Multi Threading

Multi Threading

- Java is a multithreaded programming language which means we can develop multi threaded program using Java.
- Multithreaded programs contain two or more threads that can run concurrently. This means that a single program can perform two or more tasks simultaneously.
- Thread is basically a lightweight sub-process, a smallest unit of processing.
- Multiprocessing and multithreading, both are used to achieve multitasking.
- But we use multithreading than multiprocessing because threads share a common memory area.
- They don't allocate separate memory area which saves memory, and context-switching between the threads takes less time than process.

Multi Threading

- Threads are independent. So, it doesn't affect other threads if exception occurs in a single thread.
- Java Multithreading is mostly used in games, animation etc.
- For example, one thread is writing content on a file at the same time another thread is performing spelling check.
- In Multiprocessing, Each process has its own address in memory. So, each process allocates separate memory area.
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc. So that cost of communication between the processes is high.

Multi Threading

- Disadvantage:
- If you create too many threads, you can actually degrade the performance of your program rather than enhance it.
- Remember, some overhead is associated with context switching.
- If you create too many threads, more CPU time will be spent changing contexts than executing your program.

Differences between multi threading and multitasking

- Two kinds of multi-tasking:
  - process-based multi-tasking
  - thread-based multi-tasking
- Process-based multi-tasking is about allowing several programs to execute concurrently, e.g. Java compiler and a text editor.
- Processes are heavyweight tasks:
  - that require their own address space
  - inter-process communication is expensive and limited
  - context-switching from one process to another is expensive and limited

Thread-Based Multi-Tasking

- Thread-based multi-tasking is about a single program executing concurrently
- several tasks e.g. a text editor printing and spell-checking text.
- Threads are lightweight tasks:
  - they share the same address space
  - they cooperatively share the same process
  - inter-thread communication is inexpensive
  - context-switching from one thread to another is low-cost
- Java multi-tasking is thread-based.

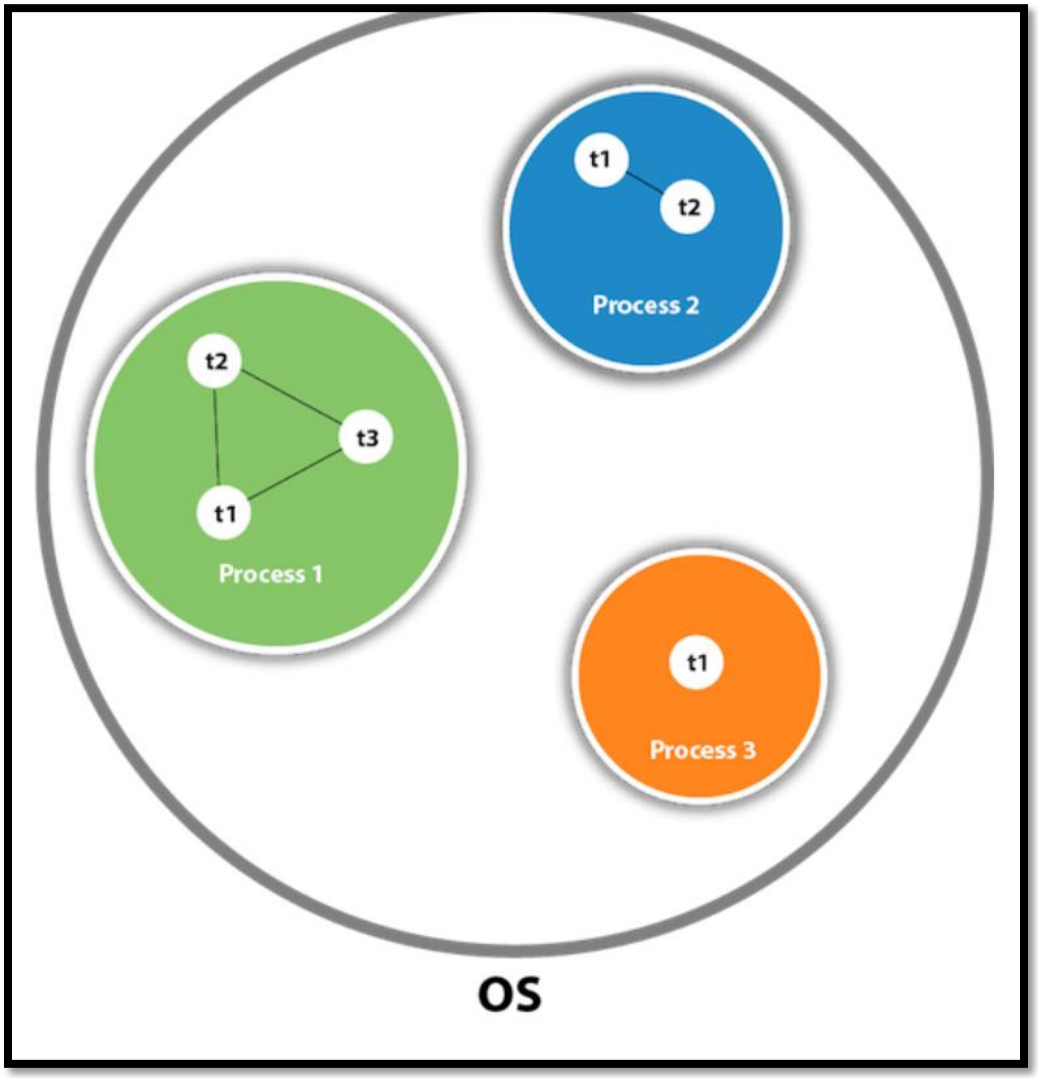
Reasons for Multi-Threading

- Multi-threading enables to write efficient programs that make the maximum use of the CPU, keeping the idle time to a minimum.
- There is plenty of idle time for interactive, networked applications:
  - the transmission rate of data over a network is much slower than the rate at which the computer can process it
  - local file system resources can be read and written at a much slower rate than can be processed by the CPU
  - of course, user input is much slower than the computer

