Java Thread

Process:

Program/Application
It has its own memory, system resource such as file , memory, io mechanism
- heap, stack, code segment(compiled code), data segment (global var)
They do not share it with other process/program/application
heap size – 64MB per process

Thread:

Path of exeecution witin a program/process.
It is a sequence of instructions that the CPU can execute independently.
All program/process has atleast one thread - main thread
Threads are lightweight processes within a process/program
Multiple threads can exist within a single process
They sharing the process's resources, such as memory and open files
Each thread has its own execution stack and registers.

stack size 512 KB to 1 MB. per thread

Multitasking:

Definition:
ability of an OS to execute multipletasks (processes or threads) concurrently.
It enhances system efficiency by use CPU time concurrently.

Types of Muktitasking:

1. Process-Based Multitasking:
   running multiple processes/programs concurrently
2. Thread-Based Multitasking: => or Multi-threading
   -running multiple threads within the same process concurrently.
   -shares the same memory space and resources of its parent process.
   thats why its called light weight process.

Advantage of Multithreading

- write very efficient programs
– that make maximum the use of the CPU,
therfoer idle time can be kept to a minimum

Single threaded vs Multithreaded

- single-threaded, one program has to wait for other program to finish
even though the CPU is sitting idle most of the time.
- single-threaded use an approach called an "event loop with polling".
- Multithreading helps to effectively make use of this idle time
- entire environment to be asynchronous

Eg, read/write - transport interface

The Java Thread Model

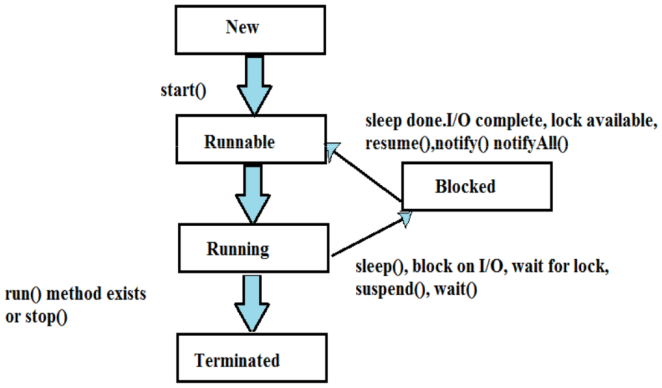
Single-threaded systems use an approach called an ***event loop with polling***.
– In this model, a single thread of control runs in an infinite loop,
polling a single event queue to decide what to do next.

– In a singled-threaded environment, when a thread blocks (that is,
suspends execution) because it is waiting for some resource, the
entire program stops running.

Thread States

– ***New*** - Thrad t = new Thread(), when thread object is created
– ***Runnable*** –ready to run = t.start(), but not yet schedules CPU time.
– ***Running*** - scheduled, and executing instructions in CPU
– ***Non-Runnable (Blocked)*** = reaches this state, if t.sleep(1000), or t.wait(), or t.suspend() called from runnig state
– ***Terminated*** = t.close()

Thread Life cycle



Thread Priorities

1. In Java, thread priorities are used to indicate the importance of a thread in relation to other threads.
2. The priority of a thread can influence the order in which threads are scheduled for execution.
3. *A higher-priority thread doesn't* ***necessarily run*** *any faster than a lower-priority thread, because Priority affects scheduling, not speed.*
4. *A thread's priority is used to decide when to switch from one running thread to the next.*

Here are the different thread priorities in Java:

MIN_PRIORITY (1): The lowest priority for a thread. This is typically used for threads that perform background tasks and are not critical to the application's performance.
NORM_PRIORITY (5): The default priority assigned to threads. Most threads in a typical application use this priority.
MAX_PRIORITY (10): The highest priority for a thread. This priority is generally reserved for threads that perform time-sensitive operations or critical tasks.

Usage:

```
Thread thread1 = new Thread(() -> { /* task */ });
thread1.setPriority(Thread.MAX_PRIORITY);
Thread thread2 = new Thread(() -> { /* task */ });
thread2.setPriority(Thread.MIN_PRIORITY);
```

```
System.out.println("Thread 1 Priority: " + thread1.getPriority());
System.out.println("Thread 2 Priority: " + thread2.getPriority());
```

The rules that determine when a context switch takes place are:
– A thread can voluntarily relinquish control. This is done by explicitly yielding, sleeping, or blocking on pending I/O. Here all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.
– A thread can be preempted by a higher-priority thread. Here a lower-priority thread is simply preempted(forcely suspended) by a higher-priority thread. i.e. As soon as a higher-priority thread wants to run, it can run. This is called preemptive multitasking.

Main Thread

- All program/process has atleast one thread - main thread, because it is executed when our program begins.
- Denotes that by default, the name of the main thread is **main**
- The main thread is created automatically when our program is started.
- Its priority is 5, which is the default value
- The main thread is important for two reasons:
    1. It is the thread from which other "child" threads will be spawned.
    2. Often, it must be the last thread to finish execution because it performs various shutdown actions.

