# Application Program Development

Segment : Threads

Presenter Name

## Agenda for Week

- Lecture
  - Java Threads
  - Synchronization in Threads
  - Threads Communication
  - Thread Monitors
- Lab
  - Part 1 : In-Lab – Design and create a GUI based Application
  - Part 2 : DIY – using the GUI designed in-lab to write events

## Outcomes

- Understanding of Threads.
- Understanding of Java Threads.
- Understanding of Thread Lifecycle.
- Examples with Java Threads and it’s management.

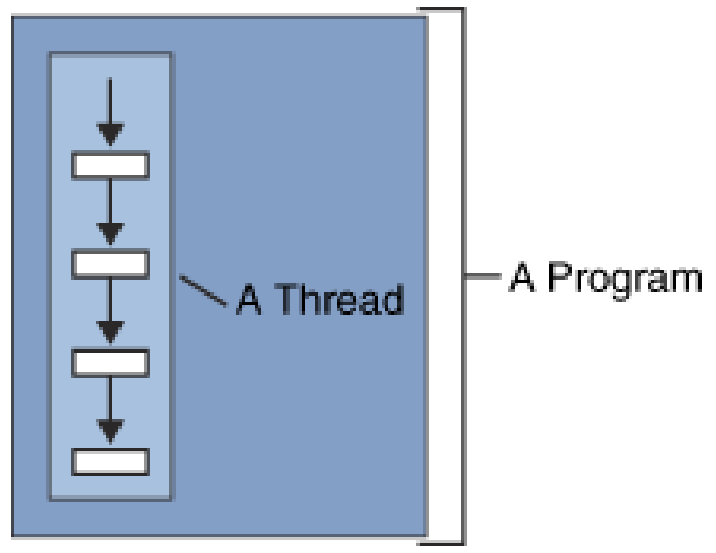
## Motivation

- Single-threaded applications can lead to poor responsiveness where lengthy activities must complete before others can begin.
- Performance is poor as the development times goes up with this.
- Poor utilization of CPU resources.
- Maintenance time cost is higher.
- Poor utilization of cache storage by not utilizing the resources properly.
- . . . . . .

## What is a Thread?

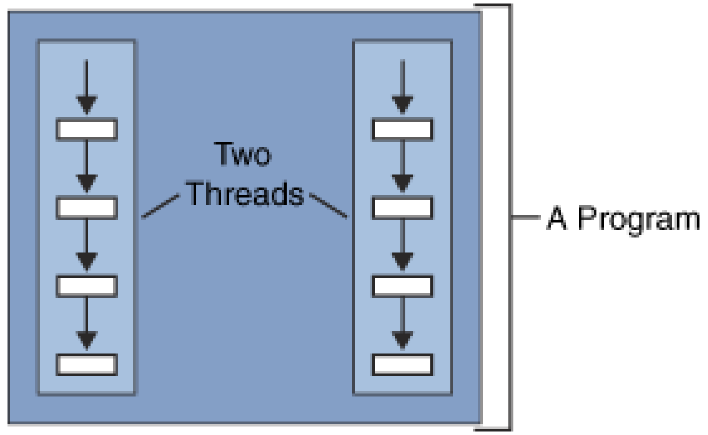
- *A thread is the flow of execution, from beginning to end, of a task.*

  - Threads organize programs into logically separate paths.
  - Thread can perform task independent of other threads (Multithreading).
  - Threads can share access to common resources.



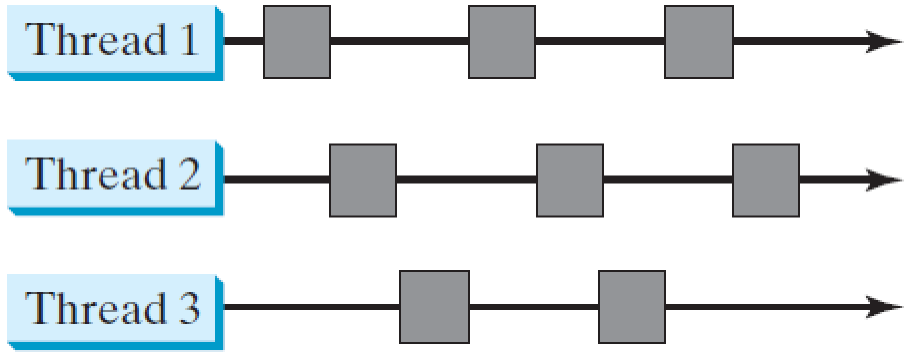
## Multi-thread

- The process of executing multiple threads simultaneously is known as multithreading.
- A simple example of multithreading would be, while you are writing a word document, your spell checker is also working and mark if you make a mistake.

