A QUICK LOOK AT JAVA THREADS

First Edition

Krishna Mohan Koyya

GLARIMY

Glarimy Technology Services
106, Vars Casa Rosa, Pai Layout,
BANGALORE – 560 016
India

www.glarimy.com

A QUICK LOOK AT

JAVA THREADS

First Edition

Contents

Preface

Welcome to *A Quick Look At Java Threads*. This is book is aimed to help the serious programmers in dealing with threads and concurrency in Java, *quickly*. When I say serious programmers, I refer to the programmers who not only just read the technology but apply the theory and explore the technology it in their own way.

The design and organization of the book helps in understanding the concepts *quickly*, looking at the illustrations immediately to see concepts in working and then exploring further beyond the concepts. Each of the chapters is divided into three parts

- Understanding the concepts
- Applying the concepts
- Going Beyond the concepts

True to their names, the part of Understanding the concepts takes you through the concepts, quickly. It presents the problem that a feature of Java addresses, the solution and other related facts. The part of Applying the concepts deals with code. It presents a full working code supplemented with the explanation and output. This cements your understanding of the concepts. Once you are armed with the concepts, then the third part Going Beyond the Concepts takes you through few interesting facets and questions. Some of them are answered and some of them left for you to explore by writing code, going through the Java documentation or just by thinking further.

All you need to use this book is JDK 1.6 and any IDE of your choice. You are expected to be good at basic Java programming. It would be easier for you to grasp the topics if you have good understanding of operating systems.

Hope this book helps in enhancing your interest and confidence in writing concurrent applications in Java, again, *quickly*.

Krishna Mohan Koyya
Bangalore
16th March 2011

About the Author

Krishna Mohan Koyya has been associated with the software industry for the last 14 years as a developer, teacher, trainer and author. His primary interests in technology include Java, object oriented architectures and Web 2.0.

As a software developer he worked with OpenView technology at Hewlett-Packard, Lucent's GSM OSS at Wipro Technologies and CiscoWorks NMS platform at Cisco Systems nearly for 10 years.

He taught Software Engineering at Sasi Institute of Engineering and Technology, Tadepalligudem for an year before taking up training and consulting as full time career in 2008.

As a trainer and consultant he handles most of the Java technologies ranging from JSE, JEE, Spring Framework, Struts Framework, JAX-WS, OSGi and etc and other technologies like Dojo framework, Mash-ups, UML and etc., Few of the clients to whom he delivered the services include Hewlett-Packard, Lucent-Alcatel, Intel, Sapient, IBM, Samsung, Oracle, Wipro, L&T and MindTree.

Currently he heads Glarimy Technology Services, Bangalore, India.

Academically he holds a bachelors in Electronics and Communications Engineering and masters in Computer Science and Technology, both from Andhra University, Visakhapatnam, India.

He can be reached at krishna@glarimy.com or at www.glarimy.com/krishna

Chapter 1

Understanding Threads

- What is a process?
- What is a thread?
- How a multi-threaded system is useful?
- Life cycle of a thread in Java
- Characteristics of a thread
- Overview of the `main` thread

Understanding the Concepts

Processes and Multi-Processing System

A process is a program under execution. Most of the modern day operating systems are multi-processing systems. They manage multiple processes simultaneously. Typically, any 0perating system allocates certain resources like memory exclusively for each of the processes and gives CPU time to each of the processes in turns. If a process can not continue for the want of, let's say, an external input, OS gives aways the time slot for another process in waiting instead of wasting precious CPU time.

Threads and Multi-Threaded System

A thread is an independent context of execution with in a process. A process can have more than one thread. All threads of a process share the resources allocated to the process. When one thread of a process can not continue further,

other thread of the same process may be given the chance to use the time slot. Hence, the process can efficiently use the resources allocated to it. If its not multi-threaded system, the process is bound to loose the time slot.

An operating system that supports this behavior is a multi-threaded operating system and a programming language that gives constructs/API to create and manage threads is called a multi-threaded programming language. Java has in-built support to the multi-threading.

Life Cycle of Thread

In Java, the life cycle of a thread includes the following states:

- NEW - Thread is just created.
- RUNNABLE - Thread is allocated with the required resources and waiting for the CPU.
- WAITING - Thread is waiting for notifications or for joining threads.
- TIMED_WAITING - Thread is waiting for notifications or for joining threads with a timeout.
- BLOCKED – Thread is waiting for acquiring lock.
- TERMINATED – Thread completes its execution.

Thread characteristics

A priority is associated with each of the threads. It ranges from 1 to 10. Default priority is 5. Thread with higher priority gets the access to resources, first. If the OS employs preemptive scheduling, higher priority threads preempts lower priority threads.

Every thread belongs to a named thread group. Threads in a given group can be managed collectively. For example, with a single call, all the threads in a group may be stopped.

A thread always starts as part of its owning process. Threads that execute even after the owning process exits are called daemon threads. Usually batch jobs run as daemon threads. A daemon thread runs when there is no non-daemon thread competing for the resources.

Every thread have an ID issued by the system. These IDs are unique among the threads in the system. An ID may be reused.

Every thread have a name which needs not be unique. Threads get their names either programmatically or by the system.

Main Thread

A Java object of `Thread` class represents a thread of execution. Java programs always have at least one thread that starts as part of its famous `main` method. Its priority is 5.
Current Thread can be accessed using the static method `currentThread`() of `Thread`.

Applying the Concepts

Objective

To print various characteristics of the currently running main thread.

