Java Foundation

Presented by
Sunish Thazath

Java Threads

Before starting about Threads in java, just read out below definitions.

Application :

- Application is a program which is designed to perform a specific task.
- For example, MS Word, Google Chrome, a video or audio player etc.

Process :

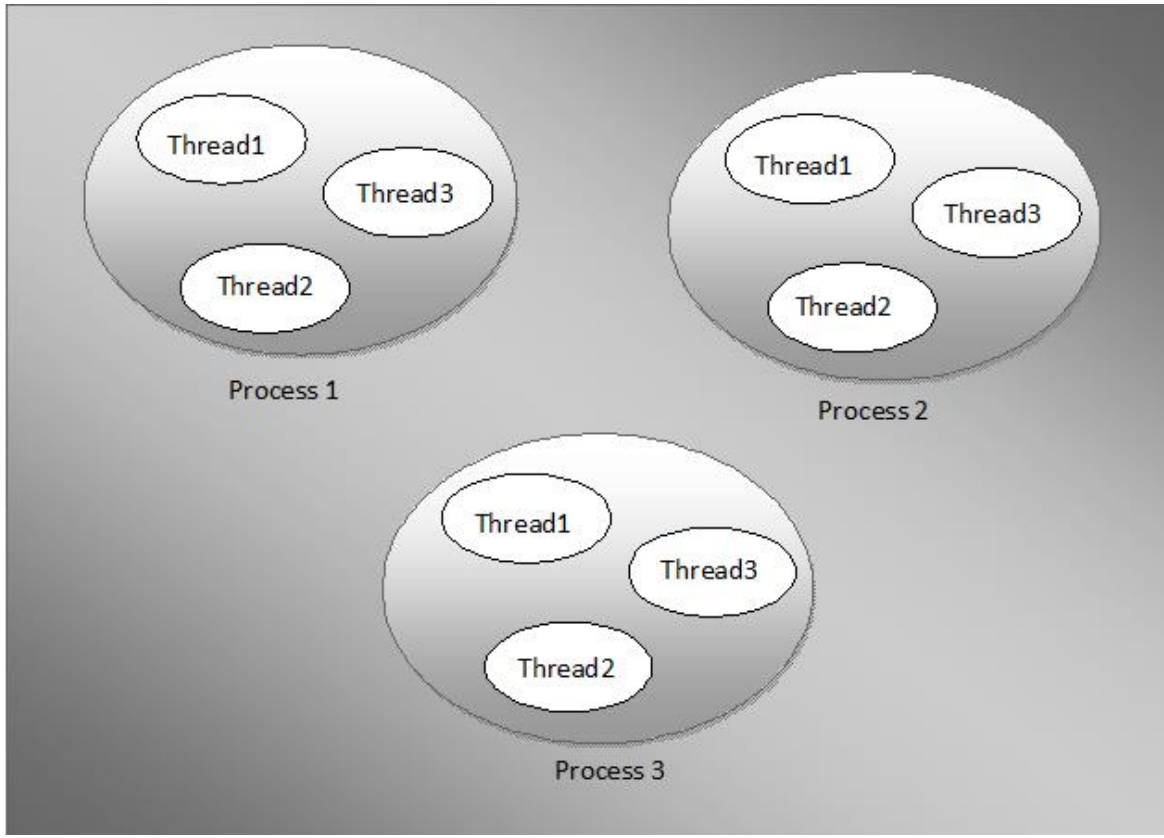
- Process is an executing instance of an application.
- For example, when you double click MS Word icon in your computer, you start a process that will run this MS word application.
- Processes are heavy weight operations that they require their own separate memory address in operating system.
- Because of the processes are stored in separate memory, communication between processes (Inter Process Communication) takes time. Context switching from one process to another process is also expensive.

Java Threads

Thread :

- Thread is a smallest executable unit of a process. Thread has it's own path of execution in a process.
- For example, when you start MS word, operating system creates a process and start the execution of a primary thread of that process.
- A process can have multiple threads.
- Threads of the same process share the memory address of that process. i.e. threads are stored inside the memory of a process.
- As the threads are stored in the same memory space, communication between threads (Inter Thread Communication) is fast.
- Context switching from one thread to another thread is also less expensive.

Processes and threads can be diagrammatically represented as below,



Java Threads

Multitasking:

- Multitasking is an operation in which multiple tasks are performed simultaneously.
- Multitasking is used to utilize CPU's idle time. Multitasking can be of two types. One is process-based and another one is thread-based.

