() throws exception {} overridden method does not throw exception

Rules w.r.t Overriding

parent: public Void methodone() throws Exception {}
child: public Void methodOne()
output: Valid.

parent: public Void methodOne(){}
child: public void methodOne() throws exception{}.
output: Invalid

parent : public void methodone () throws Exception{}
child : public void methodone () throws Exception{}
output : valid.

---

parent: public void methodone () throws IoException{}
child: public void methodone () throws IoException{}
Output: valid.

parent: public void methodone () throws IoException {}
child: public void methodone () throws fileNotfound
Exception, EOFException{}
output: valid

Instanceof

1. we can use the instanceof operator tocheck wheather the given an object is particular type or not

r instance of X

r => reference
X => Class/interface Name.

eg: ArrayList al = new ArrayList(); //inbuilt object where we can keep any type of other objects

```
al.add(new Student()); //0th position.
al.add(new Cricketer()); //1st position.
al.add(new Customer()); //2nd position.
Object o = l.get(0); // l is an arraylist object
if (o instanceof Student){
        Student s = (Student)o;
```

```
}
else if (o instanceof Customer) {
    Customer c = (Customer) o;
```

eg#2 `Thread t = new Thread();`

```
System.out.println(t instanceof Thread); //true
System.out.println(t instanceof Object); //true
System.out.println(t instanceof Runnable); //true.
```

Ex:- public class Thread extends Object implements Runnable {

}

⇒ To use instanceof operator compulsory there should be some relation between argument types. either child to parent or parent to child or same type) Otherwise we will

eg:- `String s = new String("sachin");`
`System.out.println(s instanceof Thread); //CE`

```
Thread t = new Thread();
System.out.println(t instanceof String; //CE
```

⇒ whenever we are checking the parent object is child type or not by using instanceof operator that we get false.

```
Object o = new Object();
System.out.println(o instanceof String); //false.
Object o = new String("ashok");
```

System.out.println (0 instanceof String); // true

→ for any class or interface x null instanceof x is always returns false

System.out.println (null instanceof X); // false

```
public class Test {
    public static void main (String[] args) {
        Object t = new Thread();
        System.out.println (t instance of Object); // true
        System.out.println (t instance of Thread); // true
        System.out.println (t instance of Runnable); // true
        System.out.println (t instance of String); // false
    }
}
```

Is instanceof()

Difference b/w instanceof and isinstance();

instanceof

instanceof an operator which can be used to check whether the given object is particular type or not we know at the type at begining it is available.

eg:- String s = new String ("sachin");

System.out.println (s instanceof Object); // true

// if we know the type at the begining only.

isInstance()

isInstance() is a method, present in class class, we can use isInstance() method to checked whether the given Object is particular type or not we don't know at the type at beginig it is available Dynamically at Runtime.

```
Class Test {
    public static void main(String[] args) {
        Test t = new Test();
        System.out.println(class.forName(args[0]).isInstance
(t)); //// arg[0] ---we don't know the typ at
beginig }
}
```

Java Test Test // true
Java Test String // false
Java Test Object // true.