Creating a Thread

1. Java's multithreading system is built upon the Thread class, its methods and its companion interface Runnable.
2. Thread encapsulates a thread of execution
3. Java defines two ways for creating thread:
    1. **extend the Thread class**.
    2. **implement the Runnable interface.**

Thread Class.
Constructor and Description

1. **Thread()** : Allocates a new Thread object.
2. **Thread(String name)** : Allocates a new Thread object, also sets name to thread.
3. **Thread(Runnable target)** : Allocates a new Thread object. Via runnable interface
4. **Thread(Runnable target, String name)** : add name of the string to above 3rd method.

Thread class methods

1. **getName() / setName()** - Get/Set a thread's name.
2. **GetPriority() / setPriority()** - Get/Set a thread's priority.
3. **IsAlive()** - Determine if a thread is still running.
4. **Join()** - Wait for a thread to terminate.
5. **run()**- Entry point for the thread.
6. **sleep()**- Suspend a thread for a period of time.
7. **start()**- Start a thread by calling its run method

Method 1: Creating a Thread by Extending Thread class

1. Create a new class that extends Thread class,
2. The extending class must override the run( ) method, which is the entry point for the new thread.
3. It must also call start( ) to begin execution of the new thread
4. then create an instance of that class.

Eg: Implement a thread simulating a person, say Alice is runnig a 100 meter sprint.
Aice can complete one meter in one second.

```
class ThreadOne extends Thread {
    ThreadOne() {
        super("sprinter-alice");
        System.out.println("Child thread: " + this);
    }
    public void run() {
        int meter = 0;
        while(meter < 100) {
            System.out.println("Thrad Name = " + this.getName()
                    + " runs " + meter++ + " meter ");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Child thread .. exiting");
    }
}
public class ThreadOneTest {
    public static void main(String[] args) {
        System.out.println("Main thread...! ");
        ThreadOne t1 = new ThreadOne();
        // spawn new thread
        t1.start();
    }
}
```

Output

```
Main thread...!
Child thread: Thread[sprinter-alice,5,main]
Thrad Name = sprinter-alice runs 0 meter
Thrad Name = sprinter-alice runs 1 meter
Thrad Name = sprinter-alice runs 2 meter
Thrad Name = sprinter-alice runs 3 meter
.....
.....
Thrad Name = sprinter-alice runs 99 meter
Child thread .. exiting
```

- *public void run()*
  *The program loginc is written in this function. This will be startes when **thread.start()** is called.*
  *When the **run()** function exits, the thread will be terminated.*

- *ThreadOne t1 = new ThreadOne();*
  *t1.**start**();*
  start() function is inherited from Thread class.
  *When **start()** function is called, public void run() will be invoked.*

- *ThreadOne() {*

super("sprinter-alice");
}
This is the Constructor. The call to super( ) inside this invokes the Thread constructor. The name "sprinter-alice " will be set as thread name.

Method 2: Creating a Thread by implementing **Runnable** interface.

What is Runnable Interface ?

The Runnable interface in Java is a functional interface that represents a task or a unit of work that can be executed by a thread.

It contains a single abstract method, **public void run()**, which defines the code that constitutes the task.

The Runnable interface is commonly used to create and start threads without subclassing the Thread class

Creating a thread by implementing Runnable Interface

1. Create a new class that extends Runnable interface.
2. The implementing class must override the run( ) method which is defined as abstract mthod in interface Runnable.
3. Create an object of this implementing class.
4. Create an object of Thread class by wrapping the object of implemeting class.
5. invoke thread obect.start() to begin the execution of new thread.

Eg: Implement a thread simulating a person, say Alice is runnig a 100 meter sprint. Aice can complete one meter in one second.

```
class ThreadOne implements  Runnable {
        ThreadOne() {
                System.out.println("Child thread: " + this);
        }
        public void run() {
                int meter = 0;
                String threadName = Thread.currentThread().getName();
                while(meter < 100) {
                        System.out.println("Thrad Name = " + threadName
                        + " runs " + meter++ + " meter ");
                        try {
                                Thread.sleep(1000);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
        }
}
public class ThreadOne_Runnable {
        public static void main(String[] args) {
                System.out.println("Main thread...!
                ThreadOne threadRunnable = new ThreadOne();
                Thread t = new Thread(threadRunnable. "sprinter-alice")
                t.start();
        }
}
```

Output

```
Main thread...!
Child thread: ThreadOneTest.ThreadOne@3cb5cdba
Thrad Name = sprinter-alice runs 0 meter
Thrad Name = sprinter-alice runs 1 meter
Thrad Name = sprinter-alice runs 2 meter
Thrad Name = sprinter-alice runs 3 meter
.....
.....
Thrad Name = sprinter-alice runs 99 meter
Child thread .. exiting
```

Choosing an approach – when to use Thread and Runnable

If the class which is creating the thread is already extending some other class, then it can not extend Thread class to create thread, as java does not support multiple inheritance. In this case, we implement Runnable interface to create thread.

If the class which is creating the thread is Not inheriting any other class, then we can extend Thread class to create the thread.