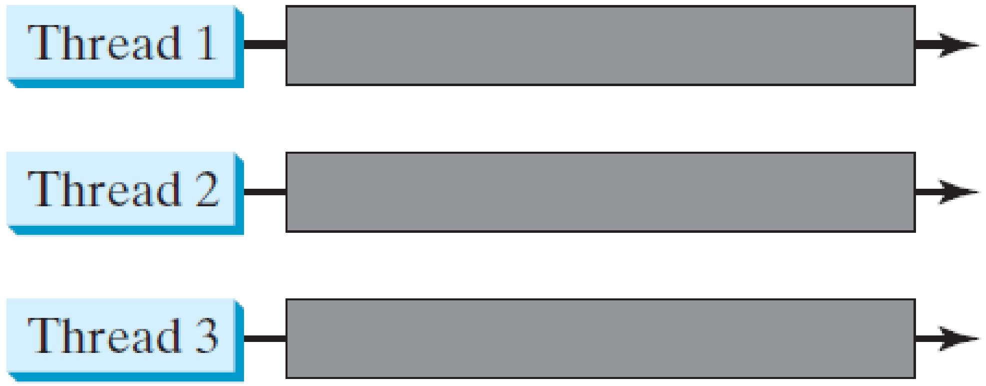


## Concept

- In a single-processor system the multiple thread share the CPU time, also known as time-sharing.
- OS is responsible for scheduling and allocating resources to them.



- In multi-processor system the multiple thread work on different processors (conceptually executing simultaneously). But practically it doesn't use multiple core at the same time, CPU makes it jump from one core to another.



## Question

- Can you choose programmatically on which core your thread is going to run?
- Yes

- Have to work bit closely to hardware level.

- Processor Affinity gives you the ability to bind or unbind a process or a thread to a CPU. This will provide you the ability to run your thread bind to a single core.