———o———

01/12/22 Multi Threading Day1

Task → Activity/piece of work

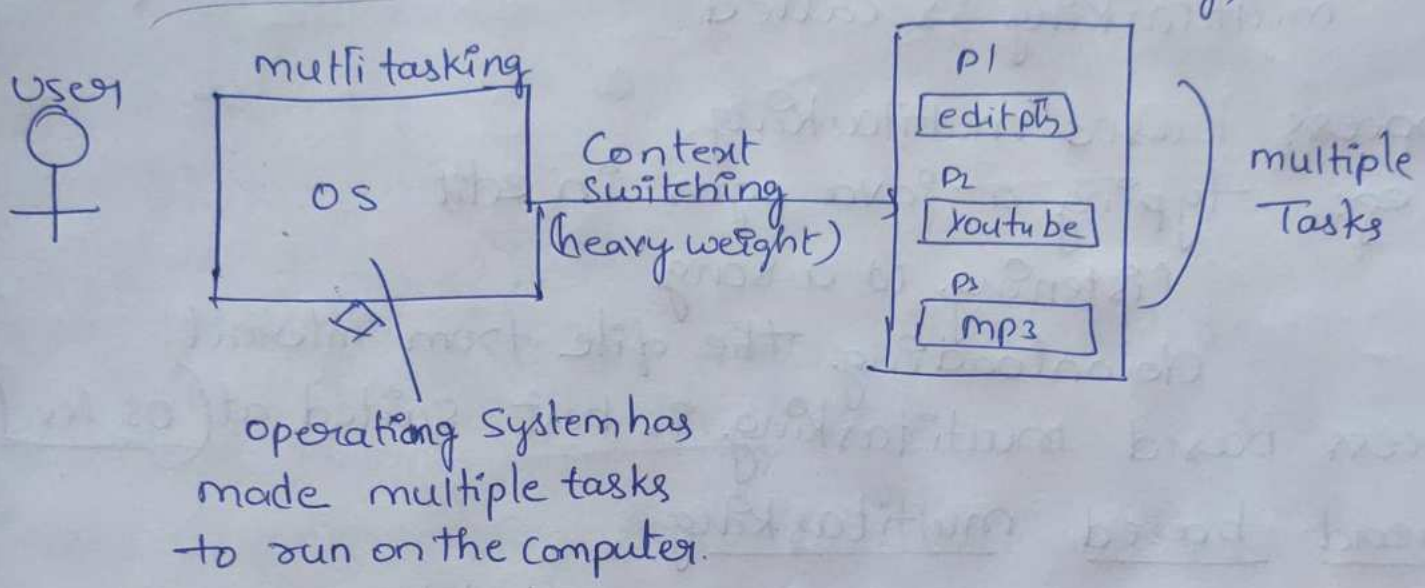
→ Syllabus

1. Introduction.
2. The ways to define, instantiate and start a new Thread
   1. By extending Thread class
   2. By implementing Runnable interface.
3. Thread class constructors
4. Thread priority.
5. Getting and setting name of a thread.
6. The methods to prevent (stop) Thread execution
   1. yeild()
   2. join()
   3. sleep()
7. Synchronization.
8. Inter Thread Communitedion.
9. Dead lock
10. Deamon Threads
11. Various Conclusion.
    1. To stop a Thread
    2. Suspend & resume of a thread.
    3. Thread group
    4. Green thread.
    5. Thread local

12. Life cycle of a Thread.

Task → Activity

process based



multi tasking — two types

1) process based multitasking

2) Thread Based multiTasking → will learn in Java (O.S)

i.e.
C++
Java → "multithreading is very easy

90% by API
10% only by developer

JAVA
|
API (Thread, Runnable, ThreadLocal
↓
packages (java.lang)
.class files.

---

multitasking :-

Executing several task simultaneously is the concept of multitasking

There are 2 types of Multiple tasking.

a) process based multitasking
b) thread based mutlipTasking.

process based multitasking

Executing several tasks simultaneously where each task is a separate independent process such type of multitasking is called "process based multitasking.

eg: typing a java pgm in edit
listening to a song
downloading the file from internet.

process based multitasking is best suited at ("os level").

Thread based multitasking

→ Executing several tasks simulatenously where each task is a seperate independent part of the same progsam, is called "thread based multitasking". Each independent part is called "Thread".

(1.) This type of multitasking is best suited at "programatic level".) The main advantages of multitasking is to reduce the response time of the system and to improve the performance.

(2.) The main & important application areas of multithreading are

a. To implement multimedia graphics
b. to develop web application servers (will learn in JEE)
c. to develop video games.
d. To develop animations.

(3) Java provides inbuilt support to work with threads through API called Thread, Runnable Thread Group, Thread local, ...

(4) To work with multithreading, java developers will code only for 10% remaining 90% java API will take care..

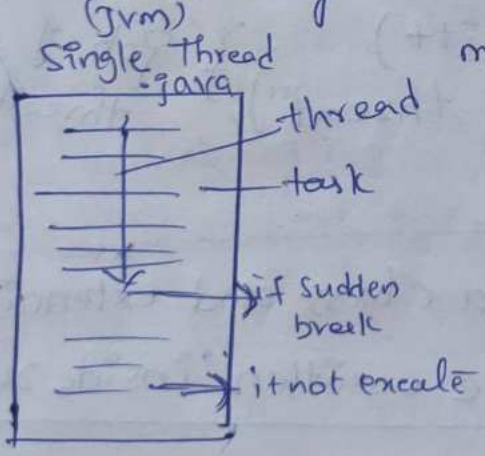
(what is thread?)

A) Seperate flow of execution is called "thread": if there is only one flow then it is called "single thread" programming for every thread there would be seperate job

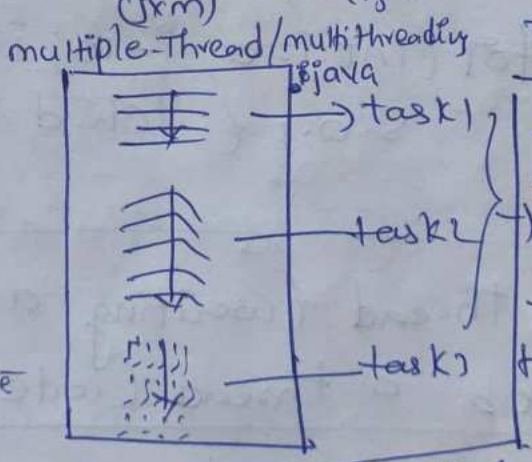
B) In java we can define a thread in 2 ways

a. Implementing Runnable interface

b. Extending Thread class (Thread are light weight) (T-S) 3min



1.