Listing 1: MainThread.java

```
1. package com.glarimy.threads;
2.
3. public class MainThread {
4.     public static void main(String[] args) {
5.         Thread main = Thread.currentThread();
6.         System.out.println("Name: " + main.getName());
7.         System.out.println("Priority: " + main.getPriority());
8.         System.out.println("ID: " + main.getId());
9.         System.out.println("State: " + main.getState());
10.        System.out.println("ThreadGroup: " + main.getThreadGroup());
11.        System.out.println("Thread Count: " + Thread.activeCount());
12.        System.out.println("Alive? " + main.isAlive());
13.        System.out.println("Daemon? " + main.isDaemon());
14.        System.out.println("Interrupted? " + main.isInterrupted());
15.    }
16.}
```

Explanation

At line 5, we are getting hold of the current thread, i.e. the main thread. And there after we are probing it to find the details of the thread.

Results

Running the above listing results in the following typical output.

```
Name: main
Priority: 5
ID: 1
State: RUNNABLE
ThreadGroup: java.lang.ThreadGroup[name=main,maxpri=10]
Thread Count: 1
Alive? true
Daemon? false
Interrupted? False
```

Going Beyond the Concepts

Interesting Questions

How many JVM threads start when running a simple Java program which is not multi-threaded?

Running a Java program always results in running a JVM. JVM itself is a multi-threaded application. The program it is running is one of them. The other threads run garbage collectors and etc., Therefore, running even a single-threaded program results in creating more than one thread, always, at JVM level.

As threads are less expensive, can we create as many number of threads as we like?

Though they are less expensive, its only in comparison with the processes. Threads still cost system resources. So, its better to have a limit on the number of threads. Good idea is to use a pool of threads with limit on maximum permissible number of threads.

Coding Exercises

1. Change the name of the main thread to 'FirstThread'.
2. Change the priority of the thread to 3. See what's the impact.
3. Make the current thread as a daemon thread. Does it make sense?

Quiz

1. When a non-daemon thread is waiting for CPU, can a daemon thread gets the CPU under any circumstances?
2. How is a thread group useful from the programming point of view?
3. Can a daemon thread have priority of 10?
4. Can thread names be duplicate?
5. Is Garbage Collector of Java a daemon?

Chapter 2

Creating User Defined Threads

- Creating User Defined Threads
- Using the `Thread` class
- Starting the threads
- Running a task as part of a thread

Understanding the Concepts

Though Java runs its own threads, application may want to have one or more threads other than the main thread. Probably the application want to run few independent tasks in those threads.

Creating a child Thread

One way of creating a thread is by instantiating the `Thread` class, as shown below.

```
Thread child = new Thread();
```

The main thread becomes parent of this new child thread. It inherits the priority of its parent. Unless the parent is a daemon, the child is never a daemon.

Running a child Thread

A thread is started by calling `start` method of the `Thread` instance.

```
child.start();
```

This in turn calls the `run` method of the child. The thread is terminated when `run` method is concluded. The implementation of the `run` method in this thread does, well, nothing, by default. That means, we run a thread, but without any benefit out of it.

Making thread to run a task

Threads can be customized to run a specific task by extending the `Thread` class i.e. by overriding `start` and `run` methods, as required. The task is typically a block of code that can run independent of the rest of the application. Such a block is what usually written in the `run` method.

Receiving results from running a thread

Remember that the `run` method of `Thread` do not return anything. The child threads run at their own time and pace. The parent thread will not be blocked to receive the results from the child thread, by default.

Applying the Concepts

Objective

To create a thread that prints numbers up to the specified limit.

Listing 2: ThreadCounter.java

```
1. package com.glarimy.threads;
2.
3. public class ThreadCounter extends Thread {
4.     private int limit;
5.
6.     public ThreadCounter(int limit) {
7.         this.limit = limit;
```

```
8.      }
9.
10.     @Override
11.     public synchronized void start() {
12.         System.out.println("Starting the CountingThread");
13.         super.start();
14.     }
15.
16.     @Override
17.     public void run() {
18.         int number = 0;
19.         while (limit-- > 0)
20.             System.out.format("CountingThread: %d%n", number++);
21.     }
22.
23.     public static void main(String[] args) {
24.         ThreadCounter thread = new ThreadCounter(10);
25.         thread.start();
26.     }
27. }
```

Explanation

The class `ThreadCounter` is extending the `Thread` class and overriding the `run` method. The `main` method instantiates and starts this thread.

Results

Running the above code prints the following, on the console.

```
Starting the CountingThread
CountingThread: 0
CountingThread: 1
CountingThread: 2
CountingThread: 3
CountingThread: 4
CountingThread: 5
CountingThread: 6
CountingThread: 7
CountingThread: 8
CountingThread: 9
```

Going Beyond the Concepts

Interesting Questions

Once the main thread starts the child thread, will child immediately start?

May not be. By calling the start method on the child thread, the main program merely putting the child in the runnable state. When exactly it runs? It depends on the CPU scheduling policies and the presence of various other threads and their priorities.

Will the main thread waits till the child thread completes its task?

No, by default. Once the child thread is started, the parent and child just continue their work and exits once they are done, without bothering about each other. In case, a thread have to wait till some other thread ends (no matter whether it is child or someone else), we need to specifically code for it.

What if a child thread wants to extend some other class? Java does not support multiple-inheritance, you know!