1. Process-based multitasking Or Multiprocessing:

- In process-based multitasking or multiprocessing, Multiple processes are executed simultaneously.
- Because of this feature only, our computer runs two or more programs simultaneously.
- For example, You can play a video file and print a text file simultaneously in your computer.

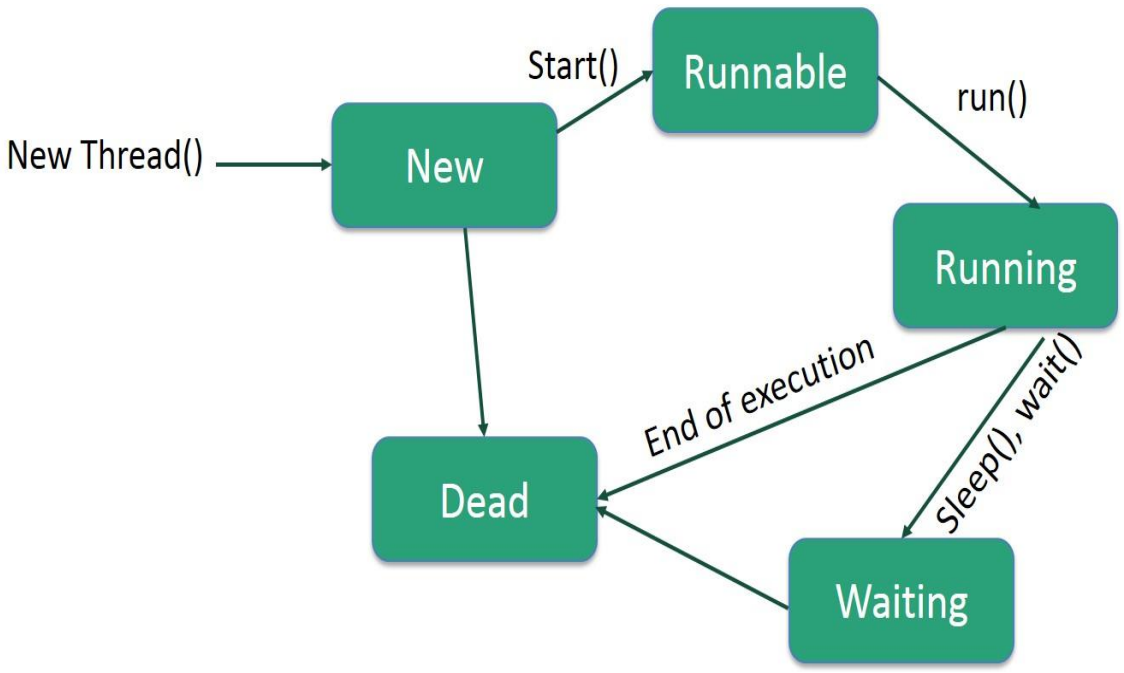
2. Thread-based Multitasking Or Multithreading:

- In thread-based multitasking or multithreading, multiple threads in a process are executed simultaneously.
- For example, MS word can print a document using background thread, at the same another thread can accept the user input so that user can create a new document.

Life Cycle of a Thread

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies.

The following diagram shows the complete life cycle of a thread.



New − A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.