Thread scheduler

→ less the response time from application more could the perform

→ To utilize the cpu time effectively in our application we multiple task and each task assign to one thread and promote "multiThreading"

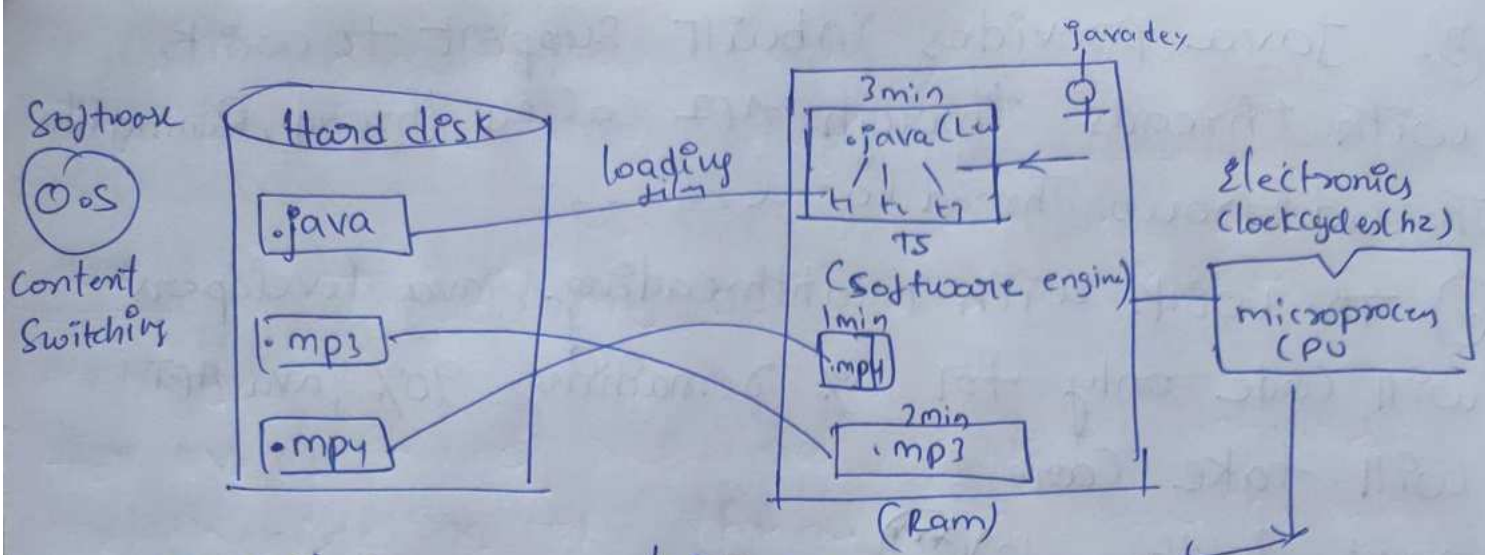
more the waiting time, performance of the application would decrease

↓

~~Delegation~~ Agenda of multitasking / multithreading

↓

to use cpu time effectively, so that performance can be improved.



51% Electronic 49% Softwa. Time allocation to every process is decided by O.S

(1.) Extending Thread Class

→ we can create a Thread by extending a thread.

```
Class myThread extends Thread {
    @override
    public void run(){
        for(int i=0; i<10; i++)
            S.O.p("Child thread");
    }
}
```

} Job of thread

defining a thread (writing a class and extending)
a Thread job a thread (code written inside run())

```
Class ThreadDemo {
    public Static void main(String ... args){
        myThread t = new myThread(); // Thread initiation
        t.start(); // Starting a thread.
        ;;;; // At this line 2 threads are there
        for(int i=1; i<=5; i++){
            S.O.p("main Thread");
    }
}
```

Behind the scences

① main thread is created automatically by JVM

② main thread creates child thread and starts the Child thread t.start();

jvm to jvm
system to System

Thread schedular (JVM)

→ if multiple threads are waiting to execute, then which thread will execute 1st is decided by Thread schedular which is part of JVM

In case of Multithreading we can't predict the exact output only possible output we can except

Since jobs of threads are important, we are not interested in the order of performance should be improved.

Case2 diff b/w ((t.start()) and (t.run()))

if we call t.start() and seperate thread will be created which is responsible to execute run() method.

if we call t.run(), no seperate thread will be created rather the method will be called just like normal method by main thread.

if we replace t.start() with t.run() then the output of the program would be

th:- ↓

Child thread - 10 times (loop)
main thread - 10 times (loop) (single thread)

Case 3 :- (Importance of Thread class start() method).

for every thread, required mandatory activities like registering the thread with ThreadScheduler will be taken care by Thread class start() method and programmer is responsible of just doing the job of the thread inside run() method

Start() acts like an assitance to programmer.

```
public void start()
{ register thread with ThreadScheculer
    All other mandatory low lerel activities.
} invoke or calling run() method.
```

we can conclude that without executing thread class start() method there is not chance of starting a new thread in java.

Duke is this start() is considered as heart of multitasking.

Case 4) ~~(if we are not overriding run() method~~

if we are not overridding run() method then thread class run() method will be executed which has empty implementation and hence we won't get any output.

eg:- Class mythread extends Thread{ }

```
Class ThreadDemo{
    public static void main (string ... args){
        my thread t= new mythread();
        t. start();
    }
}
```

it is highly recommended to override run() method, otherwise don't go for multithreading concept.