Runnable interface is having only one anstract class run(). If we dont want to use any other functions (that inherit from Thread) , then we can use runnale interface.

Java Multiple Thread

In Java, multiple threads can be used to achieve concurrency, allowing a program to execute multiple tasks in parallel. Java provides built-in support for multithreading, which can be used to improve performance, especially in multi-core processors

Question 1
Simulate a 100-meter race between three runners, where:

- **Each runner runs at a different speed (i.e., time to cover 1 meter varies).**
- **The progress of each runner is displayed as they move forward.**
- **The race ends when all runners finish.**
- **Assumption : lower speed means faster**

```
class Sprinter extends Thread{
        int speed;
        Sprinter(String name, int speed) {
                super(name);
                this.speed = speed;
        }
        public void run() {
                int meter = 0;
                while(meter < 100) {
                        System.out.println(this.getName() + " runs " + meter);
                        meter ++;
                        try {
```

```
                        Thread.sleep(speed);
                    } catch(Exception e) {
                        System.out.println(e.getMessage());
                    }
                }
                System.out.println(this.getName() + " Finished ....!");
            }
        }
        public class SprintTest {
            public static void main(String[] args) {
                // Creating three runners with different speeds (lower speed means faster)
                Sprinter bolt = new Sprinter("Bolt", 1000);      // Usain Bolt
                Sprinter lewis = new Sprinter("Lewis", 1200);  // Carl Lewil
                Sprinter gatlin = new Sprinter("Gatlin", 1500); // Justine Gatlin
                System.out.println("Sprint starts...!");
                bolt.start();
                lewis.start();
                gatlin.start();
            }
        }
```

Output

```
Sprint starts...!
Bolt runs 0 meter
Gatlin runs 0 meter
Lewis runs 0 meter
....
Bolt runs 6 meter
Bolt runs 7 meter
Lewis runs 6 meter
Gatlin runs 5 meter
....
Lewis runs 83 meter
Bolt runs 99 meter
Bolt Finished ....!
Gatlin runs 67 meter
Lewis runs 84 meter
Gatlin runs 68 meter
...
Lewis runs 99 meter
Gatlin runs 80 meter
Lewis Finished ....!
.....
Gatlin runs 99 meter
Gatlin Finished ....!
```

Question 2

Simulate a 100-meter race between turtle and rabbit:

- rabbit runs a faster speed than turtle(i.e., time to cover 1 meter varies).
- The progress of each runner is displayed as they move forward.
- The race ends when all runners finish, and the winner will be Rabbit.
- Assumption : lower speed means faster

```
class Rabbit implements Runnable{
        String name;
        Rabbit(String name) {
                this.name = name;
        }
        public void run() {
                int meter = 0;
                while(meter < 100) {
                        System.out.println(name + " runs " + meter + " meters");
                        meter ++;
                        try {
                                Thread.sleep(500);
                        } catch(Exception e) {
                                System.out.println(e.getMessage());
                        }
                }
                System.out.println(name + " Finished ....!");
        }
}
class Turtle implements Runnable{
        String name;
        Turtle(String name) {
                this.name = name;
        }
        public void run() {
                int meter = 0;
                while(meter < 100) {
                        System.out.println(name + " runs " + meter + " meters");
                        meter ++;
                        try {
                                Thread.sleep(1000);
                        } catch(Exception e) {
                                System.out.println(e.getMessage());
                        }
                }
                System.out.println(name + " Finished ....!");
        }
}
public class RabbitRaceTest {
        public static void main(String[] args) {
                Rabbit rabbit = new Rabbit("Oswald Rabbit");
                Turtle turtle = new Turtle("Toby Turtle");
                Thread rabbit_thread = new Thread(rabbit);
                Thread turtle_thread = new Thread(turtle);
                System.out.println("Sprint started...!");
```

```
                rabbit_thread.start();
                turtle_thread.start();
            }
        }
```

Output

```
Sprint started...!
Toby Turtle runs 0 meters
Oswald Rabbit runs 0 meters
Oswald Rabbit runs 1 meters
Toby Turtle runs 1 meters
...
Toby Turtle runs 5 meters
Oswald Rabbit runs 10 meters
...
Toby Turtle runs 25 meters
Oswald Rabbit runs 50 meters
...
Toby Turtle runs 44 meters
Oswald Rabbit runs 88 meters
...
Oswald Rabbit runs 99 meters
Toby Turtle runs 50 meters
Oswald Rabbit Finished ....!
Toby Turtle runs 51 meters
Toby Turtle runs 52 meters
Toby Turtle runs 53 meters
...
...
Toby Turtle runs 99 meters
Toby Turtle Finished ....!
```

