Java Programming

Multithreading - I

Module 7

Agenda

- Introduction to Multithreading
- Java's Multithreading Model
- Creating Multiple Threads
- Thread Control Mechanism

Objectives

- At the end of this module, you will be able to:
  - Compare Multitasking and Multithreading
  - List the benefits of Multithreading
  - Identify Java's Multithreading model
  - Implement java programs to create and use threads
  - Identify the techniques of controlling thread execution

Introduction to Multithreading

Need for Multithreading

Have you faced the following situations:

Your browser cannot skip to the next web page because it is downloading a file?

You cannot enter text into your current document until your word processor completes the task of saving the document to disk

What is Multitasking?

- Multitasking is synonymous with process-based multitasking, whereas multithreading is synonymous with thread-based multitasking
- All modern operating systems support multitasking
- A process is an executing instance of a program
- Process-based multitasking is the feature by which the operating system runs two or more programs concurrently

What is Multithreading?

- In multithreading, the thread is the smallest unit of code that can be dispatched by the thread scheduler
- A single program can perform two tasks using two threads
- Only one thread will be executing at any given point of time given a single-processor architecture

Multitasking Vs. Multithreading

Compared to multithreading, multitasking is characterized by the following:

- Each process requires its own separate address space
- Context switching from one process to another is a CPU-intensive task needing more time
- Inter-process communication between processes is again expensive as the communication mechanism has to span separate address spaces

These are the reasons why processes are referred to as heavyweight tasks

Multitasking Vs. Multithreading (Contd.).

- Threads cost less in terms of processor overhead because of the following reasons:
  - Multiple threads in a program share the same address space and they are part of the same process
  - Switching from one thread to another is less CPU-intensive
  - Inter-thread communication, on the other hand, is less expensive as threads in a program communicate within the same address space

- Threads are therefore called lightweight processes

Uses of Multithreading

- A multithreaded application performs two or more activities concurrently
- It is accomplished by having each activity performed by a separate thread
- Threads are the lightest tasks within a program, and they share memory space and resources with each other

Single-Threaded Systems

- 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 which instruction to execute next
  - Until this instruction returns, nothing else can happen in the system
  - This results in wastage of precious CPU cycles

Java's Multithreading Model

Java's Multithreading Model

- Java has completely done away with the event loop/ polling mechanism

- *In Java*
  - *All the libraries and classes are designed with multithreading in mind*
  - *This enables the entire system to be asynchronous*

- In Java the **java.lang.Thread** class is used to create thread-based code, imported into all Java applications by default