(ctrl+shift+t)

2/12/22 Day 2

Case 5 :- Over loading of run() method

we can overload run() method but Thread Class start() will always call run() with zero argument. If we overload run method with arguments, then we need to explicitly call argument based run method and it will be executed just like normal method.

eg:

```
Class mythread extends Thread {
    public void run(){
        System.out.println ("no arg method");
    }
    public void run(int i) {
        System.out.println ("zero arg method");
    }
}
```

```
Class ThreadableDemo {
    public static void main(String... args){
        MyThread t = new MyThread();
        t.start();
    }
}
```

Output :: No arg method //

Case 6 : Overriding of start() method

if we override start() then our start() method will be executed just like normal method, but no new Thread will be created and no new thread will be started.

Eg #1:

```
class MyThread extends Thread{
    public void run(){
        System.out.println("no arg method");
    }
    public void start(){
        System.out.println("start arg method");
    }
}
Class ThreadDemo {
    public static void main(String... args){
        myThread t = new mythread();
        t.start();
    }
}
```

output : start arg method) multithreading only not then

It is never recommend to override start() method

Case 7

Eg#1

```
class MyThread extends Thread {
    public void run() {
        System.out.println("run method");
    }
    public void start() {
        System.out.println("start method");
    }
}
class ThreadDemo {
    public static void main(String ... args)
    MyThread t = new myThread();
    t.start();
    System.out.println("main method");
}
}
```

Output: (main thread)

a. main method
b. Start method } output → produced by only main thread

Eg#2

```
class myThread extends Thread {
    public void start() {
        super.start();
        System.out.println("start method");
    }
    public void run() {
        System.out.println("run method");
    }
}
```

```
Class ThreadDemo{
    public static void main(String ... args){
        mythread t = new mythread();
        t.start();
        System.out.println("main method");
    }
}
```

Output: main thread
a. main method } output
b. start method

User defined thread.
a. run method. — output — child thread will call run() method.

Case 8: (Life cycle of a Thread)

```
myThread t = new MyThread(); //Thread is in born state.
t.start(); //thread is in ready/runnable state
```

* If thread schedular allocates cpu time then we say thread entered into Running State
* if run() is completed by thread then we say thread entered into dead state.

⇒ Once we created a Thread object then the Thread is said to be in new state or born state.

⇒ Once we call start() method then the Thread is said to be in new state or born state.

⇒ Once we call start() method then the thread will be entered into Ready or Runnable state.

⇒ if Thread Schedular allocates cpu then the thread will be entered into running state.

⇒ Once run() method completes then the thread will entered into dead state.

—o—

Case 9 :- After starting the thread, we are not supposed to start the same thread again, then we say thread is in "IllegalThreadStateException".

```
MyThread t = new MyThread(); // thread is in
                             born state.
t.start();
 ....
 ....
t.start(); // IllegalThreadStateException
```

Creation of Thread using Runnable interface

(1.) Creating a thread using java.lang.Thread class.

a. use start() from thread class

b. override run() and define the job of the thread.

(2.) Creation of a thread requirement to SUNMS is an SRS interface Runnable {

```
    void run();
}
```

| It contains only run method.

↓

```
class Thread implements Runnable { //Adapter Class
    public void start(){
        1. Register the thread with ThreadScheduler
        2. All other mandatory low level
           activities (memory level)
        3. invoke or call run() method.
    }
    public void run(){
        //job for a thread
    }
}
```

Shortcuts of eclispe

ctrl + shift + T ⇒ To open a defnition of any class

ctrl + o ⇒ to list all the methods of the class

Note :- public java.lang.Thread();

1⇒ thread class start(), followed by thread class run()

public java.lang.Thread(java.lang.Runnable);

1⇒ thread class start(), followed by implementation class of Runnable run()

Defining a thread by implementing Runnable Interface:

```
public interface Runnable {
    public abstract void run();
}
public class Thread implements Runnable {
    public void start(){
    (1) register Thread with Thread schedular
    (2) All other mandatory low level activites.
    (3) invoke run()
    }
    public void run(){
        // empty implementation.
    }
}
```

eg::1

```
Class MyRunnable implements Runnable {
    @Override.
    public void run(){
        for (int i=1; i2=10; i++)
            System.out.print("child thread");
    }
}
```

```
public class threadDemo{
    public static void main (String ... args){
    myRunnable r= new MyRunnable();
    Thread t= new Thread(r); // Call MyRunnable run
    t.start();                  ↑ (passing rnunable as argument)
    for (int i=1; i<=10; i++)
        System.out.println("Main thread");
    }
}
```

output:

Main thread
a. main thread
....
....

bchild thread
a. child thread
....
....
....

Case Study

```
MyRumable r = new MyRunnable();
Thread t1 = new thread();
Thread t2 = new thred();
```

Case 1: t1.start(). A new thread will be created, which is responsible for executing Thread class run() - empty

~~t1.start()~~

Output : main thread
main thread }-

Case 2: t2.start() A new thread will be created which is responsible to executing MyRunnable run()

Output (Main thread) : main thread
main thread
main thread
main thread
main thread

User defined thread :- child thread
child thread
child thread
child thread
child thread

Case 3 :- t1.run() No new thread will be created but Thread class run() will be executed just like normal Method call.

Output :- Main thread. main thread
main thread
main thread
main thread
main thread.