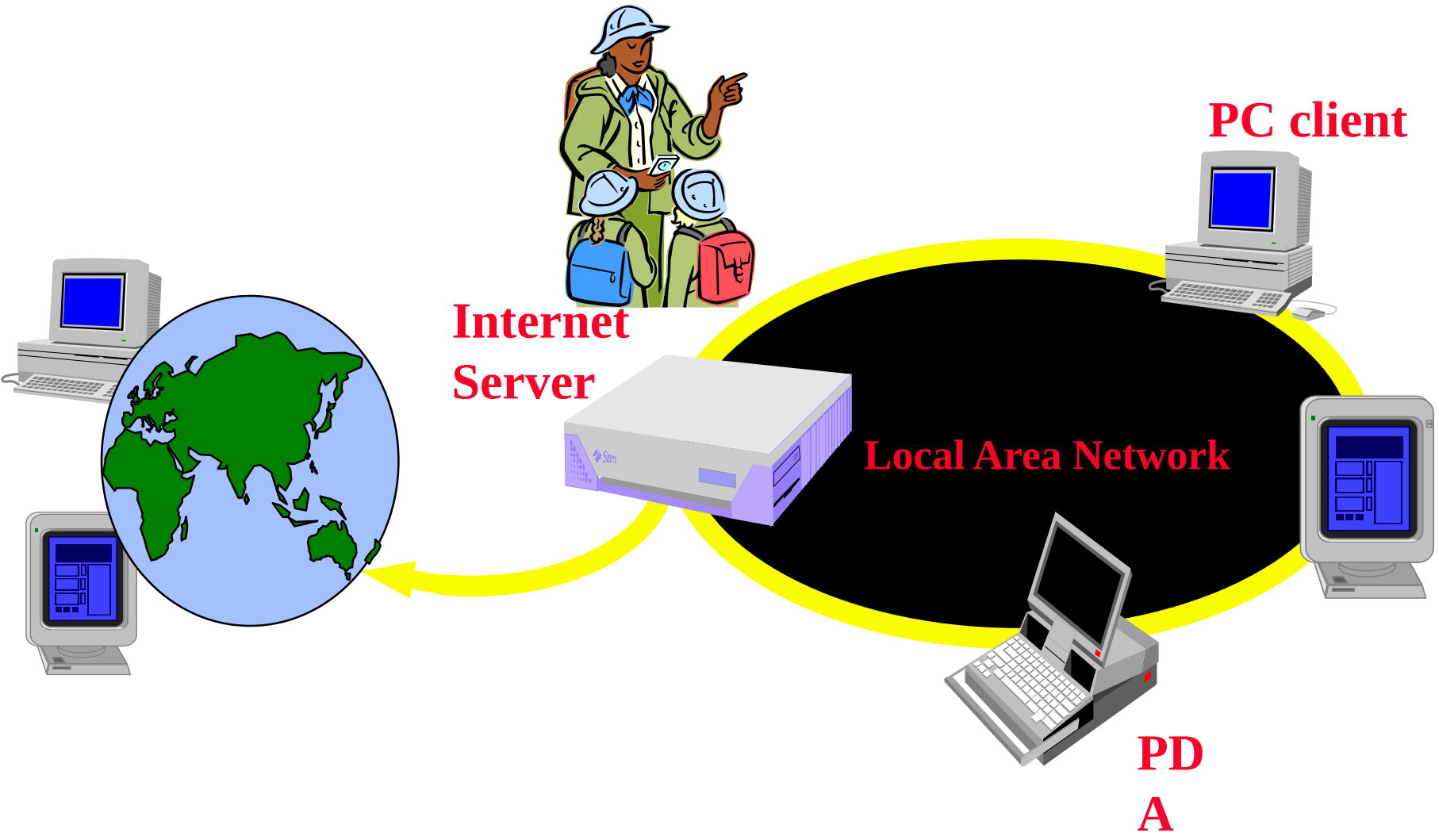
What are Threads?

- Threads are sometimes called *lightweight processes*.
- Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.
- Thread is a piece of code that run in concurrent with other threads.
- Thread is a single sequential flow of programming operations, with a definite beginning and an end.
- During the lifetime of the thread, there is only a single point of execution.
- Programming a task having multiple threads of control – Multithreading or Multithreaded Programming.

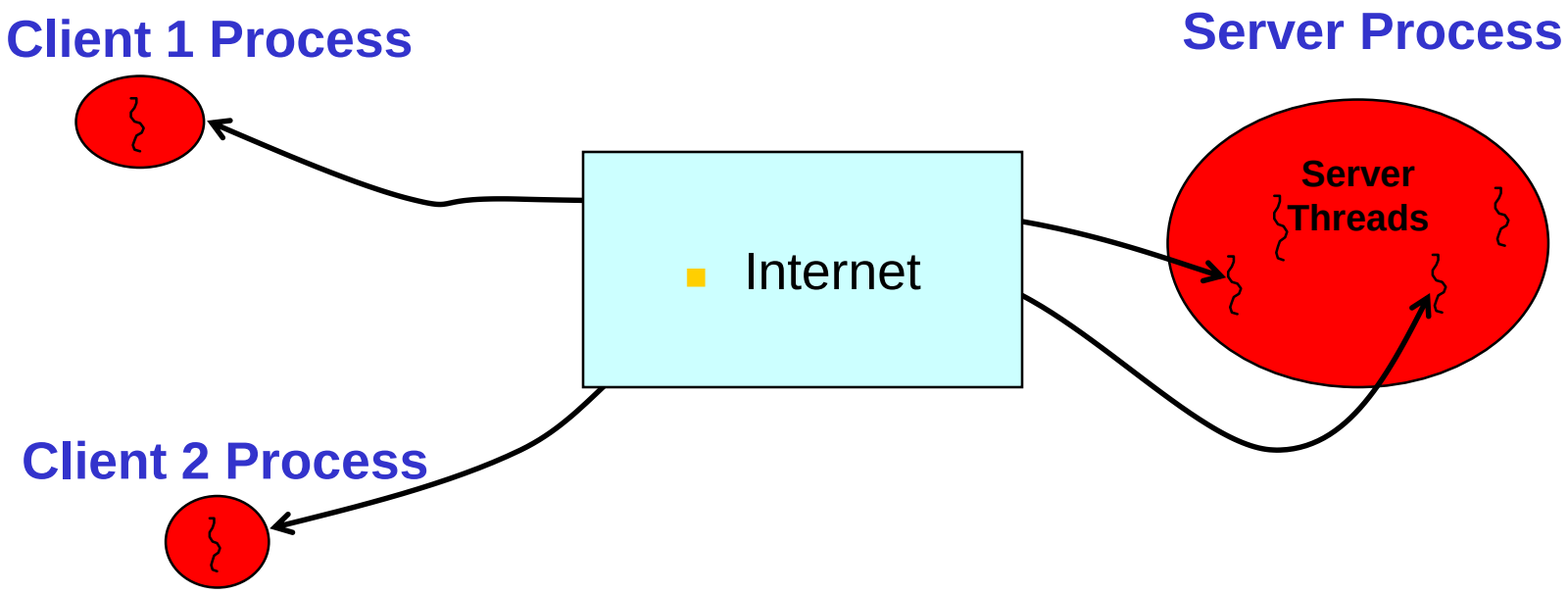
What are Threads?