Creating user defined threads by extending `Thread` class is not generally suggested precisely for this reason. If a thread have to extend some other class, then the solution is to make the thread to implement the interface `Runnable` than extending the `Thread` class. See the next section.

Can I create more than one instance of the user defined threads?

Why not? Technically no one can stop you. However, at some point of time, if the system can not handle the number of thread instances you have created, it hangs. Better if there is a practical limit on the number of thread instances.

Coding Exercises

1. Write a thread with name 'Worker'. This thread should have a `boolean` flag that is initialized as `true`. The job of the worker is to print numbers till the flag is turned `false` by the `main` thread.

2. Write a thread with name 'Worker'. Let it keep printing something in its `run` method. Start it as a daemon thread and exit. Check that the Worker still continues.

3. Create a class ThreadManager whose sole purpose is to instantiate and start a given thread. However, the ThreadManager should not allow more than 3 threads to run at any point of time.

Quiz

1. Which package the `Thread` class is available in?
2. What are the other methods a `Thread` class have to control its life cycle?
3. Can a thread mark itself as a daemon thread?
4. Can the `Thread` object be sent over a network connection?

Chapter 3

Submitting Asynchronous Tasks

- Implementing an asynchronous task
- Submitting the task to a thread
- Running the task in a thread

Understanding the Concepts

A typical multi-threaded application consists of several threads running one or other asynchronous tasks. This can be done by extending the `Thread` class as show in the earlier section. However, it may not be always possible to extend a thread class as Java does not support multiple inheritance.

Implementing an asynchronous task

When its not possible to extend a thread, the task can be separately defined by implementing the interface `Runnable`. The method `run` of the this interface will have the code that implements the task.

For example, the task 'Print' can be implemented as follows:

```
class Print implements Runnable {
   public void run() {
      //your task code goes here
   }
}
```

Submitting and running the task to a thread

Once a runnable task is ready, it can be submitted to a thread for the actual execution. When the thread is started, it invokes the `run` method of the runnable task that is submitted to it.

The following snippet submits the Print task to a thread and starts it.

```
Thread t = new Thread(new Print());
t.start();
```

Applying the Concepts

Demonstration

To create a task that prints numbers up to the specified limit and run it in a thread.

Listing 3: RunnableCounter.java

```
1. package com.glarimy.threads;
2.
3. public class RunnableCounter implements Runnable{
4.     private int limit;
5.
6.     public RunnableCounter(int limit) {
7.         this.limit = limit;
8.     }
9.
10.    @Override
11.    public void run() {
12.        int number = 0;
13.        while (limit-- > 0)
14.            System.out.format("RunnableCounter: %d%n", number++);
15.    }
16.
17.    public static void main(String[] args) {
18.        Runnable job = new RunnableCounter(10);
19.        Thread thread = new Thread(job);
20.        thread.start();
21.    }
22. }
```

Explanation

The class RunnableCounter is the task that implements `Runnable` interface and thereby its `run` method. The `main` thread creates a child thread to submit and run this task.

Results

Running the above program results in the following output.

```
RunnableCounter: 0
RunnableCounter: 1
RunnableCounter: 2
RunnableCounter: 3
RunnableCounter: 4
RunnableCounter: 5
RunnableCounter: 6
RunnableCounter: 7
RunnableCounter: 8
RunnableCounter: 9
```

Going Beyond the Concepts

Interesting Questions

What happens if the main thread directly calls the run method of a Runnable implementation?

The run method gets executed in the context of the main thread only. No new thread gets created.

What if the code within the run method throws an exception?

The run method can not throw exceptions. If the code that it executes throws any exceptions, then the method must catch and handle those exceptions

Coding Exercises

1. Implement a `Runnable` with the name 'Worker'. It should have a `boolean` flag that is initialized as `true`. The job of the worker is to print numbers till the flag is turned `false` by the `main` thread.

2. Implement a `Runnable` with the name 'Worker'. Let it print something in its `run` method. Start it as part of a daemon thread and exit. Check that the Worker still continues.

3. Create a class ThreadManager whose sole purpose is to instantiate and start a given implementation of `Runnable`. However, the ThreadManager should not allow more than 3 threads to run at any point of time.

Quiz

1. Which package the `Runnable` interface is available in?
2. Can an object of `Runnable` be sent over a network connection?

Chapter 4

Controlling Threads

- Starting, suspending, resuming and stopping a thread
- Waiting for a thread
- Making a thread to sleep

Understanding the Concepts

Controlling a thread includes starting, stopping, pausing, resuming, making it to wait or sleep and etc., Of these we already know how to start a thread. Lets see the others.

Suspending, resuming and stopping a thread

Probably the nice way of pausing and stopping a thread is by writing some programming logic rather than depending on the methods that `Thread` class offers for the purpose. They are all deprecated for the following reasons:

> Stopping a thread releases all the resources it is currently using (like files and etc.) to other waiting threads. If the resources are not in a consistent state, program behaves unpredictably.
>
> Suspending a thread may lead to deadlock. Isn't it? If suspending a thread is not good, then resuming it doesn't even arise.

Hence, the way out is to build your own logic than depending on these methods

to stop, suspend and resume threads.

Waiting for a thread

If a thread wants to wait till the other the completes its job, then it can issue the `join` call, as follows.

```
Thread t = new Thread();
t.start();
//the parent code block - 1
t.join();
//the parent code block - 2
```

In the snippet, the parent starts the child thread `t`. At some point, it issues the join call to the child. Till the child thread returns (after running it's `run` method), the parent can not execute its block – 2.