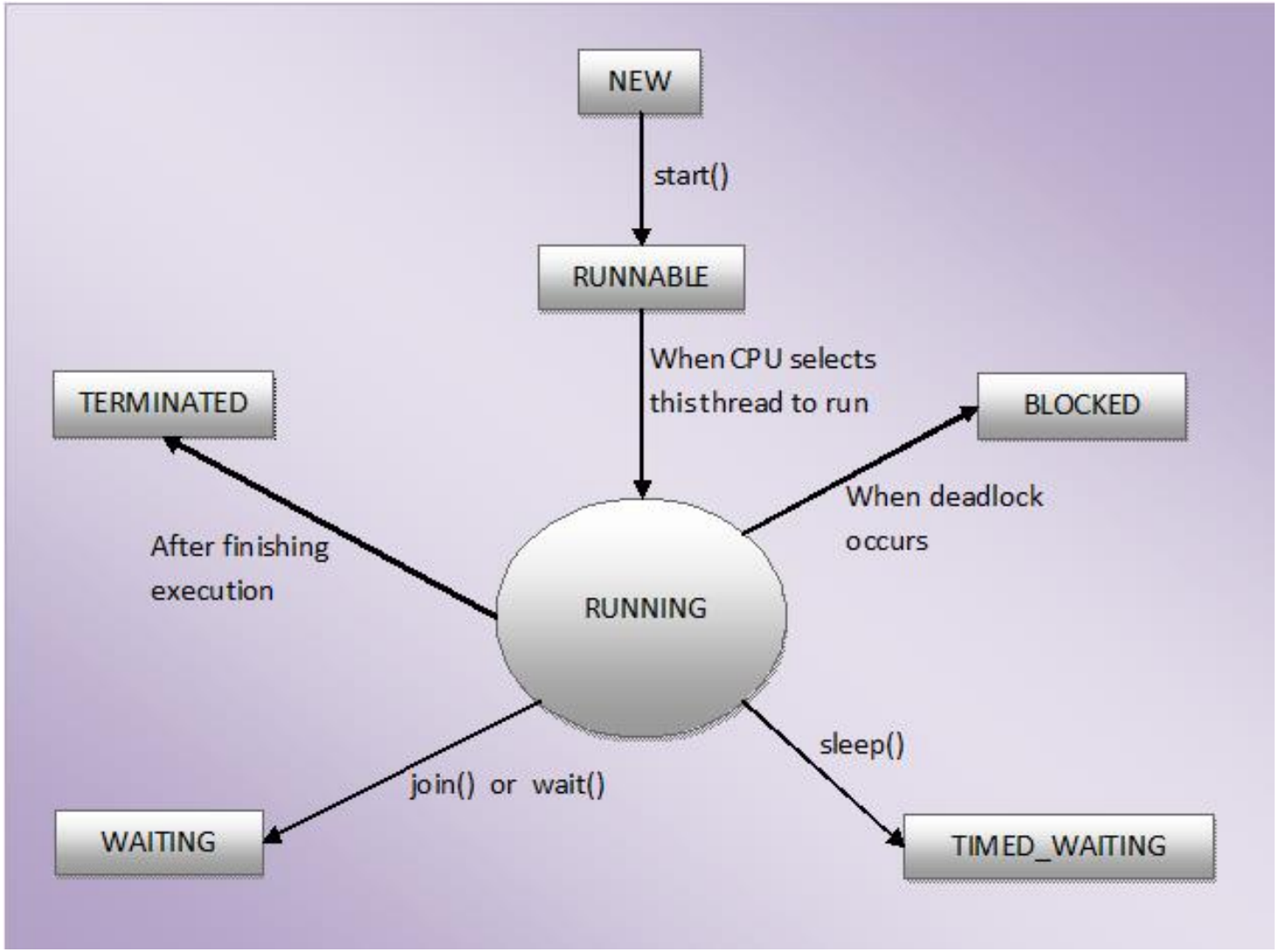
Runnable − After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.

Waiting − Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

Timed Waiting − A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.

Terminated (Dead) − A runnable thread enters the terminated state when it completes its task or otherwise terminates.

6 States of a Thread



RUNNING is not a state. This is when the thread is actually executing it's task. Just for understanding purpose have placed in the image.

Java Threads

Lets step ahead and see how Threads work in Java

```
public class ThreadsInJava
{
    //Main Thread
    public static void main(String[] args)
    {
        for (int i = 0; i <= 100; i++)
        {
            System.out.println("From Main Thread");
        }
    }
}
```

When you run this program, java command creates a primary thread or main thread which is responsible for executing main method. That’s why, execution of all java programs start with main() method.

This is an example of single thread execution. Java supports multi thread execution and let see in next slide.

Java Threads

Java supports multi thread execution. i.e. Java program can have more than one threads that can run simultaneously. This is called multithreaded programming.

- For example, in the below program, main thread creates two threads.
- The task of first thread is to print the numbers from 0 to 1000.
- The task of second thread is to print the numbers from 1001 to 2000.
- These two threads perform their task simultaneously not one after the other.