Web/Internet Applications: Serving Many Users Simultaneously

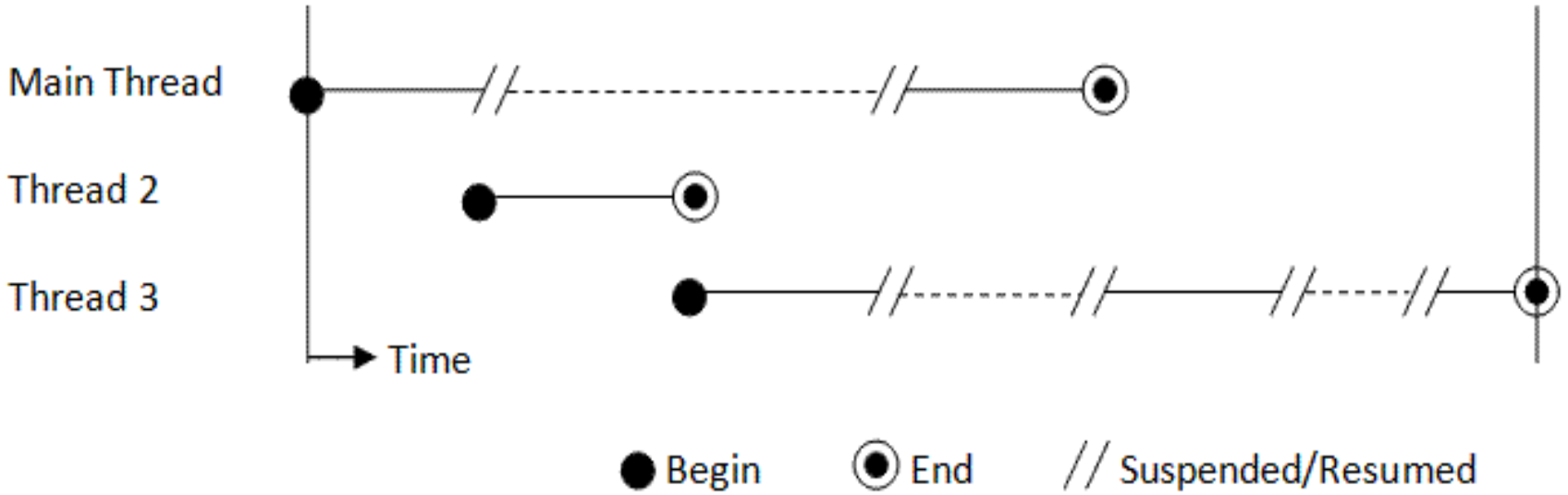


Multithreaded Server: For Serving Multiple Clients Concurrently



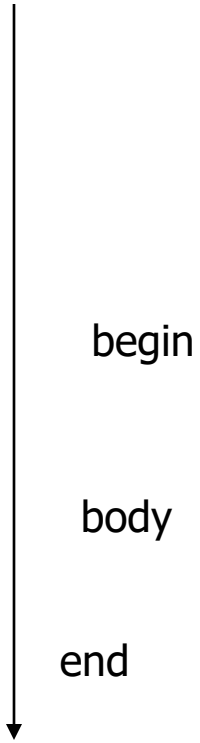
Java Threads

- Java has built in thread support for Multithreading
  - Synchronization
  - Thread Scheduling
  - Inter-Thread Communication
- A thread by itself is not a program because it cannot run on its own. Instead, it runs within a program. The following figure shows a program with 3 threads running under a single CPU:

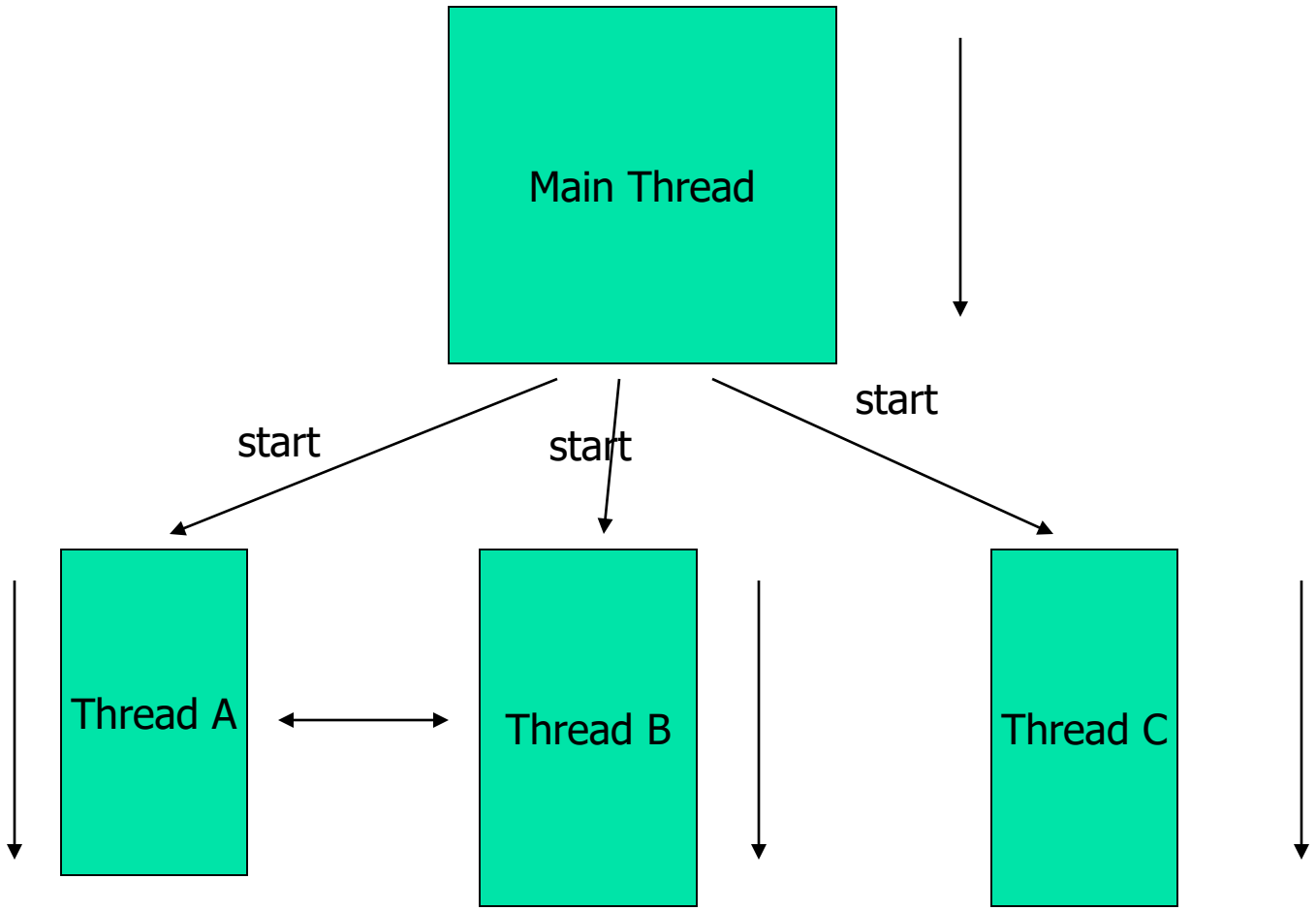


A single threaded program

```
class ABC
{
....
    public void main(..)
    {
    ...
    ..
    }
}
```