- More information can be found on Process Affinity in Java here

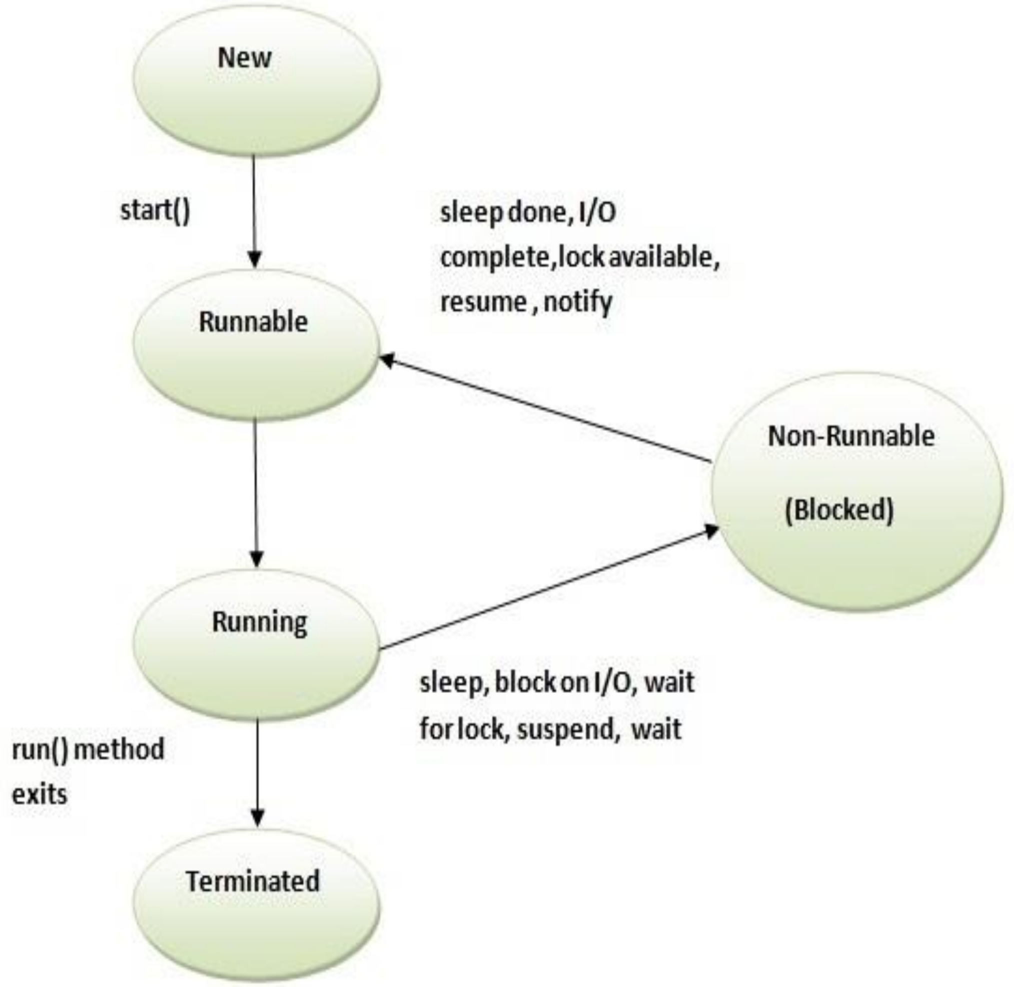
## Thread in Java

- Java provides *concurrency* available to you via language and API's
- Java can have multiple threads of executions.
- Each thread has its own method-call stack.
- A program counter.
- These two things allow java to execute thread concurrently where all threads in the application share same resources like memory and file handling etc.

# Life cycle of a thread

- The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

- New
- Runnable
- Running
- Non-Runnable (Blocked)
- Terminated


New
start()
Runnable
sleep done, I/O complete, lock available, resume, notify
Non-Runnable (Blocked)
Running
sleep, block on I/O, wait for lock, suspend, wait
run() method exits
Terminated

## 1. New

- The thread is in new state if you create an instance of Thread class but before the
- invocation of start() method.

## 2. Runnable

- The thread is in runnable state after invocation of start() method, but the thread
- scheduler has not selected it to be the running thread.

## 3. Running

- The thread is in running state if the thread scheduler has selected it.

## 4. Non-Runnable (Blocked)

- This is the state when the thread is still alive, but is currently not eligible to run.

## 5. Terminated

- A thread is in terminated or dead state when its run() method exits.

## Defining a Thread

- There are two ways to create a thread:

1. **Extend Thread Class:**

```
  public class MyThread extends Thread {

      public void run () {
     }
 }
```

- One must override **run()** method.

2. **Create a Runnable Object (Implementing the Interface):**

```
  public class MyRunnable implements Runnable {

      public void run() {
     }
 }
```

- One must implement **run()**method.

## Runnable Interface

- The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread.
- Runnable interface have only one method named run().
- Your class must override the method run and implement it.

`public void run()` is used to perform action for a thread.

# Creating and Starting a Thread

- Write a class that implements the Runnable interface. That interface has a single method called run:

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

- Place the code for your task into the run method of your class:

```
public class MyRunnable implements Runnable
{
    public void run(){
            Task statements
            . . .
    }
}
```

- Create an object of your subclass:

```
Runnable r = new MyRunnable();
```

- Construct a Thread object from the runnable object:

```
Thread t = new Thread(r);
```

- Call the start method to start the thread:

```
t.start();
```

## Thread Schedular in Java

- **Thread scheduler** in java is the part of the JVM that decides which thread should run.
  - There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.
  - Only one thread at a time can run in a single process.
  - The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.
- **Preemptive:**
  - The highest priority task executes until it enters the waiting or dead states**.**
- **Time Slicing:**
  - Task executes for a predefined slice of time and then reenters the pool of ready tasks.

## Question

- Can we start the same thread twice?
- No.
- If you do so, an *IllegalThreadStateException* is thrown.

```
public class TestThreadTwice1 implements Runnable
{
 @override
  public void run(){
   System.out.println("running...");
 }
 public static void main(String args[]){

  Thread t1 = new Thread(new
TestThreadTwice1());
  t1.start();
  t1.start();
 }
}
```

Output:

```
running
Exception in thread "main" java.lang.IllegalThreadStateException
```

## Question

- What if we call run() method directly instead start() method?
- You can call the run() method directly but then it will act as just normal method, and no multithreading will take place.

```
public class CallRunDirect implements Runnable{
  public static void main(String[] args) {
    Thread myThreadone = new Thread(new CallRunDirect());
    Thread myThreadtwo = new Thread(new CallRunDirect());

    myThreadone.run();
    myThreadtwo.run();
  }
    @Override
    public void run() {
    for(int i = 1; i < 5; i++){
    try{  Thread.sleep(500);}
    catch(InterruptedException e){ System.out.println(e); }
    System.out.println(i);
   }
  }
}
```

