Course: Object Based Modeling
Code: CS-33105
Branch: MCA-3

Lecture 12: Multi Threading

Dr. J Sathish Kumar (JSK)
(Faculty & Coordinator)

Department of Computer Science and Engineering

Motilal Nehru National Institute of Technology Allahabad,
Prayagraj-211004

Introduction

- A multithreaded program contains two or more parts that can run concurrently.
- Each part of such a program is called a *thread*, and each thread defines a separate path of execution.
- Thus, multithreading is a specialized form of multitasking.
- Two distinct types of multitasking: process-based and thread-based.
- *Process-based* multitasking is the feature that allows your computer to run two or more programs concurrently.
- In a *thread-based* multitasking environment, the thread is the smallest unit of dispatchable code.
- This means that a single program can perform two or more tasks simultaneously.
- For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

Process-based and Thread-based

- Multitasking threads require less overhead than multitasking processes.
- Processes are heavyweight tasks that require their own separate address spaces.
- Interprocess communication is expensive and limited.
- Context switching from one process to another is also costly.

- Threads,on the other hand, are lighter weight.
- They share the same address space and cooperatively share the same heavyweight process.
- Interthread communication is inexpensive, and context switching from one thread to the next is lower in cost.

Multi Threading

- Multithreading enables you to write efficient programs that make maximum use of the processing power available in the system.
- One important way multithreading achieves this is by keeping idle time to a minimum.
- This is especially important for the interactive, networked environment in which Java operates because idle time is common.
- For example,
  - The transmission rate of data over a network is much slower than the rate at which the computer can process it.
  - Even local file system resources are read and written at a much slower pace than they can be processed by the CPU.
  - And, of course, user input is much slower than the computer.
  - In a single-threaded environment, your program has to wait for each of these tasks to finish before it can proceed to the next one—even though most of the time the program is idle, waiting for input.
  - Multithreading helps you reduce this idle time because another thread can run when one is waiting.

The Thread Class and the Runnable Interface

- Java's multithreading system is built upon the Thread class, its methods, and its companion interface, Runnable.
- Thread encapsulates a thread of execution.
- To create a new thread, your program will either extend **Thread** or implement the **Runnable** interface.
- The Thread class defines several methods that help manage threads.

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

The Main Thread

- When a Java program starts up, one thread begins running immediately called the *main thread* of your program.
- The main thread is important for two reasons:
  - It is the thread from which other “child” threads will be spawned.
  - Often, it must be the last thread to finish execution because it performs various shutdown actions.
- Controlled through a Thread object
  - Obtain a reference to it by calling the method **currentThread( )**, which is a **public static** member of **Thread**.

static Thread currentThread( )

Example #1

```
// Controlling the main Thread.
class CurrentThreadDemo {
  public static void main(String args[]) {
    Thread t = Thread.currentThread();

    System.out.println("Current thread: " + t);

    // change the name of the thread
    t.setName("My Thread");
    System.out.println("After name change: " + t);

    try {
      for(int n = 5; n > 0; n--) {
        System.out.println(n);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted");
    }
  }
}
```

```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

Creating a Thread

- Java defines two ways in which this can be accomplished:
  - You can implement the **Runnable** interface.
  - You can extend the **Thread** class, itself.
- The easiest way to create a thread is to create a class that implements the **Runnable** interface.
- **Runnable** abstracts a unit of executable code.
- You can construct a thread on any object that implements **Runnable**.
- To implement **Runnable**, a class need only implement a single method called **run( )**,

public void run( )

run( ) can call other methods, use other classes, and declare variables, just like the main thread can.

Implementing Runnable

Thread(Runnable threadOb, String threadName)

- In this constructor, *threadOb* is an instance of a class that implements the **Runnable** interface. This defines where execution of the thread will begin.
- The name of the new thread is specified by *threadName*.
- After the new thread is created, it will not start running until you call its **start( )** method, which is declared within **Thread**. In essence, **start( )** executes a call to **run( )**.

void start( )

```
// Create a second thread.
class NewThread implements Runnable {
  Thread t;

  NewThread() {
    // Create a new, second thread
    t = new Thread(this, "Demo Thread");
    System.out.println("Child thread: " + t);
    t.start(); // Start the thread
  }

  // This is the entry point for the second thread.
  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.");
  }

}
```

Example #2

Passing this as the first argument indicates that you want the new thread to call the run( ) method on this object.

Example #2

```
class ThreadDemo {
  public static void main(String args[ ] ) {
    new NewThread(); // create a new 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.");
  }
}
```

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

Extending Thread

- The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class.
- The extending class must override the **run( )** method, which is the entry point for the new thread.
- It must also call **start( )** to begin execution of the new thread.

```
// Create a second thread by extending Thread
class NewThread extends Thread {

  NewThread() {
    // Create a new, second thread
    super("Demo Thread");
    System.out.println("Child thread: " + this);
    start(); // Start the thread
  }

  // This is the entry point for the second thread.
  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.");
  }
}
```

Example #3

```
class ExtendThread {
  public static void main(String args[]) {
    new NewThread(); // create a new 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.");
  }
}
```

Example #3

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```