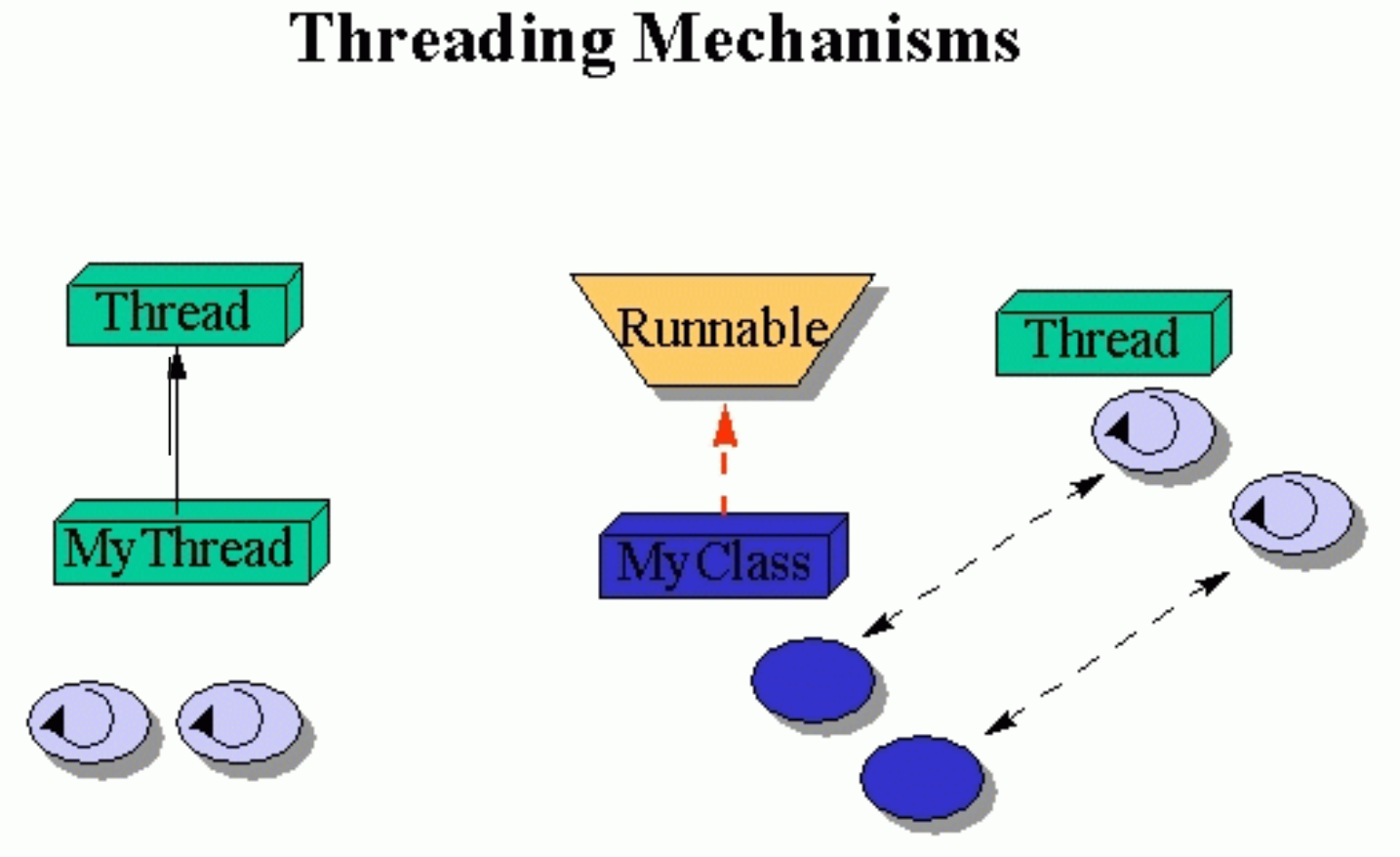
A Multithreaded Program



Threads may switch or exchange data/results

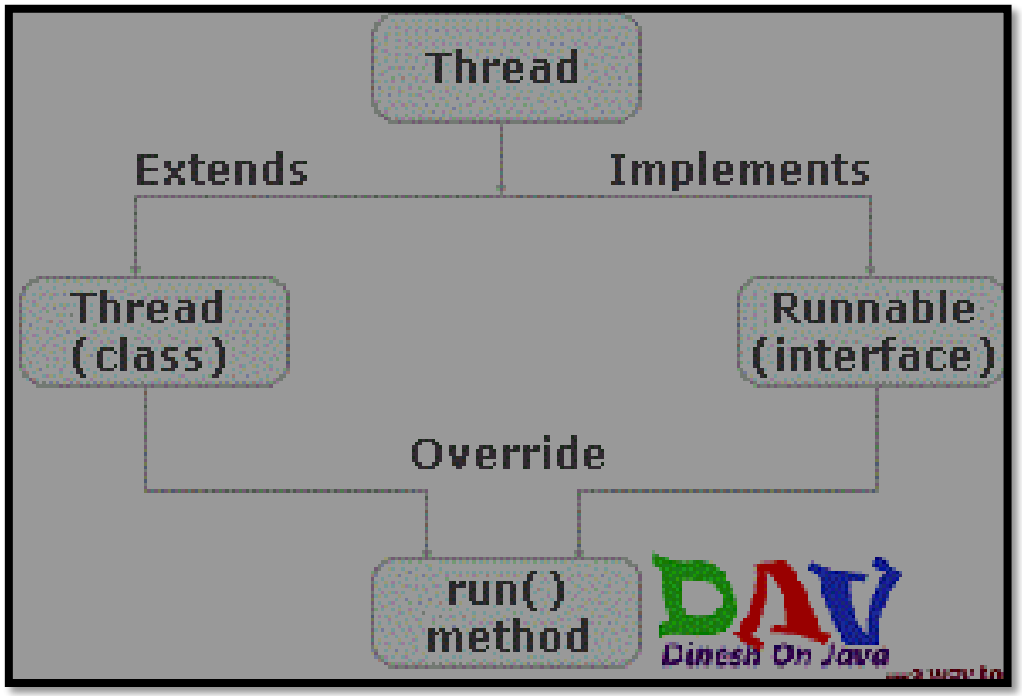
Threading Mechanisms in JAVA

- There are two ways to create a new thread in java:
  - Create a class that extends the Thread class
  - Create a class that implements the Runnable interface



Threading Mechanisms in JAVA

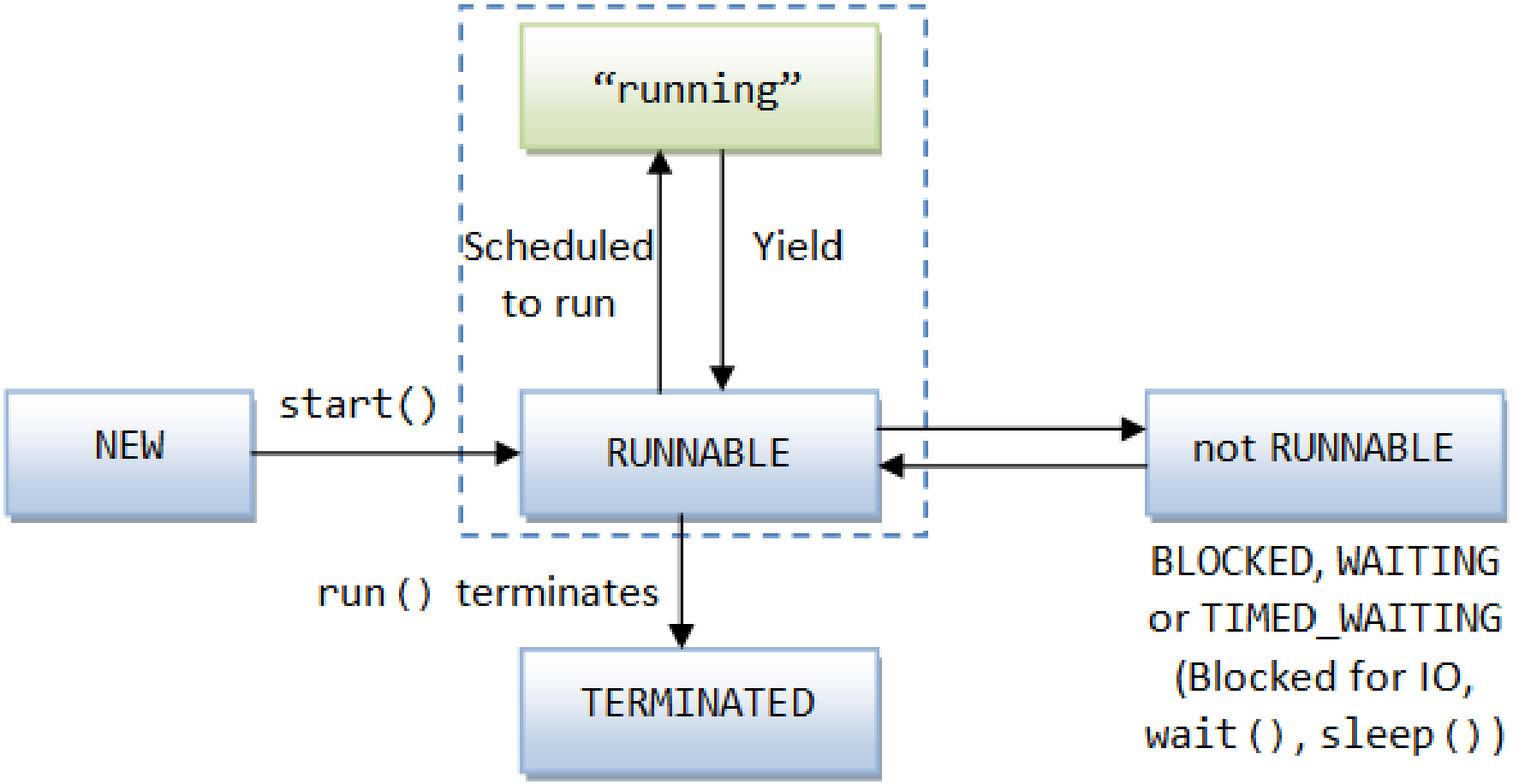
- To Execute thread , the thread first created and then start() method is invoked on the thread.
- Eventually the thread would execute and the run method would be invoked.