Making the thread to sleep

A thread can be asked to sleep for a specified time. A sleeping thread do not use any CPU time for that duration. After the timeout, the thread becomes runnable again and competes for the CPU. Static method `sleep` is used for this. The time duration is specified in milli-seconds.

Applying the concepts

Objective

To make the `main` thread to delegate a task to a child thread and wait till the child thread completes. The task of the child is to print the numbers up to a specified upper limit with an interval of 1 second.

Listing 4: Delegator.java

```
1. package com.glarimy.threads;
2.
3. public class Delegator {
4.     public static void main(String[] args) {
5.         System.out.println("Main: Creating a thread to count.");
6.         Thread thread = new Thread(new Counter(10));
```

```
7.          System.out.println("Main: Delegating and waiting...");
8.          thread.start();
9.          try {
10.             thread.join();
11.         } catch (InterruptedException e) {
12.             e.printStackTrace();
13.         }
14.         System.out.println("Main: Counting is done. Thanks");
15.     }
16. }
17.
18. class Counter implements Runnable {
19.     private int limit;
20.
21.     public Counter(int limit) {
22.         this.limit = limit;
23.     }
24.
25.     @Override
26.     public void run() {
27.         int number = 0;
28.         while (limit-- > 0) {
29.             System.out.format("Counter: %d%n", number++);
30.             try {
31.                 Thread.sleep(1000);
32.             } catch (InterruptedException e) {
33.                 e.printStackTrace();
34.                 return;
35.             }
36.         }
37.     }
38. }
```

Explanation

The class Counter is a task that implements `Runnable`. It prints numbers up to a specified limit. However, it sleeps for 1 second between successive prints.

The `main` method of the Delegator class submits the Counter task to a thread and starts it. Then it waits till the task completed by joining with the the child thread.

Results

Running the above code results in the following output.

```
Main: Creating a thread to count.
Main: Delegating and waiting...
Counter: 0
Counter: 1
Counter: 2
Counter: 3
Counter: 4
Counter: 5
Counter: 6
Counter: 7
Counter: 8
Counter: 9
Main: Counting is done. Thanks
```

Going Beyond the Concepts

Interesting Questions

Can we make a thread to sleep for less than a milli-second?

Yes. The overloaded `sleep` method can also take nano seconds as the second parameter.

How to make a thread to wait for other thread only for specified duration?

It can be done by specifying the timeout while issuing the `join` call. The `join` method is overloaded to take no arguments or timeout in terms of milli or nano seconds.

Can the current thread yield to others?

Yes. By calling `yield()` method on the current thread, we can pause the current thread and give a chance to the JVM to award time slot for other prioritized threads. If there is no other such thread waiting for the slot, the current thread immediately resumes.

However, better not to use the `yield` method as there is no way we can test its behavior and its impact.

Once the sleep timeout for a thread, will it resume immediately?

May or may not. Once the sleep time out for a thread, it just gets into the runnable state once again and based on the scheduling algorithm of the CPU, it would get its next time slot.

Coding Exercises

1. Comment the sleep call in the above code and check the impact.
2. Comment the join call in the above code and check the impact.
3. Specify a time-out of 5 seconds in the join call and check the impact.
4. Specify a time-out of 20 seconds in the join call. Will it really wait till 20 seconds even the child concludes after 10 seconds?

Quiz

1. What does the `destroy` method of `Thread` do?
2. Can we interrupt a thread that is currently running?

Chapter 5

Sharing part of an object among threads

- Identifying critical regions
- Marking a block of code as critical region in Java
- Building thread-safety

Understanding the Concepts

In a single-threaded application, a line of code is guaranteed to be executed only by a single thread. However, in a multi-threaded application, a block of code may be executed by more than one thread.

Identifying critical regions

Assume this scenario.

A block is containing 10 lines of code. Thread A is executing the 5th line of the block and lost its time slot. Then Thread B started executing the block and is able to complete it. Once the Thread A regains the time slot, it resumes its execution from the 6th line of the block.

Here, the block is being executed by more than one thread, simultaneously. Sometimes, we want only one thread to execute the block fully before other threads are given a chance to execute the same block even if they get the time slot in between. Such blocks are called as critical regions.

Marking Critical Regions

Once we identify a block of code as critical region, we can mark it so by using the Java keyword `synchronized`. By doing so, the thread that gains the access to the critical region acts as if it got the lock which others can not open. As long as the lock is with the current thread, other threads will have to wait for that lock to be released.

Using Implicit locks

Though there are better alternatives available, one of the very basic way of locking a critical region is by creating an object and using it as the lock. This is called Block level synchronization.

Here is the code snippet.

```
Object obj = new Object(); // some object, to be used as the lock
synchronized (obj){
//critical region
}
```

Thread Safety

A code that is written appropriately to work consistently in a multi-threaded environment is what we call as thread-safe code. Essentially, synchronization brings in thread safety. A thread-safe code makes sure that the critical regions are properly locked and unlocked for consistency.

Code that is used always in a single-threaded environment needs not be thread-safe.

Applying the Concepts

Objective

Creating a Counter task that can be used in a multi-threaded environment. The counter prints five numbers at regular intervals. The thread that starts the counter would continue to hold it till it is done with it. Other threads would simply wait for their turn.