The Thread class

- Java's multithreading feature is built into the **Thread** class

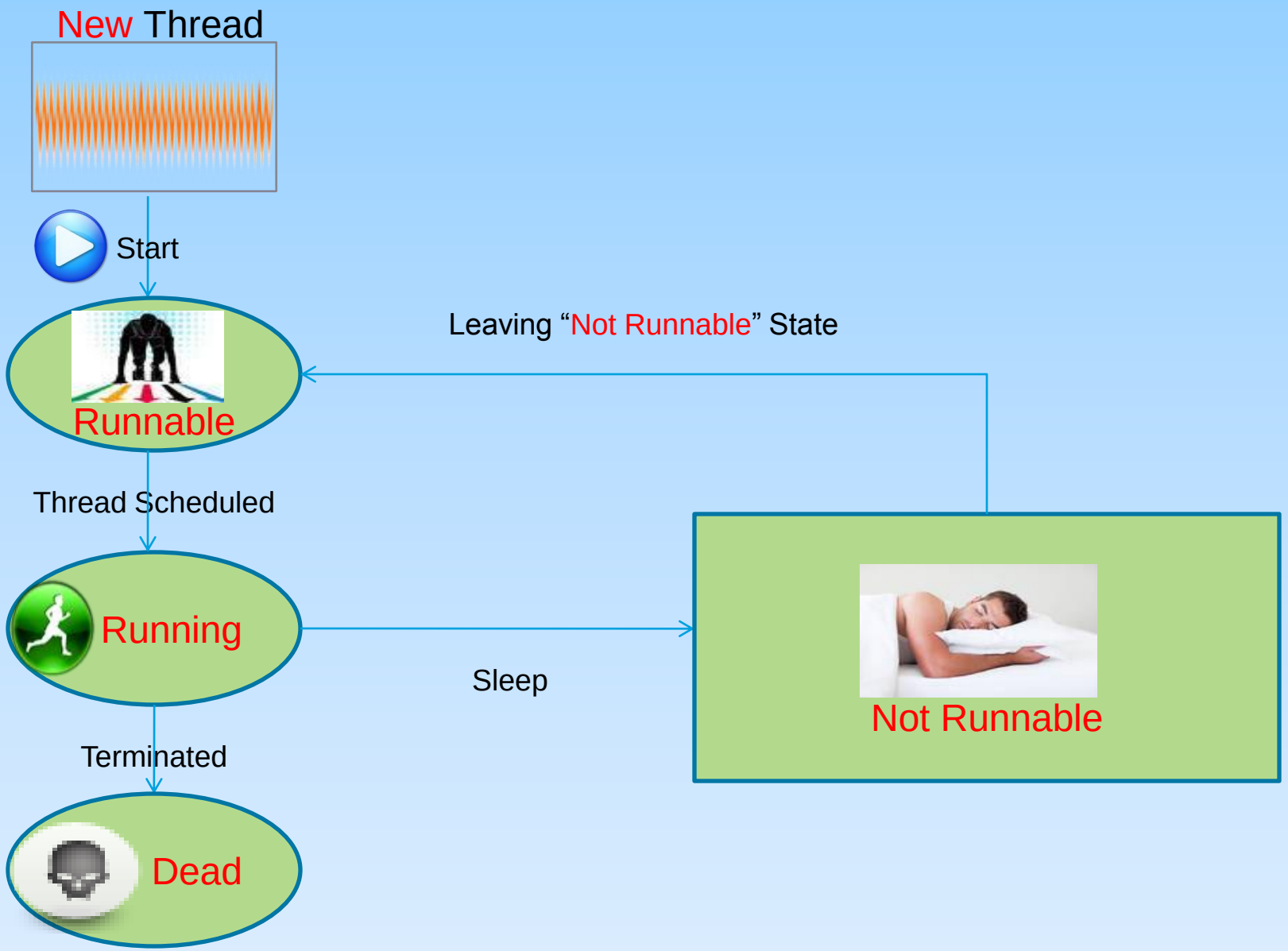
- The **Thread** class has two primary thread control methods:

  public void start() - The **start()** method starts a thread execution

  public void run() - The **run()** method actually performs the work of the thread and is the entry point for the thread

- The thread *dies* when the **run( )** method terminates
- You never call **run( )** explicitly
- The **start( )** method called on a thread automatically initiates a call to the thread's **run( )** method

Different States of a Thread



The main Thread

- When a Java program starts executing:
  - the main thread begins running
  - the main thread is immediately created when **main()** commences execution

- Information about the main or any thread can be accessed by obtaining a reference to the thread using a public, static method in the **Thread** class called **currentThread( )**

Obtaining Thread-Specific Information

```
public class ThreadInfo {
  public static void main(String args[]) {
    Thread t = Thread.currentThread( );
    System.out.println("Current Thread :" + t);
    t.setName("Demo Thread");
    System.out.println("New name of the thread :"  +
  t);
    try {
       Thread.sleep(1000);
    }
    catch (InterruptedException e) {
       System.out.println("Main Thread Interrupted");
    }
  }
}
```

Obtaining Thread-Specific Information (Contd.).

```
public static void main(String args[]) {
  Thread t = Thread.currentThread( );
  System.out.println("Current Thread :" + t);
  t.setName("Demo Thread");
  System.out.println("New name of the thread :"
+  t);
  try {
     Thread.sleep(1000);
  }
  catch (InterruptedException e) {
     System.out.println("Main Thread
Interrupted");
  }
} }
```

Creating Threads

- A thread can be created by instantiating an object of type **Thread**.
- This can be achieved in any of the following two ways :
  - implementing the **Runnable** interface
  - extending the **Thread** class
- **Creating Threads – Implementing Runnable**
  - Create a class, which must implement the **interface Runnable**
  - A thread can be constructed on any object that implements the **Runnable** interface.
  - To implement **Runnable**, a class need implement only a single method called **run( )**

Creating Threads: Implementing Runnable

After defining the class that implements Runnable, we have to create an object of type Thread from within the object of that class. This thread will end when run( ) returns or terminates.