Thread Lifecycle

- Thread exist in several states:
  1) ready to run
  2) running
  3) a running thread can be suspended
  4) a suspended thread can be resumed
  5) a thread can be blocked when waiting for a resource
  6) a thread can be terminated
- Once terminated, a thread cannot be resumed.

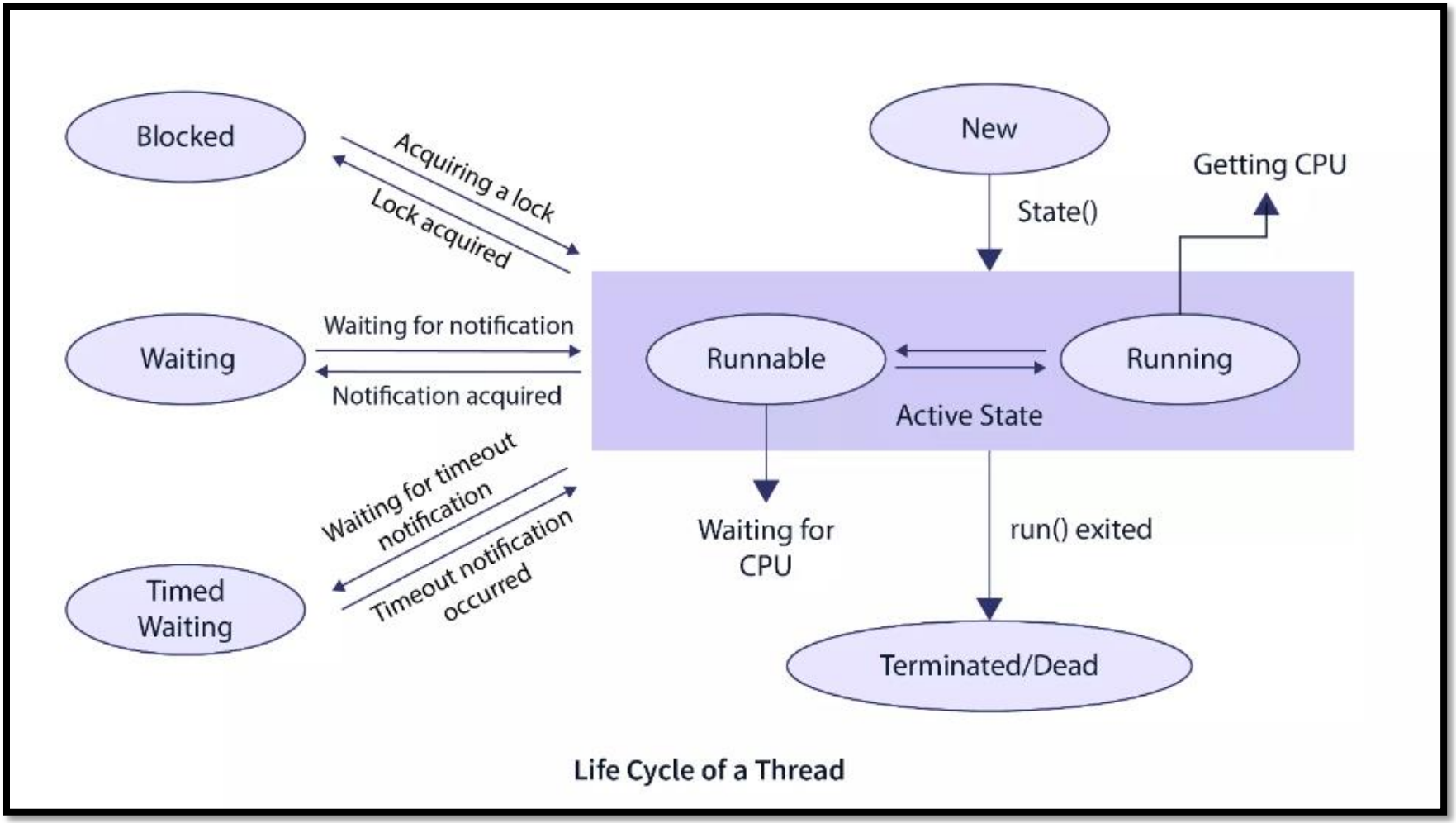
Thread Lifecycle



Thread Lifecycle

- **New state –**
  - After the creations of Thread instance the thread is in this state but before the start() method invocation. At this point, the thread is considered not alive.
- **Runnable (Ready-to-run) state –**
  - A thread start its life from Runnable state. A thread first enters runnable state after the invoking of start() method but a thread can return to this state after either running, waiting, sleeping or coming back from blocked state also. On this state a thread is waiting for a turn on the processor.
- **Running state –**
  - A thread is in running state that means the thread is currently executing. There are several ways to enter in Runnable state but there is only one way to enter in Running state: the scheduler select a thread from runnable pool.
- **Dead state –**
  - A thread can be considered dead when its run() method completes. If any thread comes on this state that means it cannot ever run again.
- **Blocked –**
  - A thread can enter in this state because of waiting the resources that are hold by another thread.

Thread Lifecycle



Class Thread & Interface Runnable

- **Interface Runnable**
  - The interface java.lang.Runnable declares one abstract method run(), which is used to specify the running behavior of the thread:
  - public void run();
- **Class Thread**
  - The class java.lang.Thread has the following constructors:
    - public **Thread**();
    - public **Thread**(String *threadName*);
    - public **Thread**(Runnable *target*);
    - public **Thread**(Runnable *target*, String *threadName*);
  - The first two constructors are used for creating a thread by sub-classing the Thread class.
  - The next two constructors are used for creating a thread with an instance of class that implements Runnable interface.

Methods in the Thread Class

1. public void start(): a thread by calling start its run method
   - Begin a new thread. JRE calls back the run() method of this class. The current thread continues.
2. public void run(): entry point for new thread
   - to specify the execution flow of the new thread. When run() completes, the thread terminates.
3. public static sleep(long millis) throws InterruptedException
4. public static sleep(long millis, int nanos) throws InterruptedException
   - Suspend the current thread and yield control to other threads for the given milliseconds (plus nanoseconds).
   - This is a static method commonly used to pause the current thread (via Thread.sleep()) so that the other threads can have a chance to execute.
5. public void interrupt()
   - You can awaken a sleep thread before the specified timing via a call to the interrupt() method.

Methods in the Thread Class

6. public static void yield()
   - hint to the scheduler that the current thread is willing to yield its current use of a processor to allow other threads to run. The scheduler is, however, free to ignore this hint. Rarely-used.
7. public boolean isAlive()
   - Return false if the thread is new or dead. Returns true if the thread is "runnable" or "not runnable".
8. public void setPriority(int p)
   - Set the priority-level of the thread, which is implementation dependent.
9. Join: wait for a thread to terminate
10. getName: obtain a thread's name

- The stop(), suspend(), and resume() methods have been deprecated in JDK 1.4,

Creating a new Thread Method-1

- To create and run a new thread by extending Thread class:
  1. Define a subclass (named or anonymous) that extends from the superclass Thread.
  2. In the subclass, override the run() method to specify the thread's operations, (and provide other implementations such as constructors, variables and methods).
  3. A client class creates an instance of this new class. This instance is called a Runnable object (because Thread class itself implements Runnable interface).
  4. The client class invokes the start() method of the Runnable object. The result is two thread running concurrently – the current thread continue after invoking the start(), and a new thread that executes run() method of the Runnable object.