Question 3

Write a Java program that implements a multi-threaded program which has three threads. First thread generates a random integer every 1 second. If the value is even, second thread computes the square of the number and prints. If the value is odd the third thread will print the value of cube of the number.

```
import java.util.*;
class CubeThread extends Thread {
        int digit;
        CubeThread(int digit) {
                this.digit = digit;
        }
        public void run() {
                System.out.println("Cube of " + digit + " = " + (Math.pow(digit, 3)));
        }
}
class SquareThread extends Thread {
        int digit;
        SquareThread(int digit) {
                this.digit = digit;
        }
        public void run() {
```

```
                System.out.println("Square of " + digit +
                        " = " + (digit*digit));
        }
    }
    class FirstThread extends Thread {
        public void run() {
            Random random = new Random();
            while (true) {
                int digit = random.nextInt(10);
                System.out.println("digit = " + digit);
                if (digit % 2 == 0 ) {
                    // even number
                    SquareThread square =
                            new SquareThread(digit);
                    square.start();
                } else {
                    // odd numer
                    CubeThread cube = new CubeThread(digit);
                    cube.start();
                }

                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
    public class Lab7_Test {
        public static void main(String[] args) {

            FirstThread thread = new FirstThread();
            thread.start();
        }
    }
```

Output:

```
digit = 8
Square of 8 = 64
digit = 9
Cube of 9 = 729.0
digit = 4
Square of 4 = 16
digit = 4
Square of 4 = 16
digit = 7
Cube of 7 = 343.0
```

Thread Synchronization

When two or more threads need access to a shared resource, it is necessary to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization

1. Key to synchronization is the concept of the **monitor** (also called a **semaphore**).

2. A **monitor** is an object that is used as a *mutually exclusive lock*, or **mutex**.
   – Only one thread can own a monitor at a given time.

3. When a **thread acquires a lock**, it is said to have entered the monitor.
   – All **other threads waiting** to enter the locked monitor
   – that is, other threads will be suspended until the first thread exits the monitor.
   – These other threads are said to be waiting for the monitor.

4. A thread that owns a monitor can reenter the same monitor if it so desires.

How to Synchronize a thread ?

Can synchronize our code below two ways.

1. synchronized Methods
2. The synchronized Statement

Both involve the use of the **synchronized** keyword

1. Using synchronized method

a) Synchronization is easy in Java, because all objects have their own implicit monitor associated with them.

b) To enter an object's monitor, just call a method that is **modified with the synchronized keyword**.

c) While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait.

d) To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

Example Program – Using synchronized method

The program simulates a simple multi-threaded environment in Java where two threads print names to the console.

Classes

Callme Class:

This class represents a shared resource with a method printName(String name).

ThreadSyncronize Class:

Implements the Runnable interface, allowing instances to be run in separate threads.
Contains a reference to a Callme object and a name string.
The run method calls printName on the Callme instance, passing the name.

ThreadSyncronizeTest Class:

Contains the main method.

It creates an instance of Callme and two ThreadSyncronize objects, each associated with a different name ("One" and "Two").
Two threads are started, each executing a ThreadSyncronize instance.

```
public class Callme {
        // class simulating shared resource
        synchronized public void printName(String name) {
                System.out.print("[" + name);
                 try {
                        Thread.sleep(1000);
                } catch(Exception e) {}
                System.out.print("]");
        }
}
public class ThreadSyncronize implements Runnable{
        Callme callmeObj;
        String name;
        ThreadSyncronize(Callme callmeObj, String name) {
                this.callmeObj = callmeObj;
                this.name = name;
        }
        public void run() {
                callmeObj.printName(name);
        }
}

public class ThreadSyncronizeTest {
        public static void main(String[] args) {
                // create object of shared resource
                Callme callmeObj = new Callme();
                ThreadSyncronize t1 = new ThreadSyncronize(callmeObj, "One");
                ThreadSyncronize t2 = new ThreadSyncronize(callmeObj, "Two");
                Thread thread1 = new Thread(t1);
                Thread thread2 = new Thread(t2);

                thread1.start();
                thread2.start();
        }
}
```

Output

[One][Two]

If synchronized key word is NOT used, the output would be

Output

[One[Two]]

2. Using synchronized statement

Creating **synchronized methods** within is an easy and effective means of achieving synchronization, **but it will not work in all cases** as given below.

1. Suppose that we want to synchronize the access to objects of a class that does not use synchronized methods.

2. Suppose this class was created by a third party, and **we do not have access to the source code** – So **we can't add synchronized** to the appropriate methods within the class.

To solve this, simply put calls to the methods defined by this class inside synchronized block.

Example Program – Using **synchronized statement**

```
public class Callme {
        // class simulating shared resource
        public void printName(String name) {
                System.out.print("[" + name);
                 try {
                        Thread.sleep(1000);
                } catch(Exception e) {}
                System.out.print("]");
        }
}
public class ThreadSyncronize implements Runnable{
        Callme callmeObj;
        String name;
        ThreadSyncronize(Callme callmeObj, String name) {
                this.callmeObj = callmeObj;
                this.name = name;
        }
        public void run() {
                synchronized (callmeObj) {
                callmeObj.printName(name);
                }
        }
}
public class ThreadSyncronizeTest {
        public static void main(String[] args) {
                // create object of shared resource
                Callme callmeObj = new Callme();
                ThreadSyncronize t1 = new ThreadSyncronize(callmeObj, "One");
                ThreadSyncronize t2 = new ThreadSyncronize(callmeObj, "Two");
                Thread thread1 = new Thread(t1);
                Thread thread2 = new Thread(t2);

                thread1.start();
                thread2.start();
        }
}
```

