Multithreaded Programming Part 2

Prof. Siddharth Shah
Department of Computer Engineering
Dharmsinh Desai University

Outline

- Creating a Thread
- Creating Multiple Threads
- Using ***isAlive()*** and ***join()***
- Thread Priorities
- Synchronization
- Monitor
- Synchronized Method
- Synchronized Statement

Creating a Thread

- In the most general sense, you create a thread by instantiating an object of type Thread.
- Java defines two ways in which this can be accomplished:
- Implement the Runnable interface.
- Extend the Thread class, itself.

Implementing Runnable - 1

- The easiest way to create a thread is to create a class that implements the Runnable interface.
- A thread can be constructed on any object that implements Runnable.
- To implement Runnable, a class needs to implement a single method called run( ), which is declared like this:
- **public void run( )**

Implementing Runnable - 2

- Inside run( ), define the code that constitutes the new thread.
- It is important to understand that run( ) can call other methods, use other classes, and declare variables, just like the main thread can.
- The only difference is that run( ) establishes the entry point for another, concurrent thread of execution within your program.
- This thread will end when run( ) returns.

Implementing Runnable - 3

- After creating a class that implements Runnable, instantiate an object of type **Thread** from within that class.
- Though Thread class defines several constructors. The one that we will use is shown here:
- **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.

Implementing Runnable - 4

- 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( ).
- The start( ) method is shown here:
- **void start( )**

Implementing Runnable : Summary

- Create a class which implements Runnable interface.
- Implement run() method by writing a code that you want to execute in your thread.
- Instantiate an object of Thread inside your class by passing the reference to your class.
- Call start() method on the newly created Thread object which in turn will call run() method that you have implemented.

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.

Choosing an Approach

- Though there are 2 different ways to create a thread in Java, most programmers feel that creating a thread by implementing Runnable is better. The reasons for that are as follows:
- Whether extending Thread class or implementing Runnable, in both cases we implements only run() method. Class should be extended if its behavior needs to be modified.
- By implementing Runnable, your thread class does not need to inherit Thread, making it free to inherit a different class.

Using *isAlive()* and *join()* - 1

- As mentioned, often main thread is to finish last.
- In the preceding examples, this is accomplished by calling sleep( ) within main( ), with a long enough delay to ensure that all child threads terminate prior to the main thread.
- However, this is hardly a satisfactory solution, and it also raises a larger question: How can one thread know when another thread has ended?
- Fortunately, Thread provides a means by which you can answer this question.

Using ***isAlive()*** and ***join()*** - 2

- Two ways exist to determine whether a thread has finished.
- First, you can call isAlive( ) on the thread. This method is defined by Thread, and its general form is shown here:
- **final boolean isAlive( )**
- The isAlive( ) method returns true if the thread upon which it is called is still running. It returns false otherwise.

Using *isAlive()* and *join()* - 3

- While isAlive( ) is occasionally useful, the method that is more commonly used to wait for a thread to finish is called join( ), shown here:
- **final void join( ) throws InterruptedException**
- This method waits until the thread on which it is called terminates.
- Its name comes from the concept of the calling thread waiting until the specified thread joins it.
- Additional forms of **join( )** allow you to specify a maximum amount of time that program wants to wait for the specified thread to terminate.

Thread Priorities - 1

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.
- In theory, over a given period of time, higher-priority threads get more CPU time than lower-priority threads.
- In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority.
- A higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower-priority thread.

Thread Priorities - 2

- In theory, threads of equal priority should get equal access to the CPU.
- But you need to be careful. Remember, Java is designed to work in a wide range of environments. Some of those environments implement multitasking fundamentally differently than others.
- For safety, threads that share the same priority should yield control once in a while.
- This ensures that all threads have a chance to run under a nonpreemptive operating system.

Thread Priorities - 3

- In practice, even in nonpreemptive environments, most threads still get a chance to run, because most threads inevitably encounter some blocking situation, such as waiting for I/O.
- When this happens, the blocked thread is suspended and other threads can run.
- But, if you want smooth multithreaded execution, you are better off not relying on this.
- Also, some types of tasks are CPU-intensive. Such threads dominate the CPU. For these types of threads, you want to yield control occasionally so that other threads can run.

Thread Priorities - 4

- To set a thread's priority, use the setPriority( ) method, which is a member of Thread. The general form is:
- **final void setPriority(int level)**
- Here, level specifies the new priority setting for the calling thread. The value of level must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**. Currently, these values are **1** and **10**, respectively. To return a thread to default priority, specify **NORM_PRIORITY**, which is currently **5**. These priorities are defined as static final variables within Thread.
- You can obtain the current priority setting by calling the getPriority( ) method of Thread, shown here:
- **final int getPriority( )**

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 is achieved is called **synchronization**.
- Java provides unique, language-level support for it.
- Key to synchronization is the concept of the monitor.

Monitor

- A monitor is an object that is used as a mutually exclusive lock. Only one thread can own a monitor at a given time.
- When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.
- These other threads are said to be waiting for the monitor.
- A thread that owns a monitor can reenter the same monitor if it so desires.
- You can synchronize your code in either of two ways:
- Synchronized Method
- Synchronized Statement

Synchronized Methods

- To achieve synchronization, thread has to enter into object's monitor.
- To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword.
- While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait.
- To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method

Synchronized Statements - 1

- While creating synchronized methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases.
- To understand why, consider the following.
- Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use synchronized methods.
- Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can’t add synchronized to the appropriate methods within the class.
- How can access to an object of this class be synchronized?

Synchronized Statements - 2

- Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a synchronized block.
- This is the general form of the synchronized statement:

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

- Here, objRef is a reference to the object being synchronized.
- A synchronized block ensures that a call to a synchronized method that is a member of objRef's class occurs only after the current thread has successfully entered objRef's monitor.