Case 4: t2.run() → No new thread will be created, but myRunnable class run() will be executed just like normal method call.

output: main thread →
Child thread main thread
child thread main thread
Child thread main thread
Child thread main thread
Child thread, main thread

Case 5: r.start() → It results in compile error
→ (cannot symbol method start() my runnable)

Case 6: r.run() No new thread will be created, but MyRunnable class run() will be executed just like normal method call.

output Mainthread.
Child thread | main thread
Child thread | main thread
Child thread | main thread
Child thread | main thread
Child thread | main thread.

(myRunnable r = new MyRunnable();
Thread t1 = new Thread();
Thread t2 = new Thread();

Case 1 : t1.start()

Case 2 : t2.start()

Case 3 : t2.run()

Case 4 : t1.run()

Case 5 : r.start()

Case 6 : r.run()

Mentors for this handwritten notes :

Hyder Abbas (linkedin.com/in/hyder-abbas-081820150/)
Nitin M (linkedin.com/in/nitin-m-110169136/)
Navin Reddy (linkedin.com/in/navinreddy20/) (Telusko)
wrttien by pallada vijaykumar
@ineuron.ai

In which of the above cases a new thread will be created which is responsible for the executation of my Runnable run() method?

t2.start();

In which of the above cases a new thread will be created!

t1.start();
t2.start();

In which of the above cases MyRunnable class run() will be executed

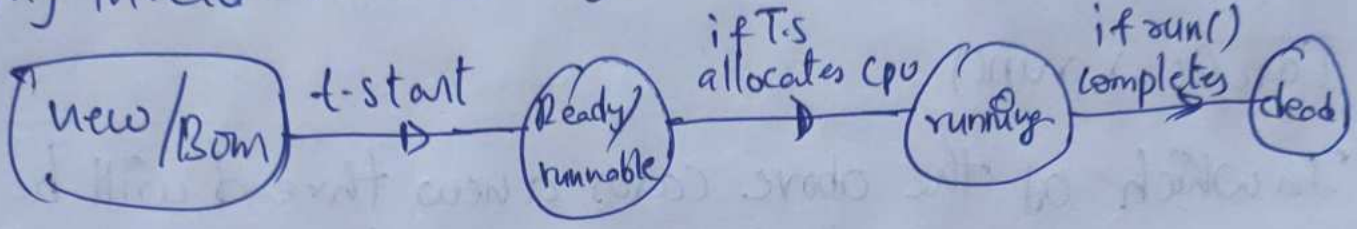
t2.start();
t2.run();
r.run();

Different approach for creating a thread will be created ?

~~t1.start();~~
~~t2.start();~~

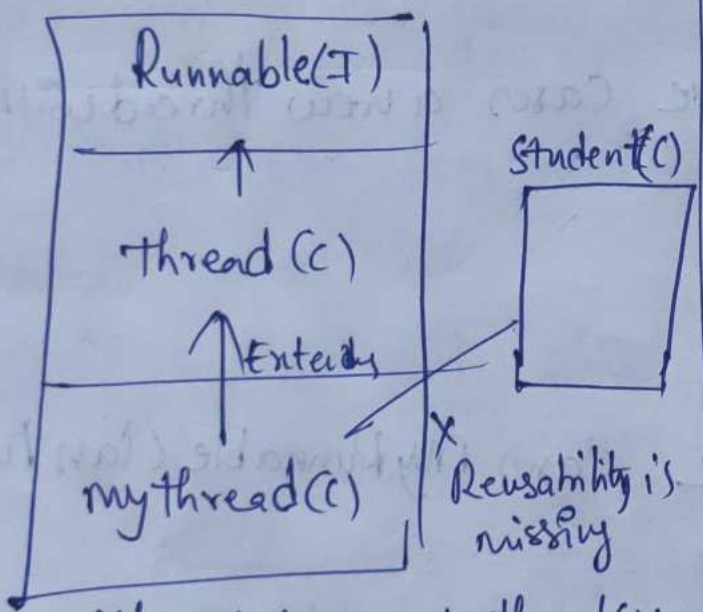
A. extending thread class.
B. implementing Runnable Interfaces

life cycle of thread



not Suggestable

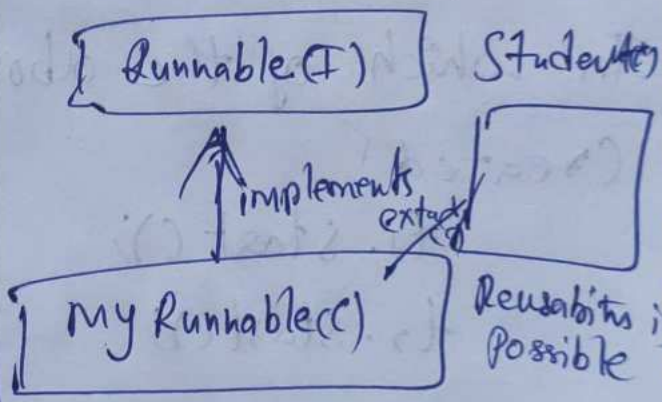
1st Approach: Extending Thread class



my Thread t = new Mythread();
t.start();

good approach

2nd Approach: implementing Runnable



MyRunnable r = new MyRunnable();
Thread t = new Thread(r)
t.start().