*Thread Synchronization Using **Join***

When you call `join()` on a thread object (like thread.join()), the thread that calls it (caller thread, for example, the main thread) will pause and wait until the specified thread finishes running.
That is caller thread wait untill thread object finishes its execution.This is rather a serialization metod rather than synchronization.

```
class Sprint extends Thread{
        int speed;
        Sprint(String name, int speed) {
                super(name);
                this.speed = speed;
        }
        public void run() {
                int meter = 0;
                while(meter < 5) {
                        System.out.println(this.getName() +
                                " runs " + meter + " meter");
                        meter ++;
                        try {
                                Thread.sleep(speed);
                        } catch(Exception e) {
                                System.out.println(e.getMessage());
                        }
                }
                System.out.println(this.getName() + " Finished ....!");
        }
}
public class SprintJoinTest {
        public static void main(String[] str) {
                Sprint bolt = new Sprint("Bolt", 1000);
                Sprint lewis = new Sprint("Lewis", 1200);
                Sprint gatlin = new Sprint("Gatlin", 1200);
                System.out.println("5 meter sprint...!");
                System.out.println("Bolt  starts...!");
                bolt.start();
                try {
                        bolt.join();
                        // main() wait here untill bolt thread finishes
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
                System.out.println("Lewis  starts...!");
                lewis.start();
                try {
                        lewis.join();
                        // main() wait here untill lewis thread finishes
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
                System.out.println("Gatlin  starts...!")
                gatlin.start();
        }
}
```

Output:

5 meter sprint...!
Bolt starts...!
Bolt runs 0 meter
Bolt runs 1 meter
Bolt runs 2 meter
Bolt runs 3 meter
Bolt runs 4 meter
Bolt Finished ....!
Lewis starts...!
Lewis runs 0 meter
Lewis runs 1 meter
Lewis runs 2 meter
Lewis runs 3 meter
Lewis runs 4 meter
Lewis Finished ....!
Gatlin starts...!
Gatlin runs 0 meter
Gatlin runs 1 meter
Gatlin runs 2 meter
Gatlin runs 3 meter
Gatlin runs 4 meter
Gatlin Finished ....!

Suspending, Resuming, and Stopping Threads

Suspending Threads:

Deprecated Approach: The suspend() method was part of the Thread class but is deprecated due to the risk of deadlock.

Alternative: Use custom flags (boolean variables) to pause a thread safely. The thread can periodically check this flag and suspend its execution.

2. Resuming Threads:

Deprecated Approach: The resume() method was also deprecated along with suspend().

Alternative: The same flag used to suspend the thread can be checked in a loop. When the flag is reset (indicating the thread should resume), the thread can continue its execution.

3. **Stopping Threads:**

Deprecated Approach: The stop() method is deprecated because it forces the thread to terminate, leading to potential resource leaks.

Alternative: Use a shared flag to signal the thread to finish its task and exit safely. The thread should check this flag regularly to decide when to terminate itself.

Question 2
Simulate a 100-meter race between turtle and rabbit:

- rabbit runs a faster speed than turtle(i.e., time to cover 1 meter varies).
- The progress of each runner is displayed as they move forward.
- Turtle go ahead faster, and sleep for a while and wakseup resume the race
- But tutle continue in a slow pace and finishes first.
- The race ends when all runners finish
- Assumption : lower speed means faster

```
public class Rabbit extends Thread {
        Rabbit(String name) {
                super(name);
        }
        public void run() {
                int count = 0;
                while(count <= 50) {
                        System.out.println(this.getName() +
                                        "  meter = " + count++);
                        try {
                                Thread.sleep(500);
                        } catch (InterruptedException e) {
                        }
                }
                System.out.println(this.getName() +
                                "  Finished the race.....!");
        }
}
```

```
public class Tortoise extends Thread {
        Tortoise(String name) {
                super(name);
        }
        public void run() {
                int count = 0;
                while(count <= 50) {
                        System.out.println(this.getName() +
                                        "  meter = " + count++);
                        try {
                                Thread.sleep(700);
                        } catch (InterruptedException e) {
                        }
                }
                System.out.println(this.getName() +
                                "  Finished the race.....!");
        }
}
public class Race  {
        public static void main(String[] args) {
                Rabbit rabbit = new Rabbit("Oswald Rabbit");
                Tortoise tortiose = new Tortoise("Toby Turtle");

                System.out.println("race started....!");
                tortiose.start();
                rabbit.start();
                // race continue for 20 sec
                try {
                        Thread.sleep(20000);
                } catch (InterruptedException e) {}
                //rabit thread sleeps for 12 second
                rabbit.suspend();
                System.out.println("Rabbit Sleeps for a while....!");
                try {
                        Thread.sleep(12000);
                } catch (InterruptedException e) {}
                //Rabbit Wakes up and  continue race....!
                System.out.println("Rabbit Wakes up and  continue race....!");
                rabbit.resume();
        }
}
```