Listing 5: CountingThread

```
1. package com.glarimy.threads;
2.
3. class SharedCounter {
4.     private int value;
5.     private Object lock;
6.
7.     public SharedCounter() {
8.         lock = new Object();
9.     }
10.
11.    public void count(long delay) {
12.
13.        String name = Thread.currentThread().getName();
14.        synchronized (lock) {
15.            for (int i = 0; i < 5; i++, value++) {
16.                System.out.println(name + " : " + value);
17.                try {
18.                    Thread.sleep(delay);
19.                } catch (InterruptedException e) {
20.                }
21.            }
22.        }
23.    }
24.}
25.
26.class CountingThread extends Thread {
27.    private SharedCounter counter;
28.    private long delay;
29.
30.    public CountingThread(SharedCounter counter, long delay) {
31.        this.counter = counter;
32.        this.delay = delay;
33.    }
34.
35.    @Override
36.    public void run() {
37.        counter.count(delay);
38.    }
39.
40.    public static void main(String[] args) {
41.        SharedCounter counter = new SharedCounter();
42.        CountingThread one = new CountingThread(counter, 2000);
43.        CountingThread two = new CountingThread(counter, 5000);
44.        one.start();
```

```
45.         two.start();
46.     }
47. }
```

Explanation

SharedCounter is a class that prints numbers at specified interval. This counter can be used by any thread. However, the synchronized block can be executed only by one thread at a time.

The CountingThread is a thread that gets an instance of SharedCounter and uses it.

The `main` method creates and starts two instances of CountingThread. Interestingly, both of them are given the same instance of SharedCounter. Now, the race is between these two threads to use the counter. Whichever threads starts using the shared counter first, it holds it till the end. Only then the other thread can start using the shared counter.

Results

The results may vary each time. But, the bottom line is to see that only one thread can execute the synchronized block. Here is one of the possible output:

```
Thread-0 : 0
Thread-0 : 1
Thread-0 : 2
Thread-0 : 3
Thread-0 : 4
Thread-1 : 5
Thread-1 : 6
Thread-1 : 7
Thread-1 : 8
Thread-1 : 9
```

Beyond the Concepts

Interesting Questions

If a thread holds a block level lock, other thread can execute the code in the other blocks of the same shared object?

Yes. Block level synchronization blocks the other threads from executing only the synchronized blocks marked by the locking object.

Synchronization may lead to dead locks, isn't it?

Yes, of course. Think about two threads locking two different blocks while waiting for the other block held by other thread. It would lead to dead lock. The deadlocks must be prevented while designing and developing multi-threaded applications.

So, synchronization is bad. Why should we use it?

Without synchronization, the application may become inconsistent. With synchronization without proper care, it may lead to deadlock. The solution is to use synchronization with care.

Marking blocks synchronized would make applications thread-safe. Why do not we make every block as synchronized? Isn't a good idea?

Surely not. Synchronization is a costly affair. The JVM will have to track which thread is holding which lock, which other threads are waiting for that lock and etc., It slows down the application. You must mark it synchronized only if the block must not be executed by more than one thread at a time. In all other cases, you must not, for better performance.

Is there any Java class that is thread-safe?

Collection classes like `Vector` are thread safe while `ArrayList` is not thread safe. `StringBuffer` is thread safe while `StringBuilder` is not thread safe.

Coding Exercises

1. Comment out the start and end of the synchronized block and re-run the application. What is the impact? How can you explain it?
2. Implement the Singleton pattern to work in multi-threaded environment without using `Enum`.

Quiz

1. If a thread could not obtain a block level lock, it gets into which of the states?
2. If more than one thread wait for the lock, which of them obtains it once lock is freed-up by the current thread?
3. Is there a way for a thread to wait for a lock only up to certain time?

Chapter 6

Sharing the whole object among threads

- How to lock the whole shared object?

Understanding the Concepts

The block level synchronization that we have seen in the earlier section blocks only the specific block. However, in some circumstances, we would like to block the entire object from being used by any other thread. Method level synchronization essentially achieves that objective.

Method Level Synchronization

By marking a method synchronized, the thread that executes the method would block the entire object from other threads. Other threads can not execute any of the synchronized blocks of the same object.

Applying the Concepts

Objective

Developing a shared counter that is exclusively used by one thread at a time.

Listing 6: CountingThread.java

```
1. package com.glarimy.threads;
```

```
2. class SharedCounter {
3.     private int value;
4.
5.     public synchronized void count(long delay) {
6.         String name = Thread.currentThread().getName();
7.         for (int i = 0; i < 5; i++, value++) {
8.             System.out.println(name + " : " + value);
9.             try {
10.                Thread.sleep(delay);
11.            } catch (InterruptedException e) {
12.            }
13.        }
14.    }
15.}
16.class CountingThread extends Thread {
17.    private SharedCounter counter;
18.    private long delay;
19.
20.    public CountingThread(SharedCounter counter, long delay) {
21.        this.counter = counter;
22.        this.delay = delay;
23.    }
24.
25.    @Override
26.    public void run() {
27.        counter.count(delay);
28.    }
29.
30.    public static void main(String[] args) {
31.        SharedCounter counter = new SharedCounter();
32.        CountingThread one = new CountingThread(counter, 2000);
33.        CountingThread two = new CountingThread(counter, 5000);
34.        one.start();
35.        two.start();
36.    }
37.}
```

Explanation

The method count of the SharedCounter is marked as synchronized. Whichever thread that starts executing the method would hold the entire object for itself.