Q. Which approach is good in above?

→ a. Implements Runnable Interface is recommended. becaz our class can extend other class through which Inheritance benifit can brought into our class. Internally performance and memory level is also good when we work with Interface.

b. if we work with extends feature then we will miss out inheritance benefit becoz already our class has inherited the feature from "Threadclass" so we usually don't prefere .extends approach rather implements approach is used in real time for working with "multi threading".

Various constructors available in Thread class

a. thread t = new thread() ← daily using (✓)
b. thread t = new Thread (Runnable r)
c. thread t = new thread (String name)
d. thread t = new thread (Runnable r, String name)
e. thread t = new thread (ThreadGroup g, String name);
f. thread t = new Thread (ThreadGroup g, Runnable r, String name);
g. thread t = new Thread (ThreadGroup g, Runnable r, String name, long Stacksize);
h. thread t = new Thread (ThreadGroup g (Runable r));

Alternate approach to define a thread (not recommended)

```
class mythread extends Thread {
    public void run() {
        System.out.println ("child thread");
    }
}
class ThreadDemo {
    public static void main (String ... args) {
```

```
    mythread t = new mythread();
    Thread t1 = new Thread(t);
      t1.start();
    System.out.println("main thread");
}
```

(but not recommended)

Output: 2 threads are created.

main thread
main thread
Child thread
Child thread

Internally related

Runnable
↑
Thread
↑
mythread

Names of the thread

Internally for every thread, there would be a name for the thread.

a. name given by jvm
b. name given by the user.

eg:
```
class mythread extends Thread {
}
public class TestApp {
    public static void main(String ... args){
```

```
System.out.println(Thread.currentThread().getName());
                                                    //main

myThread t = new myThread();
t.start();
System.out.println(t.getName()); //Thread-0

Thread.currentThread().setName("yash"); //yash

System.out.println(Thread.currentThread().getName());
                                                    //yash

System.out.println(10/0);
    // Exception in thread "yash". java.lang.Arthim
                                        etic Exception:
                                        by/by Zero
        TestApp.main()
    }
}
```

→ It is also possible to change the name of the thread using setName(). | thread.currentthread().setName("vijay")

→ It is possible to get the name of the thread using getName(). thread.currentthread().getName(); (Vijay)

methods :

```
public final String getName();
public final void setName(String name);
```

eg#2.

```
class mythread extends Thread {
    @Override.
```

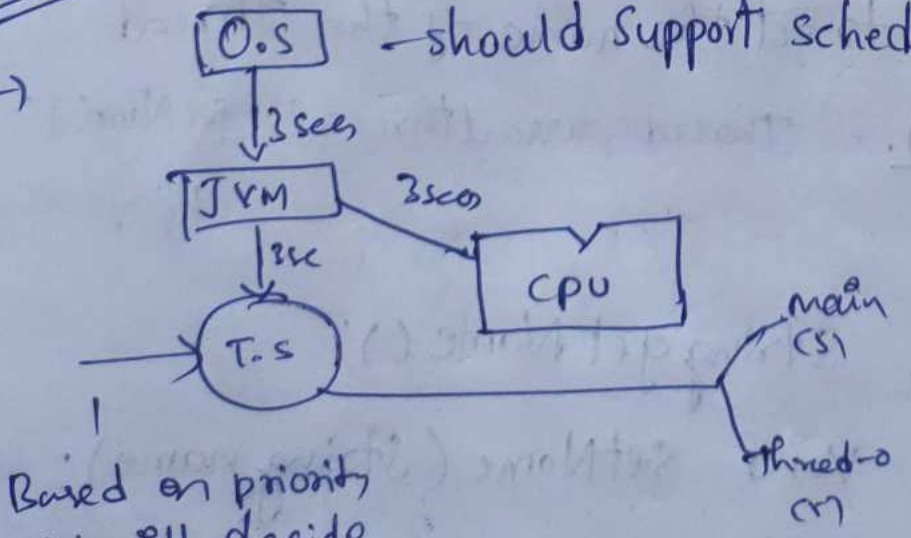
```
public void run() {
    System.out.println("run() executed by
                        thread::"+thread.
                        currentThread().getName());
}
}
public class TestApp{
    public static void main(String...args){
        myThread t=new myThread();
        t.start();
System.out.println("main() executed by thread::"
+thread.currentThread().getName());
}}
```

Output :: run() executed by thread :: thread-0
main() executed by thread :: main

5/12/22

→ O.S —should support scheduling algorithms!