Output

```
race started....!
Toby Turtle runs meters = 0
Oswald Rabbit runs meters = 0
Oswald Rabbit runs meters = 1
Toby Turtle runs meters = 1
...
Oswald Rabbit runs meters = 15
Toby Turtle runs meters = 11
```

...
Oswald Rabbit runs meters = 28
Toby Turtle runs meters = 21
...
Rabbit Sleeps for a while....!
...
Toby Turtle runs meters = 29
Toby Turtle runs meters = 30
...
Toby Turtle runs meters = 44
Toby Turtle runs meters = 45
Rabbit Wakes up and continue race....!
Oswald Rabbit runs meters = 40
Toby Turtle runs meters = 46
...
Toby Turtle runs meters = 50
Oswald Rabbit runs meters = 47
Toby Turtle Finished the race.....!
Oswald Rabbit runs meters = 48
Oswald Rabbit runs meters = 49
Oswald Rabbit runs meters = 50
Oswald Rabbit Finished the race.....!

The Modern Way of Suspending, Resuming, and Stopping Threads

suspend( ), resume( ) and stop( ) methods defined by Thread must not be used for new Java programs.

– These functions are deprecated(not allowed) now. Because they caused serious failures.

A thread must be designed so that the run( ) method periodically checks to determine whether that thread should suspend, resume, or stop its own execution.

– This is accomplished by establishing a flag variable that indicates the execution state of the thread.
– As long as this flag is set to "running," the run( ) method must continue to let the thread execute.
– If this variable is set to "suspend," the thread must pause.
– If it is set to "stop," the thread must terminate.

wait( ) and notify( ) methods are inherited from Object can be used to control the execution of a thread.

***– wait( )** method is invoked to suspend the execution of the thread.*
***– notify( )** to wake up the thread.*

Question 2 – Use modern way of suspend and resume, that is wait & resume
Simulate a 100-meter race between turtle and rabbit:
- rabbit runs a faster speed than turtle(i.e., time to cover 1 meter varies).
- The progress of each runner is displayed as they move forward.
- Turtle go ahead faster, and sleep for a while, later wakseup and resume the race
- But turtle continue in a slow pace and finishes first.
- The race ends when all runners finish
- Assumption : lower speed means faster

1. *Here in thread class we propose to implement my_suspend(), my_resume() and my_start() with the help of a boolean flag say, suspendFlag.*
2. *We can not override suspend(), resume() etc because they are defined as final class. Thats why we implement our own functions.*
3. *my_suspend() will set the suspend flag to **true**.*
4. *The run() function will be modified such that in every iteration it checks the flag status in a while loop. If flag status is true (suspended) the thread will call **wait()** to suspend the thread in the loop.*
5. *my_resume() function will set the susupend flag false, and issue **notify()** in this thread, so that run wil resume.*

The rabbit class has implemented these code...!

```
class Rabbit extends Thread {
        boolean suspendFlag = false;
        Rabbit(String name) {
                super(name);
        }
        public void my_start() {
                System.out.println("Sprint Started ....!");
                suspendFlag = false;
                this.start();
        }
        public void my_suspend() {
                suspendFlag = true;
        }
        public synchronized void my_resume() {
                suspendFlag = false;
                this.notify();
        }
        public void run() {
                int count = 0;
                while(count <= 50) {
                        synchronized (this) {
                                while (suspendFlag) {
                                        try {
                                                this.wait();
                                        } catch (InterruptedException e) {
                                        }
                                }
                        }
                        System.out.println(this.getName() +
                                        " runs meters = " + count++);
```

```
                        try {
                                Thread.sleep(500);
                        } catch (InterruptedException e) {
                        }
                }
                System.out.println(this.getName() +"  Finished the race.....!");
        }
}
class Tortoise extends Thread {
        Tortoise(String name) {
                super(name);
        }
        public void run() {
                int count = 0;
                while(count <= 50) {
                        System.out.println(this.getName() +
                                        " runs meters = " + count++);
                        try {
                                Thread.sleep(700);
                        } catch (InterruptedException e) {
                        }
                }

                System.out.println(this.getName() +
                                "  Finished the race.....!");
        }
}
public class RaceTest {
        public static void main(String[] args) {
                Rabbit rabbit = new Rabbit("Oswald Rabbit");
                Tortoise tortiose = new Tortoise("Toby Turtle");
                tortiose.start();
                rabbit.my_start();
                // race continue for 20 sec
                try {
                        Thread.sleep(20000);
                } catch (InterruptedException e) {}

                // after 20 seconds, rabbit sleep fro 12 seconds..!
                rabbit.my_suspend();
                System.out.println("Rabbit Sleeps for a while....!");

                try {
                        Thread.sleep(12000);
                } catch (InterruptedException e) {}

                // after 12 seconds, rabbit wake up and continue race
                System.out.println("Rabbit Wakes up and  continue race....!");
                rabbit.my_resume();
        }
}
```

University Questions

1. Discuss the methods of creating threads in Java using appropriate examples.

Ans

Method 1: by extending Thread class

Method 2: by implementing Runnable Interface.