Results
Same as from the earlier demonstration

Chapter 7

Inter-thread communication

- Making the current thread to wait
- Notifying a waiting thread to resume
- Establishing a sort of communication among threads

Understanding the Concepts

By obtaining a lock, the current thread is making other threads (those want to execute the same synchronized object) to wait, which is fine. What if the current thread itself wants to wait for some other reasons? Making the current thread to sleep solves the problem? No. Remember! Sleeping is not waiting. Threads always sleep for a specific duration. Once the time out, thread resumes. What we want to is to make the thread to wait, indefinitely, till someone notifies it.

Making the current thread to wait

In any synchronized block, the current thread can be asked to wait by issuing the `wait` call. The thread that executes this statement would relinquish the CPU and goes to wait state. The thread waits till it receives a notification to return back to runnable state.

Notifying a thread to resume

In any synchronized block of the same object, a thread can issue the calls

`notify()` or `notifyAll()`. The `notify` call notifies the senior most waiting thread on the some object while the `notifyAll` call notifies all the waiting threads for the same object.

Its always suggestible to use `notifyAll` as all the waiting threads get a chance to try their luck in gaining CPU. Otherwise, a hacker may put a monster thread always consuming the `notify` calls thereby denying opportunity for the other eligible threads from getting the CPU.

Establishing communication between threads

With suitable usage of `wait` and `notify` calls, different threads accessing a common shared object can carry on their tasks hand-in-hand. Think about the following.

You have two threads called Producer and Consumer. Every time consumer reads data from the queue, it notifies the producer to write data into the queue. The producer writes the data into the queue and notifes the consumer to read it. With this understanding, consumer needs not to poll the queue to check if new data is available and producer also needs to poll the queue if the old data is consumed.

Applying the concepts

Objective

To develop a counter which can be incremented by a thread named as writer and read by other thread named as reader. The deal is that the reader should not read the same value twice and writer should not increment the value until the reader reads it.

Listing 7: CountingThread.java

```
1. package com.glarimy.threads;
2.
3. class CommonCounter {
4.     private int value;
5.     private boolean okToRead = false;
6.
```

```
7.     public synchronized void read() {
8.        String name = Thread.currentThread().getName();
9.        try {
10.          while (!okToRead) {
11.             System.out.format("%s : waiting %n", name);
12.             wait();
13.             System.out.format("%s : resuming %n", name);
14.          }
15.
16.          Thread.sleep(1000);
17.          okToRead = false;
18.          System.out.format("%s : reading: %d and notifying %n",
19.                name, value);
20.          notifyAll();
21.       } catch (Exception e) {
22.          e.printStackTrace();
23.       }
24.    }
25.
26.    public synchronized void write() {
27.       String name = Thread.currentThread().getName();
28.       try {
29.          while (okToRead) {
30.             System.out.format("%s : waiting %n", name);
31.             wait();
32.             System.out.format("%s : resuming %n", name);
33.          }
34.          value++;
35.          Thread.sleep(5000);
36.          okToRead = true;
37.          System.out.format("%s : written & notifying %n", name);
38.          notifyAll();
39.       } catch (Exception e) {
40.          e.printStackTrace();
41.       }
42.    }
43. }
44.
45. class Reader extends Thread {
46.    private CommonCounter counter;
47.
48.    public Reader(CommonCounter counter) {
49.       this.counter = counter;
50.       this.setName("Reader");
51.    }
52.
```

```
53.    @Override
54.    public void run() {
55.        while (true)
56.            counter.read();
57.    }
58. }
59.
60. class Writer extends Thread {
61.    private CommonCounter counter;
62.
63.    public Writer(CommonCounter counter) {
64.        this.counter = counter;
65.        this.setName("Writer");
66.    }
67.
68.    @Override
69.    public void run() {
70.        while (true)
71.            counter.write();
72.    }
73. }
74.
75. public class CountingThread {
76.
77.    public static void main(String[] args) {
78.        CommonCounter counter = new CommonCounter();
79.        Reader reader = new Reader(counter);
80.        Writer writer = new Writer(counter);
81.        writer.start();
82.        reader.start();
83.    }
84. }
```

Explanation

CommonCounter is having a boolean flag 'okToRead'.

The read method prints the value only if this flag is set to true. If it is not true, it just waits. Once someone notifies, it re-checks if the flag is really true, reads the value and sets the flag back to false. Once its job is done, it notifies the others saying that value can now be changed.

The write method does exactly opposite. If the flag okToRead is set to false, it goes and increments the value and notify others to read it. If the flag is set to

okToRead, it just waits till some reads and notifies it.

Results

The following is a part of a typical output of this program. You can observe that the values are being read in proper order without missing/repeating any..

```
Writer : written & notifying
Reader : reading: 1 and notifying
Reader : waiting
Writer : written & notifying
Writer : waiting
Reader : resuming
Reader : reading: 2 and notifying
Writer : resuming
Writer : written & notifying
Writer : waiting
Reader : reading: 3 and notifying
Reader : waiting
Writer : resuming
Writer : written & notifying
Writer : waiting
Reader : resuming
Reader : reading: 4 and notifying
Writer : resuming
```

Going Beyond the Concepts

Interesting Questions

What happens to the locks held by the thread when it goes to wait state?

A waiting thread releases the object locks that it holds so that other threads waiting for the same object can work on it. As wait and notify happens programmatically, it is the programmers job to make sure that the system is in consistent state before going to to the wait state.

Why can't the wait and notify calls be made from a non-synchronized method?