Output:

```
1
2
3
4
1
2
3
4
```

# Method - sleep

- The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

```
public class CallRunDirect implements Runnable{
  public static void main(String[] args) {
    Thread myThreadone = new Thread(new CallRunDirect());
    Thread myThreadtwo = new Thread(new CallRunDirect());

    myThreadone.start();
    myThreadtwo.start();
  }
    @Override
    public void run() {
    for(int i = 1; i < 5; i++){
    try{  Thread.sleep(500);} //TimeUnit.SECONDS.sleep(1);
    catch(InterruptedException e){ System.out.println(e); }
    System.out.println(i);
   }
  }
}
```

Output:

```
1
1
2
2
3
3
4
4
```

## Pausing Execution - sleep

```
public static void sleep(long millis) throws
InterruptedException

public static void sleep(long miliseconds, int
nanos)throws InterruptedException
```

- Causes the current thread to suspend execution for specified period.
- If you sleep a thread for the specified time, the thread scheduler picks up another thread and so on.
- InterruptedException is a checked Exception, which means if an **interrupt()** method (rarely used) is called then this exception will be invoked.

# pausing execution - `join`

- You can use the join() method to force one thread to wait for another thread to finish.

```
public class JoinThread implements Runnable {
        public static void main(String[] args) {
                Thread t1 = new Thread(new JoinThread());
                Thread t2 = new Thread(new JoinThread());
                Thread t3 = new Thread(new JoinThread());

                t1.start();
                try {
                        t1.join();
                }catch(InterruptedException e) {System.out.print(e);}
                t2.start();
                t3.start();
        }
        @Override
        public void run() {
                for(int i = 1; i <= 5; i++) {
                        try {
                                Thread.sleep(500);
                                }catch(InterruptedException e) {System.out.print(e);}
                        System.out.println(i);
                } } }
```

Output:

```
1
2
3
4
5
1
1
2
2
3
3
4
4
5
5
```

- getName(): You can use the method to get the default name of the thread.
- setName(String): You can use the method to set the name of the thread.
- getId(): Use the method to get the ID assigned to the thread.
- currentThread(): returns a reference to the currently executing thread object.
- isAlive(): The isAlive() method is used to find out the state of a thread. It returns true if a thread is in the Ready, Blocked, or Running state; it returns false if a thread is new and has not started or if it is finished.
- interupt(): The interrupt() method interrupts a thread in the following way: If a thread is currently in the Ready or Running state, its interrupted flag is set; if a thread is currently blocked, it is awakened and enters the Ready state, and an java.io.InterruptedException is thrown.
- isInterupt(): The isInterrupt() method tests whether the thread is interrupted.
- Thread Priority: Each thread is assigned a default priority of `Thread.NORM_PRIORITY`.
  - You can reset the priority using `setPriority(int priority)`.
  - Some constants for priorities include
    - `Thread.MIN_PRIORITY`
    - `Thread.MAX_PRIORITY`
    - `Thread.NORM_PRIORITY`

## Thread Pools

- A thread pool can be used to execute tasks efficiently.
- Suppose you have multiple threads to work with.
  - Creating each task and then starting each one of them separately can cause
    - Limited throughput
    - Poor performance
- Using a thread pool is an ideal way to manage the number of tasks executing concurrently.

## Java Executor

- *Executor* interface is being provided for executing tasks in the thread pool.
- *ExecutorService* interface is being provided for maintaining and controlling the tasks.
- *ExecutorService* interface is a sub-interface of *Executor* interface.
- Creating the executor using ExecutorService.

```
ExecutorService executor = Executors.newFixedThreadPool(1);
```

Number of threads

- What will happen if you have more tasks then the threads in the pool?

```
ExecutorService executor = Executors.newFixedThreadPool(1);
executor.execute(new PrintChar('a',50));
executor.execute(new PrintNumber(50));
executor.execute(new PrintString("Hello",50));
```

- The three runnable tasks will be executed sequentially.

- What if I do not want my threads to be fixed?

```
ExecutorService executor = Executors.newCashedThreadPool();
executor.execute(new PrintChar('a',50));
executor.execute(new PrintNumber(50));
executor.execute(new PrintString("Hello",50));
```

- New threads will be created for each waiting task and all the tasks will be running concurrently.