Runnable interface has abstract function public void run()

Method1:

```
class ThreadOne extends Thread {
    ThreadOne() {
        super("sprinter-alice");
        System.out.println("Child thread: " + this);
    }
    public void run() {
        int meter = 0;
        while(meter < 100) {
            System.out.println("Thrad Name = " + this.getName()
                    + " runs " + meter++ + " meter ");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Child thread .. exiting");
    }
}
public class ThreadOneTest {
    public static void main(String[] args) {
        System.out.println("Main thread...! ");
        ThreadOne t1 = new ThreadOne();
        // spawn new thread
        t1.start();
    }
}
```

Method2:

```
class ThreadOne implements  Runnable {
    ThreadOne() {
        System.out.println("Child thread: " + this);
    }
    public void run() {
        int meter = 0;
        String threadName = Thread.currentThread().getName();
        while(meter < 100) {
            System.out.println("Thrad Name = " + threadName
            + " runs " + meter++ + " meter ");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

```
}
public class ThreadOne_Runnable {
        public static void main(String[] args) {
                System.out.println("Main thread...!
                ThreadOne threadRunnable = new ThreadOne();
                Thread t = new Thread(threadRunnable. "sprinter-alice")
                t.start();
        }
}
```

2. Write a Java program that creates multiple child threads to print odd and even numbers from 50-100

Ans:

```
        class EvenThread extends Thread {
           public void run() {
              System.out.println("Even numbers from 50 to 100:");
              for (int i = 50; i <= 100; i++) {
                 if (i % 2 == 0) {
                    System.out.println(i);
                 }
              }
           }
        }

        class OddThread extends Thread {
           public void run() {
              System.out.println("Odd numbers from 50 to 100:");
              for (int i = 50; i <= 100; i++) {
                 if (i % 2 != 0) {
                    System.out.println(i);
                 }
              }
           }
        }
        public class PrintNumbers {

                public static void main(String[] args) {
                        EvenThread even =  new EvenThread();
                        OddThread odd = new OddThread();
                        even.start();
                        odd.start();
                }
        }
```

3. Write a Java program that creates multiple child threads to print odd and even numbers from 50-100 in seperate lines. .. like

Even numbers from 50 to 100:
50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94 96 98 100
Odd numbers from 50 to 100:
51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91 93 95 97 99

Ans:

```
        class EvenThread extends Thread {
```

```
    public void run() {
        System.out.println("Even numbers from 50 to 100:");
        for (int i = 50; i <= 100; i++) {
            if (i % 2 == 0) {
                System.out.print(i + " ");
            }
        }
    }
}
class OddThread extends Thread {
    public void run() {
        System.out.println("\nOdd numbers from 50 to 100:");
        for (int i = 50; i <= 100; i++) {
            if (i % 2 != 0) {
                System.out.print(i + " ");
            }
        }
    }
}
public class PrintNumbers {

        public static void main(String[] args) {
                EvenThread even =  new EvenThread();
                OddThread odd = new OddThread();
                even.start();
                try {
                        even.join();
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
                odd.start();
        }
}
```

4. What is Thread Synchronization? With an example illustrate the working of any one technique used for Thread synchronization in Java.

Ans:

1- using "sychronized method"

2- using "synchronized statemen"

Refer ***Thread Synchronization*** *in this document*

5. What are thread priorities? How can you assign priority values for threads(3) created in Java?

Ans:

1. In Java, thread priorities are used to indicate the importance of a thread in relation to other threads.
2. The priority of a thread can influence the order in which threads are scheduled for execution.
3. A higher-priority thread doesn't necessarily run any faster than a lower-priority thread, because Priority affects scheduling, not speed.
4. A *thread's priority is used to decide when to switch from one running thread to the next.*

Here are the different thread priorities in Java:

MIN_PRIORITY (1): The lowest priority for a thread. This is typically used for threads that perform background tasks and are not critical to the application's performance.

NORM_PRIORITY (5): The default priority assigned to threads. Most threads in a typical application use this priority.

MAX_PRIORITY (10): The highest priority for a thread. This priority is generally reserved for threads that perform time-sensitive operations or critical tasks.

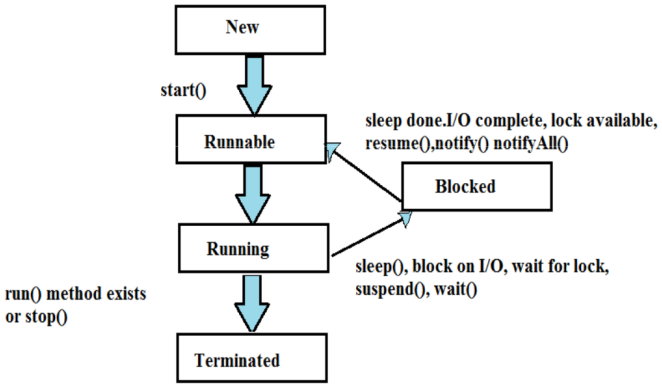
Usage:

```
Thread thread1 = new Thread(() -> { /* task */ });
thread1.setPriority(Thread.MAX_PRIORITY);
Thread thread2 = new Thread(() -> { /* task */ });
thread2.setPriority(Thread.MIN_PRIORITY);
System.out.println("Thread 1 Priority: " + thread1.getPriority());
System.out.println("Thread 2 Priority: " + thread2.getPriority());
```

6. Write a Java program to create two threads: One for displaying all odd numbers(5) between 1 and 100 and second thread for displaying all even numbers between 1 and 100.

Refer Question 2 above, Modify it to 1 instead of 50

7. Draw the lifecycle of a thread showing the different states and methods invoked.



8. Write a java program to create two threads, one for writing odd numbers and another for writing even numbers up to 100 into two different files.

import java.io.*;

```
class EvenThread extends Thread {
    public void run() {
        try {
            System.out.println("Even numbers from 50 to 100:");

            // open odd.txt to write
            BufferedWriter bw = new BufferedWriter(new
FileWriter("even_file.txt"));

            for (int i = 1; i <= 100; i++) {
                if (i % 2 == 0) {
                    System.out.print(i + " ");
                    bw.write(String.valueOf(i) + " ");
                }
            }
            bw.close();
        }catch (IOException e) {
        }
```

```
        }
    }

    class OddThread extends Thread {
        public void run() {
            try {
                System.out.println("\nOdd numbers from 50 to 100:");
                // open odd.txt to write
                BufferedWriter bw =
                    new BufferedWriter(new FileWriter("odd_file.txt"));
                for (int i = 1; i <= 100; i++) {
                    if (i % 2 == 1) {
                        System.out.print(i + " ");
                        bw.write(String.valueOf(i) + " ");
                    }
                }
                bw.close();
            }catch (IOException e) {
            }
        }
    }
    public class PrintNumbers {
        public static void main(String[] args) {
            EvenThread even =  new EvenThread();
            OddThread odd = new OddThread();
            even.start();
            odd.start();
        }
    }
```

9. What are the advantages of multi-threading in Java?

Ans:

1. Better CPU Utilization

Multithreading allows multiple tasks to run simultaneously, ensuring that idle CPU cycles are reduced. In single-threaded applications, if one task is waiting (e.g., for I/O), the entire CPU remains idle, while in a multithreaded application, other threads can use the CPU during that wait time.

2. Concurrency

Multithreading enables concurrent execution of tasks, improving responsiveness. For example, in a GUI application, one thread can handle user input while another performs background computations. This results in smoother user interaction without freezing the UI.

3. Improved Performance in I/O-Heavy Applications

In I/O-bound applications (e.g., reading from disk or network), threads can wait for I/O operations to complete without blocking the entire program. In contrast, a single-threaded application would be blocked until the I/O operation finishes.

4. Faster Task Completion

Multiple threads can divide a large task into smaller subtasks, which can be processed in parallel on multi-core systems. This can lead to faster task completion compared to single-threaded execution, where only one task is processed at a time.

5. Resource Sharing
Threads share the same memory and resources, which reduces overhead compared to multiple processes that require separate memory spaces. This is more efficient for performing related tasks that need to share data (e.g., web server handling multiple client requests).

6. Responsive Applications
Multithreading allows background tasks (e.g., data loading or computations) to be performed without blocking the main thread. For example, a web browser can download files in the background while you continue browsing.

10. Write a Java program that creates three threads. First thread generates a random positive number (>l) every I second. If the number is even, the second thread prints all even numbers between I and the generated number. If the number is odd, the third thread will print all odd numbers between 1 and the generated number

Ans:

```
import java.util.*;
class Oddthread extends Thread {
        int num;
        Oddthread(int num) {
                this.num = num;
        }
        public void run() {
                System.out.println("Priting odd numbers");
                for (int i=1; i<num; i++) {
                        if (i%2 == 1 ) {
                                System.out.print(i + " ");
                        }
                }
                System.out.println();
        }
}
class Eventhread extends Thread {
        int num;
        Eventhread(int num) {
                this.num = num;
        }
        public void run() {
                System.out.println("Priting even numbers");
                for (int i=1; i<num; i++) {
                        if (i%2 == 0 ) {
                                System.out.print(i + " ");
                        }
                }
                System.out.println();
        }
}
```

```
class FirstThread extends Thread{
	public void run() {
		Random rand = new Random();
		int digit = 0;
		Thread thred = null;
		while(true) {
			digit = rand.nextInt(50);
			System.out.println("digit = " + digit);
			if (digit %2 == 0) {
				thred = new Eventhread(digit);
			} else {
				thred = new Oddthread(digit);
			}
			thred.start();
			try {
				thred.join();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
			try {
				Thread.sleep(2000);
			} catch (InterruptedException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}
		}
	}
}
public class RandomGenTest {
	public static void main(String[] args) {
		FirstThread ft = new FirstThread();
		ft.start();
	}
}
```

Output

```
digit = 41
Priting odd numbers
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
digit = 28
Priting even numbers
2 4 6 8 10 12 14 16 18 20 22 24 26
.....
```