These calls are essentially for the threads to exchange the locks between them. Unless the threads are currently holding any lock, there is nothing to wait for and nothing to notify for. By making a method synchronized, the current thread that is executing the method would automatically hold the lock for the entire object. Looking from this point of view, it make sense to have the `wait` and `notify` calls only in the synchronized methods. Isn't it?

In case more than one thread is waiting for the object and all of them receive the notification through `notifyAll`, which of them gets the object (lock) ?

When `notifyAll` is used, all the waiting threads for the object would be notified so that they come out of wait state and enters runnable state. From here, its the scheduling policy that determines who gets the CPU. Usually the thread with the highest priority is the one which gets the CPU and thereby its the one which regains the object.

Coding Exercises

1. Comment the `wait` call in the above code and see the impact.
2. Comment the `notify` call in the above code and see the impact.
3. Comment both `wait` and `notify` calls in the above code and see the impact.
4. Re-write the application to handle a common queue. Writer keeps on writing into the queue as long as there is space and Reader reads and removes the values from the queue as long as there are values. Writer waits if there is no space in the queue. Reader waits if there is no value in the queue.

Quiz

1. Between `notify` and `notifyAll`, which is the better option and why?
2. Can a thread wait only for a specified duration?

Chapter 8

Using Java API to work with locks

- Reentrant Locks
- Java Lock API

Understanding the Concepts

Java locks are reentrant. It means that a thread that holds the lock on an object can again request for another lock on the same object. To put it in a different way, the object that is executing a synchronized method can also execute other synchronized methods of the same object.

Java Lock API

Java provides us with different API besides the low level support (using the keywords like `synchronized`) to work with the locks. The API includes classes like `Lock`, `ReentrantLock` and etc., for this purpose.

Instead of using any random object as the implicit lock, we can create an instance of `Lock` using this API. However, just by creating the object of `Lock` the object will not be locked. For actual locking, we need to call the `lock()` and `unlock()` methods on the object appropriately. Look at the following snippet.

```
Lock lock = new ReentrantLock();
//application code
lock.lock();
```

```
    //critical region
    lock.unlock();
```

Applying the Concepts

Objective

Creating a Counter that can be used by only one thread at a time.

Listing 8: LockedCounter.java

```
1. package com.glarimy.threads;
2.
3. import java.util.concurrent.locks.Lock;
4. import java.util.concurrent.locks.ReentrantLock;
5.
6. class LockedCounter {
7.     private int value;
8.     private Lock lock;
9.
10.    public LockedCounter() {
11.        lock = new ReentrantLock();
12.    }
13.
14.    public void count(long delay) {
15.        lock.lock();
16.        String name = Thread.currentThread().getName();
17.        for (int i = 0; i < 5; i++, value++) {
18.            System.out.println(name + " : " + value);
19.            try {
20.                Thread.sleep(delay);
21.            } catch (InterruptedException e) {
22.            }
23.        }
24.        lock.unlock();
25.    }
26.}
27.
28.class CountingThread extends Thread {
29.    private LockedCounter counter;
30.    private long delay;
31.
32.    public CountingThread(LockedCounter counter, long delay) {
33.        this.counter = counter;
```

```
34.         this.delay = delay;
35.     }
36.
37.     @Override
38.     public void run() {
39.         counter.count(delay);
40.     }
41.
42.     public static void main(String[] args) {
43.         LockedCounter counter = new LockedCounter();
44.         CountingThread one = new CountingThread(counter, 2000);
45.         CountingThread two = new CountingThread(counter, 5000);
46.         one.start();
47.         two.start();
48.     }
49. }
```

Explanation

Its is essentially the same program like what we have already seen in the earlier lising using block level locks. The only difference is that we are using explicit `Lock` object in this version.

Results

Running the code gives the following output.

```
Thread-0 : 0
Thread-0 : 1
Thread-0 : 2
Thread-0 : 3
Thread-0 : 4
Thread-1 : 5
Thread-1 : 6
Thread-1 : 7
Thread-1 : 8
Thread-1 : 9
```

Objective

To create a counter that can be used only by one thread at a time. Other threads must wait only for specified duration to use the counter and leave if they are unsuccessful in gaining the access to the counter.

Listing 9: LazyCoutingThread.java

```
1. package com.glarimy.threads;
2.
3. import java.util.concurrent.TimeUnit;
4. import java.util.concurrent.locks.Lock;
5. import java.util.concurrent.locks.ReentrantLock;
6.
7. class LazyCounter {
8.     private int value;
9.     private Lock lock;
10.
11.    public LazyCounter() {
12.        lock = new ReentrantLock();
13.    }
14.
15.    public void count(long delay) {
16.        String name = Thread.currentThread().getName();
17.        try {
18.            if (lock.tryLock(1, TimeUnit.SECONDS) == false) {
19.                System.out.println(name + " : Giving up ...");
20.                return;
21.            }
22.            for (int i = 0; i < 5; i++, value++) {
23.                System.out.println(name + " : " + value);
24.                Thread.sleep(delay);
25.            }
26.        } catch (InterruptedException e1) {
27.            System.out.println(name + " : Backing off ...");
28.        }
29.        lock.unlock();
30.    }
31.}
32.
33.class LazyCountingThread extends Thread {
34.    private LazyCounter counter;
35.    private long delay;
36.
```

```
37.     public LazyCountingThread(LazyCounter counter, long delay) {
38.         this.counter = counter;
39.         this.delay = delay;
40.     }
41.
42.     @Override
43.     public void run() {
44.         counter.count(delay);
45.     }
46.
47.     public static void main(String[] args) {
48.         LazyCounter counter = new LazyCounter();
49.         LazyCountingThread one
50.             = new LazyCountingThread(counter, 2000);
51.         LazyCountingThread two
52.             = new LazyCountingThread(counter, 5000);
53.         one.start();
54.         two.start();
55.     }
56. }
```

Explanation

This is just a small variation from the earlier listing. Instead of waiting for the locks indefinitely, it waits only for up to a specified time.