Ranking priority 10

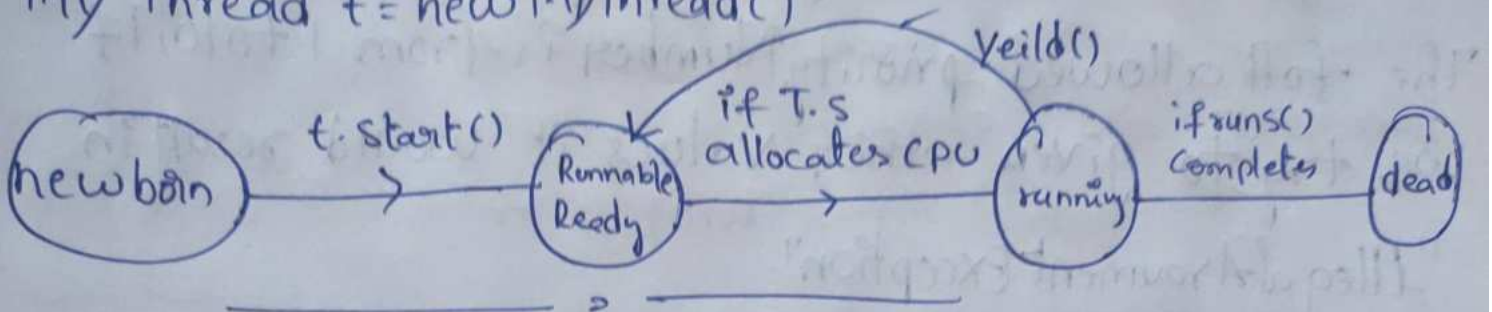


JVM will create main thread and it only start with a default priority 5

if both the threads have same priority then, Ts uses algorithm which is vendor dependant.

life Cycle of Thread. — (1) Ex: Yeild

My Thread t = new MyThread()



Thread priorities (priorities)

* For every Thread in java has some priority.
* Valid range of priority is 1 to 10, it is not 0 to 10.
* if we try to give a different value the it would result in "IllegalArgumentException".

Thread.MIN_PRIORITY = 1
Thread.NORM_PRIORITY = 5
Thread.MAX_PRIORITY = 10

* Thread class does not have priorities is Thread. LOW_PRIORITY, Thread.HIGH_PRIORITY.
* Thread scheduler allocates cpu time based on "priority"
* if both threads have the same priority then when which thread will get a chance as a pgm we can't predict bcoz it is vendor dependent.
* we can set and get priority values of the thread using the following methods.

↓

a. public final void setPriority(int priorityNumber)
b. public final int getPriority().

The ~~foll~~ allowed priorityNumber is from 1 to 10, if we try to given other values it would result in "IllegalArgumentException".

System.out.println(Thread.currentThread().setPriority(100);
// IllegalArgumentException.

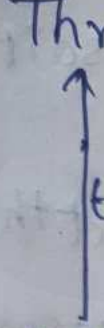
Default priority

The default priority for only main thread is "5", where as for other threads priority will be inherited from parent to child.
parent Thread priority will be given as child thread priority.

Eg#1 → class MyThread extends Thread {
}
public class TestApp {
public static void main(String ... args) {
System.out.println(Thread.currentThread().getPriority()); //5
Thread.currentThread().setPriority(7);
MyThread t = new MyThread();
System.out.println(Thread.currentThread().getPriority()); //7.
}
}

myThread t;
new myThread.

reference



myThread is creating by "mainThread", so priority of "mainThread" will be shared as a priority for "MyThread".

mainthread, parent of child thread.

Eg #2

```
class Mythread extends Thread {
    @override
    public void run(){
        for (int i=1; i<=5; i++ ){
            System.out.println ("Child thread");
    } } }

public class TestApp {
    public static void main (string ... args){
        myThread t = new Mythread ();
        t.setpriority (7); //line 1
        t.start ();
        for (int i=1 ; i<=5 ; i++) {
            System.out.println ("main thread");
    } } }
```

Since priority of child thread is more than main thread, jvm will execute child thread first whereas for the parent thread priority is 5 so it will get last chance.

if we comment line-1, then we can't predict the order of execution beco2 both the threads have same priority.

Some platform won't provide proper support for thread priorities.

eg:- Window 7, Windows 10, ...

We can prevent threads from Execution.

a. Yeild() ← refer (1) life cycle.
b. join()
c. sleep()

Yeild() ⇒ it creates to pause current executing Thread for giving chance for waiting threads of same priority..

* if there is no waiting Threads have low priority then same thread can continue its execution

* if all the threads have same priority and if they are waiting then which thread will get chance we can't except; it depends on Thread scheduler.

* The thread which is yeild, when it will get the chance once again depends on the mercy on "thread schedular" and we can't expect exactly

public static native void yeild()

- mythread t = new Mythread() // new state or born state
  t.start() // enter into ready state / runnable state.
- if thread schedular allocates processor then enters into running state

a. if running Thread calls yeild() then it enters into runnable state

if run() is finished with execution then it ~~comes~~ enters into dead state

eg#1.
```
class MyThread extends Thread{
    @Override
    public void run(){
      for (int i=1 ; i<=5 ; i++ ){
          System.out.println ("child thread");
          Thread.yield (); //line1
      }
    }
}
public class TestApp {
    public static void main(String ... args){
      myThread t = new Mythread();
      t.start();
      for (int i=1; i<=5 ; i++){
          System.out.println ("parent thread");
      }
    }
}
```

Note: if we comment line-1, then we can't expect the output becoz both the threads have same priority then which thread the Threadschedular will Schedule is not in the hands of progsammer but if we don't Comment line-1, then there is a possibility of main thread getting no more no of times, so main thread execution is faster than child thread will get chance.