Creating a new Thread Method-1

- To create and run a new thread by extending Thread class:
- to create a new class that extends Thread class and then create an instance of that class.
- The extending class must override the run( ) method, which is the entry point for the new thread.
- Once Thread object is created, you can start it by calling start( ) method, which executes a call to run( ) method.

Creating a new Thread Method-1

```
For example,
class MyThread extends Thread {
 // override the run() method
       @Override
       public void run() {
              // Thread's running behavior
       }
// constructors, other variables and methods ..... }

public class Client {
       public static void main(String[] args) {
              ..... // Start a new thread
             MyThread t1 = new MyThread();
             t1.start(); // Called back run()
             .....
             // Start another thread
             new MyThread().start();
             .....
       }
}
```

Creating a new Thread Method-1

```
class ChildClass extends Thread {
    // overriding the run() method
    public void run() {
        System.out.println("Run method of the child class.");
    }
}

public class Test {
    public static void main(String args[]) {
        // creating object of the sub class.
        ChildClass cc = new ChildClass();

        // starting the new thread execution.
        cc.start();
    }
}
```

Run method of the child class.

Creating a new Thread Method-2

- To create and run a new thread by implementing Runnable interface:
  1. Define a class that implements the Runnable interface.
  2. In the class, provide implementation to the abstract method run() to specify the thread's operations, (and provide other implementations such as constructors, variables and methods).
  3. A client class creates an instance of this new class. The instance is called a Runnable object.
  4. The client class then constructs a new Thread object with the Runnable object as argument to the constructor.
  5. invokes the start() method. The start() called back the run() in the Runnable object (instead of the Thread class).

Creating a new Thread Method-2

- After implementing runnable interface, the class needs to implement the run() method, which has following form:
- **public void run( )**
- This method provides entry point for the thread and you will put you complete business logic inside this method.
- After that, you will instantiate a Thread object using the following constructor:
- **Thread (Runnable threadObj, String threadName) ;**
- Where, threadObj is an instance of a class that implements the Runnable interface and threadName is the name given to the new thread.
- Once Thread object is created, you can start it by calling start( ) method, which executes a call to run( ) method.
- **Void start()**

Creating a new Thread Method-2

```
class MyRunnable implements Runnable {
// provide implementation to abstract method run()
	@Override
	public void run() {
		// Thread's running behavior
	}
	 ..... // constructors, other variables and methods
}


public class Client {
	.....
	Thread t = new Thread(new MyRunnable());
	t.start();
	...
}
```

Creating a new Thread Method-2

```
class ImplementingClass implements Runnable {
    // overriding the run() method
    public void run() {
        System.out.println("Run method of the implementing class.");
    }
}

public class Test {
    public static void main(String args[]) {
        // creating object of the implementing class.
        ImplementingClass ic = new ImplementingClass();

        // passing the reference to the Thread class constructor.
        Thread t = new Thread(ic);

        // starting the new thread execution.
        t.start();
    }
}
```

Run method of the child class.

Creating multiple threads in JAVA

Using Extend thread class

- Extend Thread class
- Override its run() method
- Instantiate the thread subclass and call its start() method.

Syntax:

```
class MyThread extends Thread
{
        Public void run()
        {
        // code to be executed in new thread
        }
}
//Create and start multiple threads
MyThread t1=new MyThread();
t1.start();
MyThread t2=new MyThread();
t2.start();
```

Creating multiple threads in JAVA

```
public class MyThread extends Thread
{
String task;    // Declare a String variable to represent task.
MyThread(String task)
{
  this.task = task;
}
public void run()
{
 for(int i = 1; i <= 5; i++)
 {
  System.out.println(task+ " : " +i);
  try
 { Thread.sleep(1000); } // Pause the thread execution for 1000 milliseconds.
 catch(InterruptedException ie) {
   System.out.println(ie.getMessage()); }
 } } // end of for loop // end of run() method.
public static void main(String[] args)
{
// Create two objects to represent two tasks.
  MyThread th1 = new MyThread("Cut the ticket"); // Passing task as an argument to its constructor.
  MyThread th2 = new MyThread("Show your seat number");
// Create two objects of Thread class and pass two objects as parameter to constructor of Thread class.
  Thread t1 = new Thread(th1);
  Thread t2 = new Thread(th2);
  t1.start();
  t2.start();
  }
}
```

Output:

```
Cut the ticket : 1
Show your seat number : 1
Show your seat number : 2
Cut the ticket : 2
Show your seat number : 3
Cut the ticket : 3
Show your seat number : 4
Cut the ticket : 4
Show your seat number : 5
Cut the ticket : 5
```

Creating multiple threads in JAVA

Using Implement Runnable

- Implement Runnable Interface
- Override its run() method with code you want to execute
- Instantiate the thread Object and pass to its constructor.
- Call start() method.

Syntax:

```
class MyRunnable implements Runnable{
public void run(){
// code to be executed in new thread
}}
//Create and start multiple threads
MyRunnable R1=new MyRunnable();
Thread t1=new Thread(R1);
t1.start();
MyRunnable R2=new MyRunnable();
Thread t2=new Thread(R2);
t2.start();
```

Creating multiple threads in JAVA

```
public class MyThread1 implements Runnable
{
Thread t;
MyThread1(String s)
{
t=new Thread(this,s);
t.start();
}
public void run()
{
 for(int i = 1; i <= 5; i++)
 {
  System.out.println(Thread Name+ " : " +Thread.currentThread().getName());
  try
 { Thread.sleep(1000); } // Pause the thread execution for 1000 milliseconds.
 catch(InterruptedException ie) {
   System.out.println(ie.getMessage()); }
 } } // end of for loop // end of run() method.
}
Public class student{
public static void main(String[] args)
{
System.out.println("Thread Name:"+Thread.CurrentThread().getName());
// Create two objects to represent two tasks/threads.
  MyThread1 th1 = new MyThread1("My Thread1"); // Passing task as an argument to its constructor.
  MyThread1 th2 = new MyThread("My Thread2");
}
}
```

```
Thread Name: main
Thread Name: My Thread 1
Thread Name: My Thread 2
Thread Name: My Thread 1
Thread Name: My Thread 2
Thread Name: My Thread 1
Thread Name: My Thread 2
Thread Name: My Thread 1
Thread Name: My Thread 2
Thread Name: My Thread 1
Thread Name: My Thread 2
```

Method-1 Example

```
public class MyThread extends Thread {
	private String name;
	public MyThread(String name) { // constructor
		this.name = name;
	}
	// Override the run() method to specify the thread's
running behavior
	@Override
	public void run() {
		for (int i = 1; i <= 5; ++i) {
			System.out.println(name + ": " + i);
			yield();
		}
	}
}

public class TestMyThread {
	public static void main(String[] args) {
		Thread t1 = new MyThread("Thread 1")
		Thread t2 = new MyThread("Thread 2“)
		Thread t3 = new MyThread("Thread 3“)
		t1.start();
		t2.start();
		t1.start();
	}
}
```