This is mandatory because a thread object confers multithreaded functionality to the object from which it is created.

Therefore, at the moment of thread creation, the thread object must know the reference of the object to which it has to confer multithreaded functionality. This point is borne out by one of the constructors of the Thread class.

Creating Threads: Implementing Runnable (Contd.).

The Thread class defines several constructors one of which is:

Thread(Runnable threadOb, String threadName)

In this constructor, “threadOb” is an instance of a class implementing the Runnable interface and it ensures that the thread is associated with the run( ) method of the object implementing Runnable

When the run() method is called, the thread is believed to be in execution.

String threadName – we associate a name with this thread.

Creating Threads: Implementing Runnable (Contd.).

```
public class DemoThread1 implements Runnable {
  Thread t;
  DemoThread1() {
    t = new Thread(this, "Demo1 Thread");
    System.out.println("Child1 Thread: " + t);
    t.start();
  }
  public void run() {
    try {
          for(int i=5; i>0; i--) {
        System.out.println("Child1 Thread: " + i);
        Thread.sleep(500);
          }
    }
    catch (InterruptedException e) {
         System.out.println("Child1 Thread is Interrupted");
    }
    System.out.println("Child1 Thread RIP");
  }
}
```

Creating Threads: Implementing Runnable (Contd.).

```
class ThreadImpl {
  public static void main(String args[]) {
    new DemoThread1();

    try {
          for(int i=5; i>0; i--) {
       System.out.println("The Main Thread: " + i);
       Thread.sleep(1000);
         }
    }
    catch (InterruptedException e) {
         System.out.println("Main Thread is Interrupted");
    }
    System.out.println("Main Thread RIP");
  }
}
```

Extending Thread

- We can also create Threads by extending the Thread class:
  - Instantiate the class that extends Thread
  - This class must override run() method
  - The code that should run as a thread will be part of this run() method
  - We must call the start() method on this thread
  - start( ) in turn calls the thread's run( ) method

- The earlier program is rewritten by extending the Thread class.

Extending Thread Example

```
public class DemoThread2 extends Thread {
  DemoThread2() {
    super("Demo Thread2");
    System.out.println("Child2 Thread:" + this);
    start();
  }
public void run() {
    try {
          for(int i=5; i>0; i--) {
       System.out.println("Child2 Thread: " + i);
       Thread.sleep(500);
         }
    }
    catch (InterruptedException e)  {
       System.out.println("Child2 Thread is Interrupted");
    }
    System.out.println ("Child2 Thread RIP" );
  }
}
```

Extending Thread Example (Contd.).

```
class ThreadImpl {
  public static void main(String args[]) {
    new DemoThread2();
    try {
          for(int i=5; i>0; i--) {
       System.out.println("Child2 Thread: " + i);
       Thread.sleep(1000);
          }
    }
    catch (InterruptedException e) {
       System.out.println("The Main Thread
  isInterrupted");
    }
    System.out.println("Main Thread RIP");
  }
}
```

Implementing Runnable or Extending Thread ?

- A modeling heuristic pertaining to hierarchies says that classes should be extended only when they are enhanced or modified in some way

- So, if the sole aim is to define an entry point for the thread by overriding the **run( )** method and not override any of the **Thread** class' other methods, it is recommended to implement the **Runnable** interface

Creating Multiple threads

Creating Multiple Threads

- You can launch as many threads as your program needs
- The following example is a program spawning multiple threads:

```
public class DemoThread3 implements Runnable {
  String tname;
  Thread t;

  DemoThread3(String thread_name) {
    tname = thread_name;
    t = new Thread(this, tname);
    System.out.println("Just created Thread: " + t);
    t.start();
  }
```

Creating Multiple Threads (Contd.).

```
  public void run() {
    try {
          for(int z=5; z>0; z--) {
       System.out.println("kid Thread: " + z);
       Thread.sleep(1000);
           }
    }
    catch (InterruptedException e) {
          System.out.println(name + "is
  Interrupted");
    }
    System.out.println(name +"is dying");
  }
}
```

Creating Multiple Threads (Contd.).

```
class MultiThreadImplement {
  public static void main(String args[]) {
    new DemoThread3("One");
    new DemoThread3("Two");
    new DemoThread3("Three");
    try {
          Thread.sleep(10000);
    }
    catch (InterruptedException e) {
          System.out.println("The Main Thread is
  Interrupted");
    }
    System.out.println("Main Thread is about to
  dye");
  }
}
```

Time to think…!!!