Note: Some platforms won't provide proper support for yield(), because it is getting the execution code from other language preferebly from 'c';

(b) join()

if the thread has to wait until the other thread finishes its execution then we need to go for join().
if t1 executes t2 join() then t1 should should wait till t2 finishes its execution.
t1 will be entered into waiting state until t2 completes, once t2 completes then
t1 can continue with its execution.

eg#1
Venue fixing ⟹ t1.start()
wedding card printing ⟹ t2.start() ⟹ t1.start
join()
wedding card distrubtion ⟹ t2.start() ⟹ t2.join().

Prototype of join()

public final void join() throws InterruptException.
public final void join(long ms) throws InterruptException
public final void join(long ms, int ms) throws interruptExcepti

Note: while one thread is in waiting state and if one more thread interupts then it would result in "InterruptedException". InterruptException is Checked Exception which should always be handled.

```
Thread t = new Thread(); // new / born state
    t.start(); // ready / runnable state.
```

→ if T.S allocates cpu time then Thread enters into running state.

→ if currently executing Thread invokes t.join()/t.join(1000), t.join(1000, 100), then it would enter into waiting state.

→ if the thread finishes the execution / time expires / interupted then it would come back to ready state / runnable state.

→ if run() is completed then it would enter into dead State.

eg#1

```
class Mythread extends Thread {
    @Override
    public void run() {
        for (int i=1; i<=10; i++) {
```

```
            System.out.println("sita Thread");
            try {
                Thread.sleep(2000);
            }
            Catch (InterruptedException e){
            }
        }
    }
}
public class Test3 {
    public static void main(String...args)throws
                                        InterruptedExceptio
    Mythread t=new mythread();
        t.start();
        t.join(10000); //line-n1
        for (int=1; i<=10; i++){
            system.out.println("rama thread");
        }
    }
}
```

⇒ if line-n1 is commented then we can't predict the output beco2 it is the duty of the T.S to assign C.P.U time

⇒ if line-n1 is not commented, then rama thread (main thread) will enter into waiting state till Sita thread (child thread) finishes its execution.

Output

2. Threads

a. Child thread
   Sita thread
   sita thread
   ...

b. Main thread
   rama thread
   rama thread
   ...

Waiting of Child Thread until completing main Thread.

we can make main thread to wait for child thread. as well as we can make child thread also to wait for main thread

eg #1

```
class Mythread extends Thread {
    Static Thread mt;
    @ overaride
    public void run() {
        try {
            mt.join();
        }
        Catch (InterruptException e) {
        }
    for ( int i=1;i<=10;i++) {
            System.out.println ("child thread");
} } }
```

```
public class Test3 {
    public static void main (String ... args)
    mythread.mt = Thread.currentThread(); throws InterruptedExcep

    mythread t = new Mythread();
      t.start();

    for (int i=1 ; i<=10 ; i++){
          System.out.println ("main thread");
          Thread.sleep (2000); //to see sleep.
    }
}
```

Output

2 Threads (main thread, child thread)

```
main Thread
  a. main thread
     ...
     ...
Child thread
  a. child thread.
     ...
     ...
```

Eg#2

```
class MyThread extends Thread {
      static Thread mt;
      @override
      public void run(){
          try{
              mt.join();
```

```
        catch (InterruptException e){
        }
        for (int i=1; i<=10; i++ ){
            System.out.println ("child thread");
        }
    }
}

public class Test3 {
    public static void main (String...args)throws IE{
        mythread t= new mythread();
        t.start();
        t.start();
        t.join();
        for (int i=1; i<=10; i++){
            System.out.println ("main thread");
            Thread.sleep (2000); //20 sec sleep
        }
    }
}
```

output:

2 threads (main, child thread)

main thread

:

:

Child thread.

Note: if both the threads invoke t.join(), mt.join() then the program would result in "deadlock".

eg #3

```
public class Test3 {
    public static void main(String... args) throws
                                                IE{
        Thread.currentThread().join();
    }
}
```

Output:: Deadlock, beco2 main thread is waiting for the main thread itself

xxx

Sleep()

if a thread don't want to perform any operation for a particular amount of time then we should go for sleep().

Signature

public static native void sleep(long ms) throws IE

public static void sleep(long ms, int ms) throws IE

every sleep method throws InterruptedException, which is a checkedException so we should compulsorily handle the exception using try catch or by throws keyword otherwise it would result in compile time error

```
Thread t = new Thread(); //new or born state.
t.start() // ready/runnable state
```

→ if T.S allocates cpu time then it would enter into running state

→ if run() completes then it would enter into dead state

→ if running thread invokes sleep(1000)/sleep(1000,100) then it would enter into sleeping state.

→ if time expires / if sleeping thread got interrupted then thread would become back to "ready/runnable state

eg#1

```
public class SlideRotator {
    public static void main(String...args) throwsIE {
        for(int i=1; i<=10; i++){
            System.out.println("slide:"+i);
            Thread.sleep(5000);
        }
    }
}
```

Output:

```
slide::1
slide::2
slide::3
slide::4
slide::5
slide::6
slide::7
slide::8
slide::9
slide::10
```