Output

- The test class allocates and starts three threads. The output is as follows:

```
Thread 1: 1
Thread 3: 1
Thread 1: 2
Thread 2: 1
Thread 1: 3
Thread 3: 2
Thread 2: 2
Thread 3: 3
Thread 1: 4
Thread 1: 5
Thread 3: 4
Thread 3: 5
Thread 2: 3
Thread 2: 4
Thread 2: 5
```

New Thread: Extend Thread class

- The second way to create a new thread:

  1) create a new class that extends Thread

  2) create an instance of that class

- Thread provides both run and start methods:

  1) the extending class must override run

  2) it must also call the start method

Example3: New Thread

```
//A class NewThread that extend Thread class:
class NewThread extends Thread {
    /*Create a new thread by calling the Thread's constructor and start
      method: */
   NewThread(){
        super("Demo Thread");
        System.out.println("Child thread: " + this);
        start();
   }
/*This is the entry point for the newly created thread – a five-
   iterations loop with a half-second pause between the iterations all
   within try/catch: */
public void run() {
   try {
        for (int i = 5; i > 0; i--) {
               System.out.println("Child Thread: " + i);
               Thread.sleep(500);
        }
   } catch (InterruptedException e) {
        System.out.println("Child interrupted.");
   }
   System.out.println("Exiting child thread.");
   }  }
```

Example3: New Thread

```
class ExtendThread {
   public static void main(String args[]) {
//A new thread is created as an object of NewThread:
   new NewThread();
//After calling the NewThread start method,control returns here.
//Both threads (new and main) continue concurrently.
//Here is the loop for the main thread:
   try {
        for (int i = 5; i > 0; i--) {
               System.out.println("Main Thread: " + i);
               Thread.sleep(1000);
        }
        } catch (InterruptedException e) {
               System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
   }
}
```

Thread Priorities

- Every Java thread has a **priority** that helps the operating system determine the **order in which threads are scheduled**.
- Java priorities are in the range between **MIN_PRIORITY** (a constant of 1) and **MAX_PRIORITY** (a constant of 10).
- By default, every thread is given priority **NORM_PRIORITY** (a constant of 5).
- Threads with **higher** priority are more important to a program and should be allocated **processor time** before **lower**-priority threads.
- The thread scheduler mainly uses **preemptive** or **time slicing** scheduling to schedule the threads.

Thread Priorities

Constant	Description
Thread.MIN_PRIORITY	The Minimum Priority of any thread(01)
Thread.MAX_PRIORITY	The Maximum Priority of any thread(10)
Thread.NORM_PRIORITY	The normal Priority of any thread(05)

- Methods of Thread Priority

Method	Description
SetPriority()	To set the Priority of thread
getPriority()	To get the Priority of thread

Thread Priorities

```
class MyThread extends Thread
{
        public void run()
        {
                System.out.println("Thread Running...");
        }

        public static void main(String[]args)
        {
                MyThread p1 = new MyThread();
                MyThread p2 = new MyThread();
                MyThread p3 = new MyThread();
                p1.start();
                System.out.println("P1 thread priority : " + p1.getPriority());
                System.out.println("P2 thread priority : " + p2.getPriority());
                System.out.println("P3 thread priority : " + p3.getPriority());
        }
}
```

P1 thread priority : 5
Thread Running...
P2 thread priority : 5
P3 thread priority : 5

Thread Priorities

```
class MyThread extends Thread
{
        public void run()
        {
                System.out.println("Thread Running...");
        }

        public static void main(String[]args)
        {
                MyThread p1 = new MyThread();
                p1.start();
                System.out.println("max thread priority : " + p1.MAX_PRIORITY);
                System.out.println("min thread priority : " + p1.MIN_PRIORITY);
                System.out.println("normal thread priority : " + p1.NORM_PRIORITY);
        }
}
```

Output:
Thread Running...
max thread priority : 10
min thread priority : 1
normal thread priority : 5

Thread Priorities

```
class MyThread extends Thread
{
        public void run()
        {
                System.out.println("Thread Running... "+Thread.currentThread().getName());
        }

        public static void main(String[]args)
        {
                MyThread p1 = new MyThread();
                MyThread p2 = new MyThread();
                // Starting thread
                p1.start();
                p2.start();
                // Setting priority
                p1.setPriority(2);
                // Getting -priority
                p2.setPriority(1);
                int p = p1.getPriority();
                int p22 = p2.getPriority();
                System.out.println("first thread priority : " + p);
                System.out.println("second thread priority : " + p22);

        }
}
```

Thread Running... Thread-0
first thread priority : 5
second thread priority : 1
Thread Running... Thread-1

Example4: Thread Priorities

```
class TestMultiPriority1 extends Thread{
 public void run(){
   System.out.println("running thread name is:"+Thread.currentThread().
   getName());
   System.out.println("running thread priority is:"+Thread.currentThrea
   d().getPriority());
  }

 public static void main(String args[]){
  TestMultiPriority1 m1=new TestMultiPriority1();
  TestMultiPriority1 m2=new TestMultiPriority1();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
  m1.start();
  m2.start();
 }
}
```

Threads: Synchronization

- Multi-threading introduces asynchronous behavior to a program.
- How to ensure synchronous behavior when we need it?
- For instance, how to prevent two threads from simultaneously writing and reading the same object?
- Java implementation of monitors:
  - classes can define so-called synchronized methods
  - each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called
  - once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object

Threads: Synchronization

- It refers to the concept where only one thread is executed at a time while other threads are in the waiting state. This process is called thread synchronization.
- It is used because it avoids interference of thread and the problem of inconsistency.
- In java, thread synchronization is further divided into two types:
- Mutual exclusive- it will keep the threads from interfering with each other while sharing any resources.
- Inter-thread communication- It is a mechanism in java in which a thread running in the critical section is paused and another thread is allowed to enter or lock the same critical section that is executed.

Threads: Synchronization

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.
- The process by which this synchronization achieved is called thread synchronization.
- The synchronized keyword in Java creates a block of code referred to as a critical section.
- Every Java object with a critical section of code gets a lock associated with the object.
- To enter a critical section, a thread needs to obtain the corresponding object's lock.

Threads: Synchronization

Syntax:

```
synchronized(object)
{
// statements to be synchronized
}
```

- Here, object is a reference to the object being synchronized.
- A synchronized block ensures that a call to a method that is a member of object occurs only after the current thread has successfully entered object's critical section.
- If a block is declared as synchronized then the code which is written inside a method is only executed instead of the whole code. It is used when sequential access to code is required.

Threads: Synchronization

```
public class Table
{
// non synchronized method
void printTable(int n)
{
for(int i = 1; i <= 5; i++)
{
 System.out.println(n * i);
 try
 {
   Thread.sleep(500);
 }
catch(InterruptedException ie)
{
 System.out.println(ie);
 }
 }}
}
```

```
public class Thread1 extends
Thread
{
Table t;
// Declaring t as class type table
// Declaring parameterized
constructor and passing variable t
as a parameter to the thread.
Thread1(Table t)
{
 this.t = t;
}
public void run()
{
 t.printTable(2);
 }
}
```

Threads: Synchronization

```
public class Thread2 extends
Thread
{
Table t;
// Declaring t as class type
table
Thread2(Table t)
{
 this.t = t;
}
public void run()
{
  t.printTable(10);
 }
}
```

```
public class UnsynchronizedMethod
{
public static void main(String[]
args)
{
// Creating an object of Table
class.
    Table obj = new Table();
    Thread1 t1 = new Thread1(obj);
    Thread2 t2 = new Thread2(obj);
      t1.start();
      t2.start();
 }
}
```

Output:

2
10
4
20
30
6
40
8
50
10

Threads: Synchronization

```
class Table
{
 void printTable(int n){
//This is a synchronized block
   synchronized(this){
     for(int i=1;i<=5;i++){
      System.out.println(n*i);
      try{
       Thread.sleep(400);
      }
catch(Exception e)
{ System.out.println(e);}
     }
   }
 }
//end of the synchronized block
}
```

```
public class Thread1 extends
Thread
{
Table t;
// Declaring t as class type table
Thread1(Table t)
{
 this.t = t;
}
public void run()
{
 t.printTable(5);
 }
}
```

Threads: Synchronization

```
public class Thread2 extends
Thread
{
Table t;
// Declaring t as class type
table
Thread2(Table t)
{
 this.t = t;
}
public void run()
{
  t.printTable(100);
 }
}
```

```
public class
TestSynchronizedBlock1
{
public static void main(String
args[])
{
Table obj = new Table();
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

Output:

5
10
15
20
25
100
200
300
400
500

Inter-thread Communication

- Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.
- Inter-thread communication is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.
- To avoid polling(It is usually implemented by loop), Inter-thread communication is implemented by following methods of Object class:
- **wait()**
- **notify()**
- **notify all()**

Inter-thread Communication

wait( ):

- This method tells the calling thread to give up the critical section and go to sleep until some other thread enters the same critical section and calls notify( ).

notify( ):

- This method wakes up the first thread that called wait( ) on the same object.
- The notify() method wakes up a single thread that is waiting on this object's monitor.

notifyAll( ):

- This method wakes up all the threads that called wait( ) on the same object. The highest priority thread will run first.
- Above all methods are implemented as final in Object class.
- All three methods can be called only from within a synchronized context.

Inter-thread Communication

wait()	sleep()
The wait() method releases the lock.	The sleep() method doesn't release the lock.
It is a method of Object class	It is a method of Thread class
It is the non-static method	It is the static method
It should be notified by notify() or notifyAll() methods	After the specified amount of time, sleep is completed.

Inter-thread communication

```
public class A
{ int i;
 boolean flag = false;
synchronized void deliver(int i)
{
 if(flag)
 try
 { wait();
// Wait till a notification is
received from Thread2.
 }
 catch(InterruptedException ie)
 { System.out.println(ie);}
   this.i = i;
   flag = true;
 // When data production is over, it will store
true into flag.
   System.out.println("Data Delivered:
" +i);
   notify();
 // When data production is over, it will
notify Thread2 to use it.
 }
```

```
synchronized int receive()
{
if(!flag)
try {
 wait();
// Wait till a notification is received
from Thread1.
}
catch(InterruptedException ie){
 System.out.println(ie);
}
 System.out.println("Data
Received: " + I);
  flag = false;
// It will store false into flag when
data is received.
  notify();
// When data received is over, it will
notify Thread1 to produce next data.
  return i;
 } }
```

Inter-thread communication

```
public class Thread1 extends Thread
{
 A obj;
 Thread1(A obj)
 {
  this.obj = obj;
 }
public void run()
{
for(int j = 1; j <= 5; j++){
 obj.deliver(j);
  } }}
public class Thread2 extends Thread
{
A obj;
Thread2(A obj)
{
 this.obj = obj;
}
public void run()
{
for(int k = 0; k <= 5; k++){
 obj.receive();
}}}
```

```
public class Communication
{
public static void main(String[]
args)
{
 A obj = new A();
// Creating an object of class A.

// Creating two thread objects and pass
reference variable obj as parameter to
Thread1 and Thread2.
Thread1 t1 = new Thread1(obj);
Thread2 t2 = new Thread2(obj);
// Run both threads.
  t1.start();
  t2.start();
 }
}
```

Output:

Data Delivered: 1
Data Received: 1
Data Delivered: 2
Data Received: 2
Data Delivered: 3
Data Received: 3
Data Delivered: 4
Data Received: 4
Data Delivered: 5
Data Received: 5

Inter-thread Communication

- In this example program, wait() and notify() methods are called inside deliver() and receive() method. Both methods enable Thread1 to notify Thread2 after producing data and wait until Thread2 complete using.
- In the same way, Thread2 after using data notifies Thread1 and waits until Thread1 produces and delivers the next data. Thus, the output comes in a synchronized form.
- If the flag is true, Thread2 takes data from Thread1 and use it. When Thread1 is busy producing data, now and then Thread2 will check flag is true or not.
- If the flag shows false, then Thread2 will wait for the object until it receives a notification from a notify() method.
- When the data production is over, Thread1 will send a notification immediately to Thread2 to receive data. In this way, Thread1 and Thread2 communicate with each other efficiently.

isAlive() method

- The isAlive() method of thread class tests if the thread is alive.
- A thread is considered alive when the start() method of thread class has been called and the thread is not yet dead. This method returns true if the thread is still running and not finished.

Syntax

- **public final boolean isAlive()**

Return

- This method will return true if the thread is alive otherwise returns false.

isAlive() method

```
public class JavaIsAliveExp extends Thread
{
    public void run()
    {
        try
        {
            Thread.sleep(300);
            System.out.println("is run() method isAlive "+Thread.currentThread().isAlive());
        }
        catch (InterruptedException ie) {
        }
    }
    public static void main(String[] args)
    {
        JavaIsAliveExp t1 = new JavaIsAliveExp();
        System.out.println("before starting thread isAlive: "+t1.isAlive());
        t1.start();
        System.out.println("after starting thread isAlive: "+t1.isAlive());
    }
}
```

before starting thread isAlive: false
after starting thread isAlive: true
is run() method isAlive true

join() method

- The join() method of thread class waits for a thread to die.
- It is used when you want one thread to wait for completion of another.
- This process is like a relay race where the second runner waits until the first runner comes and hand over the flag to him.
- A join() is a final method of Thread class and it can be used to join the start of a thread's execution to the end of another thread's execution so that a thread will not start running until another thread has ended.
- If the join() method is called on a thread instance, the currently running thread will block until the thread instance has finished executing.

Syntax

- **public final void join() throws InterruptedException**
- It does not **return** any value.

join() method

```
public class MyThread extends Thread
{
        public void run()
        {
                System.out.println("r1 ");
                try {
                        Thread.sleep(500);
                }catch(InterruptedException ie){ }
                System.out.println("r2 ");
        }
        public static void main(String[] args)
        {
                MyThread t1=new MyThread();
                MyThread t2=new MyThread();
                t1.start();

                try{
                        t1.join();      //Waiting for t1 to finish
                }catch(InterruptedException ie){}

                t2.start();
        }
}
```

r1
r2
r1
r2

Exception Handling in Threads

- When we call a sleep() method in a Java program, it must be enclosed in try block and followed by catch block.
- This is because sleep() method throws an exception named InterruptedException that should be caught. If we fail to catch this exception, the program will not compile.
- JVM (Java Runtime System) will throw an exception named IllegalThreadStateException whenever we attempt to call a method that a thread cannot handle in the given state.
- For example, a thread that is in a sleeping state cannot deal with the resume() method because a sleeping thread cannot accept instructions. The same thing is true for the suspend() method when it is used on a blocked thread.

Exception Handling in Threads

```
class MyThread extends Thread{
   public void run(){
      System.out.println("Throwing in " +"MyThread");
      throw new RuntimeException();
   }
}
public class Main {
   public static void main(String[] args) {
      MyThread t = new MyThread();
      t.start();
      try {
         Thread.sleep(1000);
      } catch (Exception x) {
         System.out.println("Caught it" + x);
      }
      System.out.println("Exiting main");
   }
}
```

```
Throwing in MyThread
Exception in thread "Thread-0"
java.lang.RuntimeException
at testapp.MyThread.run(Main.java:19)
Exiting main
```