In the program that was just demonstrated, if we replace t.start() in the constructor of DemoThread with t.run(), what will be the consequence ?

Will all the threads run? Beginning with the commencement of main() and till the end of main(), how many threads in total get executed?

1. No difference. The program runs successfully with four threads(3 kid threads and main), running simultaneously as the program that was just demonstrated
2. The program runs successfully, but all the four threads run one after the other
3. The program fails to compile
4. The program executes without any error but only one thread gets executed, i.e. main() thread
5. The program throws “Thread not started exception”

Thread Control Mechanism

Control Thread Execution

- Two ways exist by which you can determine whether a thread has finished:
- The **isAlive( )** method will return true if the thread upon which it is called is still running; else it will return false
- The **join( )** method waits until the thread on which it is called terminates.

Control Thread Execution (Contd.).

```
public class DemoThread implements Runnable {
  String name; Thread t;
  DemoThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("New Thread: " + t);
    t.start();
  }
public void run() {
    try {
          for(int i=5; i>0; i--) {
       System.out.println("Child Thread: " + i);
       Thread.sleep(1000); }
         }
    catch (InterruptedException e) {
            System.out.println(name + "Interrupted");
    }
    System.out.println(name +"Exiting");
  }
}
```

Control Thread Execution (Contd.).

```
  public void run() {
    try {
          for(int i=5; i>0; i--) {
       System.out.println("Child Thread: " + i);
       Thread.sleep(1000); }
          }
    catch (InterruptedException e) {
            System.out.println(name +
  "Interrupted");
    }
    System.out.println(name +"Exiting");
  }
}
```

Control Thread Execution (Contd.).

```
class MultiThreadImpl {
  public static void main(String args[]) {
    DemoThread t1 = new DemoThread("One");
    DemoThread t2 = new DemoThread("Two");
    DemoThread t3 = new DemoThread("Three");
    System.out.println("Thread One is alive: " +
  t1.t.isAlive());
    System.out.println("Thread Two is alive: " +
  t2.t.isAlive());
    System.out.println("Thread Three is alive: "
  + t1.t.isAlive());
```

Control Thread Execution (Contd.).

```
    try {
            System.out.println("Waiting for child
  threads to finish");
            t1.t.join();
            t2.t.join();
            t3.t.join();
    }
    catch (InterruptedException e) {
            System.out.println("Main thread
  interrupted");
    }
    System.out.println("Thread One is alive: " +
  t1.t.isAlive());
    System.out.println("Thread One is alive: " +
  t1.t.isAlive());
    System.out.println("Thread One is alive: " +
  t1.t.isAlive());
    System.out.println("Main thread exiting");
  }
}
```

Review

Fill in the blanks :

1. isAlive() method returns _____ , while join() method returns void.
2. Thread.sleep(10000) means the thread will be sleeping for _____ seconds.
3. The first method of creating a class which will be having thread capability, is by extending the class Thread while the second method is by implementing _____ interface.
4. The main thread, by default, has a priority _____.
5. When the run() method terminates, the thread is supposed to be in _____ state.

Summary

- In this module, you were able to:
  - Compare Multitasking and Multithreading
  - List the benefits of Multithreading
  - Identify Java's Multithreading model
  - Implement java programs to create and use threads
  - Identify the techniques of controlling thread execution

References

1. Schildt, H. Java: *The Complete Reference. J2SETM*. Ed 5. New Delhi: McGraw Hill-Osborne, 2005.

2. Tutorial point (2012). *Java: Multithreading.* Retrieved on March 29, 2012, from, http://www.tutorialspoint.com/java/java_multithreading.htm

3. Oracle (2012). The Java Tutorials: Concurrency**.** Retrieved on March 29, 2012, from, http://docs.oracle.com/javase/tutorial/essential/concurrency/

Thank You