Thread-1

```
//Defining first thread with task
//The task of this thread is to print the numbers from 0 to 1000 times
class Thread1 extends Thread
{
   @Override
   public void run()
   {
      for(int i = 0; i <= 1000; i++)
      {
         System.out.println(i);
      }
   }
}
```

Thread-2

```
//Defining second thread with task
//The task of this thread is to print the numbers from 1001 to 2000
class Thread2 extends Thread
{
   @Override
   public void run()
   {
      for(int i = 1001; i<= 2000; i++)
      {
         System.out.println(i);
      }
   }
}
```

Main Thread

```
public class ThreadsInJava
{
   //Main Thread
   public static void main(String[] args)
   {
      //Creating first thread
      Thread1 t1 = new Thread1();
      t1.start();

      //Creating second thread
      Thread2 t2 = new Thread2();
      t2.start();
   }
}
```

Creating Threads in Java

There are two ways of creating threads in Java language:-

1. **By extending java.lang.Thread class**

- You can create your own thread by extending Thread class of java.lang package.
- You have to override run() method of Thread class and keep the task which you want your thread to perform in this run() method.
- Here is the syntax of creating a thread by extending Thread class.

```
class MyThread extends Thread
{
    @Override
    public void run()
    {
        //Keep the task to be performed here
    }
}
```

- After defining your thread, create an object of your thread and call start() method where ever you want this task to be performed. Like this,

```
MyThread myThread = new MyThread();
myThread.start();
```

Creating Threads in Java

How to create a thread by extending Thread class and how to start it?

```
class MyThread extends Thread
{
    @Override
    public void run()
    {
        //Task of this thread is to print multiplication of successive numbers up to 1000 numbers
        for(int i = 0; i < 1000; i++)
        {
            System.out.println(i+" * "+(i+1)+" = " + i * (i+1));
        }
    }
}

public class ThreadsInJava
{
    //Main Thread
    public static void main(String[] args)
    {
        //Creating and starting MyThread.
        MyThread myThread = new MyThread();
        myThread.start();
    }
}
```

Creating Threads in Java

2. By implementing java.lang.Runnable interface

- Another method of creating a thread is to implement Runnable interface of java.lang package.
- Runnable interface has only one abstract method i.e run().
- You have to implement this method and keep the task to be performed in this method.
- Here is the syntax for creating a thread by implementing Runnable interface.

```
class MyThread implements Runnable
{
    @Override
    public void run()
    {
        //Keep the task to be performed here
    }
}
```

- After defining the thread, create an object to java.lang.Thread class through a constructor which takes Runnable type as an argument and pass the object of your thread that implements Runnable interface as an argument to it and call the start() method.Like this,

```
MyThread myThread = new MyThread();    //Creating object of your thread that implements Runnable interface
Thread t = new Thread(myThread);       //passing your thread object to the constructor of Thread class
t.start();                             //Starting the thread
```

Creating Threads in Java

How to create a thread by implementing Runnable interface and how to start it?

```
class MyThread implements Runnable
{
    @Override
    public void run()
    {
        //Task of this thread is to print multiplication of successive numbers up to 1000 numbers
        for(int i = 0; i < 1000; i++)
        {
            System.out.println(i+" * "+(i+1)+" = " + i * (i+1));
        }
    }
}

public class ThreadsInJava
{
    //Main Thread
    public static void main(String[] args)
    {
        MyThread myThread = new MyThread();    //instantiating your thread class that implements Runnable interface
         Thread t = new Thread(myThread);       //Creating an object to Thread class by passing your thread as an argument
         t.start();                             //Starting the thread
    }
}
```

Types of Thread

❑ **User Thread**

- User threads are threads which are created by the application or user.
- They are high priority threads. JVM (Java Virtual Machine) will not exit until all user threads finish their execution.
- JVM wait for these threads to finish their task. These threads are foreground threads.

❑ **Daemon Thread**

- Daemon threads are threads which are mostly created by the JVM.
- These threads always run in background. These threads are used to perform some background tasks like garbage collection and house-keeping tasks.
- These threads are less priority threads.
- JVM will not wait for these threads to finish their execution.
- JVM will exit as soon as all user threads finish their execution.
- JVM doesn’t wait for daemon threads to finish their task.

Synchronization

- ✓ When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issues.

- ✓ For example, if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.

- ✓ So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time.

- ✓ Java programming language provides a very handy way of creating threads and synchronizing their task by using **synchronized** blocks.

- ✓ Following is the general form of the synchronized statement

```
synchronized(objectidentifier) {
   // Access shared variables and other shared resources
}
```

Here, the **objectidentifier** is a reference to an object.

Multithreading - Without Synchronization

```
class PrintDemo {
  public void printCount() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Counter   ---   "
+ i );
      }
    } catch (Exception e) {
      System.out.println("Thread
interrupted.");
    }
  }
}
```

```
Starting Thread - 1
Starting Thread - 2
Counter   ---   5
Counter   ---   4
Counter   ---   3
Counter   ---   5
Counter   ---   2
Counter   ---   1
Counter   ---   4
Thread Thread - 1  exiting.
Counter   ---   3
Counter   ---   2
Counter   ---   1
Thread Thread - 2  exiting.
```

```
class ThreadDemo extends Thread {
  private Thread t;
  private String threadName;
  PrintDemo  PD;

  ThreadDemo( String name,  PrintDemo pd) {
    threadName = name;
    PD = pd;
  }

  public void run() {
    PD.printCount();
    System.out.println("Thread " +  threadName + "
exiting.");
  }

  public void start () {
    System.out.println("Starting " +  threadName );
    if (t == null) {
      t = new Thread (this, threadName);
      t.start ();
    }
  }
}
```

```
public class TestThread {
  public static void main(String args[]) {

    PrintDemo PD = new PrintDemo();

    ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD
    ThreadDemo T2 = new ThreadDemo( "Thread - 2 ", PD

    T1.start();
    T2.start();

    // wait for threads to end
      try {
      T1.join();
      T2.join();
    } catch ( Exception e) {
      System.out.println("Interrupted");
    }
  }
}
```

Multithreading - With Synchronization

```
class PrintDemo {
  public void printCount() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Counter   ---   "
+ i );
      }
    } catch (Exception e) {
      System.out.println("Thread
interrupted.");
    }
  }
}
```

```
Starting Thread - 1
Starting Thread - 2
Counter   ---   5
Counter   ---   4
Counter   ---   3
Counter   ---   5
Counter   ---   2
Counter   ---   1
Counter   ---   4
Thread Thread - 1  exiting.
Counter   ---   3
Counter   ---   2
Counter   ---   1
Thread Thread - 2  exiting.
```

```
class ThreadDemo extends Thread {
  private Thread t;
  private String threadName;
  PrintDemo  PD;

  ThreadDemo( String name,  PrintDemo pd) {
    threadName = name;
    PD = pd;
  }

  public void run() { //make it synchronized
    synchronized(PD) {
      PD.printCount();
    }
    System.out.println("Thread " +  threadName + "
exiting.");
  }

  public void start () {
    System.out.println("Starting " +  threadName );
    if (t == null) {
      t = new Thread (this, threadName);
      t.start ();
    }
  }
}
```

```
public class TestThread {
  public static void main(String args[]) {

    PrintDemo PD = new PrintDemo();

    ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD
    ThreadDemo T2 = new ThreadDemo( "Thread - 2 ", PD

    T1.start();
    T2.start();

    // wait for threads to end
      try {
      T1.join();
      T2.join();
    } catch ( Exception e) {
      System.out.println("Interrupted");
    }
  }
}
```

Deadlock

- Deadlock in java is a condition which occurs when two or more threads get blocked waiting for each other for an infinite period of time to release the resources(Locks) they hold.

- Deadlock is the common problem in multi threaded programming which can completely stops the execution of an application. So, extra care need to be taken while writing the multi threaded programs so that deadlock never occurs.

- **Refer** sample program, thread t1 and t2 are concurrent threads i.e they are executing their task simultaneously. There are two Shared class objects, s1 and s2, which are shared by both the threads. Shared class has two synchronized methods, methodOne() and methodTwo(). That means, only one thread can execute these methods at a given time.

- First, thread t1 enters the methodOne() of s1 object by acquiring the object lock of s1. At the same time, thread t2also enters the methodTwo() of s2 object by acquiring the object lock of s2. methodOne() of s1 object, currently executing by thread t1, calls methodTwo() of s2 object from it's body. So, thread t1 tries to acquire the object lock of s2 object. But object lock of s2 object is already acquired by thread t2. So, thread t1 waits for thread t2 to release the object lock of s2 object.

- At the same time, thread t2 is also executing methodTwo() of s2 object. methodTwo() of s2 object also makes a call to methodOne() of s1 object. So, thread t2 tries to acquire the object lock of s1 object. But, it is already acquired by thread t1. So, thread t2 also waits for thread t1 to release the object lock of s1 object.

- Thus, both the threads wait for each other to release the object locks they own. They wait for infinite period of time to get the object locks owned by opposite threads. This condition of threads waiting forever is called Deadlock.


Thank You!