Results

```
Thread-0 : 0
Thread-1 : Giving up ...
Thread-0 : 1
Thread-0 : 2
Thread-0 : 3
Thread-0 : 4
```

Beyond the Concepts

Interesting Questions

When many other threads are waiting for accessing a reentrant lock, which of them gets it? Or what is `fairness`?

By default, there is no specific order. However, as part of the construction of reentrant lock, if `fairness` is set to `true`, then the longest waiting thread would get the access to the lock. However, it may be that same thread may access the lock again and again which leads to lower over-all through-put from the application.

If an application just want to lock the application for reading the data, what is the point in blocking the other threads who also want to just read the data? Is there any way to block only simultaneous write operations but allow simultaneous read operations?

Yes, it is easily possible using the API. Instead of using the `ReentrantLock`, one can use `ReadWriteLock` implementation. One such implementations is `ReentrantReadWriteLock`. This class offers `ReadLock` and `WriteLock` separately. Many reader threads can hold intances of the `ReadLock`. But, only one writer thread can hold `WriteLock`. No thread can get even `ReadLock` when a thread has a `WriteLock`.

Coding Exercises

1. Re-write the application using the `fairness` parameter.
2. Re-write the application using the `ReadWriteLock` API.

Quiz

1. What is the package that offers the lock API?
2. Which version of JDK offers this API?
3. Is the `Lock` a super interface of `ReadWriteLock`?
4. By default, the `Lock` object deals with the `fairness` or not?

Chapter 9

Working with Thread Pools

- Pooling the threads
- Using Executor Service

Understanding the Concepts

In theory, it is possible to create any number of thread instances to run tasks. However, the more the number of threads, more the overhead. The application may stop responding after some time because it is busy in managing the threads. Instead of brining down the system, it make sense to limit the number of threads that can run at any point of time. Let us see how it can be achieved.

Introducing Thread Pools

A thread itself does nothing, we know! It merely executes a runnable task that is handed over to it. Then, how about creating a thread manager who takes the charge of creating threads and submitting the tasks? Such a manager itself can be a thread. It manages a collection of threads. This collection is popularly called as Thread Pool and the threads that are being managed are called as worker threads. Lets see how it works.

A manager creates the threads and submit the tasks. The manager may have a fixed set of worker threads always or it may keep on creating worker threads on need basis up to a maximum limit. If more threads can not be created, those

tasks may be queued up. Once a worker thread completes its job, the manager can create a new thread and submit a task from the queue.

Executor Service

Java in JDK 5 introduced an interesting feature called `ExecutorService` which is essentially such a manager. An executor service can be created using the factory `Executors`. Here is a typical usage assuming that the MyTask is an implementation of `Runnable`.

```
ExecutorService executor = Executors.newFixedThreadPool();
executor.submit(new MyTask());
//application code
executor.shutdown();
```

Shutting down the executor would not stop the running threads. It wait till the submitted tasks complete.

Applying the Concepts

Objective

To develop a counter that prints numbers up to a limit. However, this version should run the counter using an executor.

Listing 10: RunnableCounter.java

```
1.  package com.glarimy.concurrency;
2.
3.  import java.util.Date;
4.  import java.util.concurrent.ExecutorService;
5.  import java.util.concurrent.Executors;
6.  import java.util.concurrent.RejectedExecutionException;
7.
8.  public class RunnableCounter implements Runnable {
9.     private int limit;
10.
11.    public RunnableCounter(int limit) {
12.       this.limit = limit;
```

```
13.    }
14.
15.    @Override
16.    public void run() {
17.       int number = 0;
18.       while (limit-- > 0) {
19.          try {
20.             Thread.sleep(1000);
21.          } catch (InterruptedException e) {
22.
23.          }
24.          System.out.println("RunnableCounter: " + number++);
25.       }
26.    }
27.
28.    public static void main(String[] args) {
29.
30.       ExecutorService pool = Executors.newFixedThreadPool(3);
31.       System.out.println("Submitted the task at " + new Date());
32.       pool.submit(new RunnableCounter(10));
33.       System.out.println("Requested shutdown at " + new Date());
34.       pool.shutdown();
35.       try {
36.          System.out.println("Trying to submit new tasks...");
37.          pool.submit(new RunnableCounter(2));
38.       } catch (RejectedExecutionException e) {
39.          System.out.println("Executor refuses new tasks!");
40.       }
41.    }
42.}
```

Explanation

The counter is the same that we have used earlier. Nothing is changed. However, instead of submitting it to a `Thread` directly, we submit it to an `ExecutorService`. The service itself is created as thread pool with a fixed size of three.

Soon after submitting the task, the service is requested to shut down. However, it waits till the submitted task is completed. However, after shutdown request, it would not accept any more new submissions even if still waits the current tasks to be completed. It throws `RejectedExecutionException`.

Results

Running this application gives the following results. Note that the `ExecutorService` is taking time to shutdown and rejecting any new tasks in the meantime.

```
Submitted the task at Fri Mar 04 14:35:45 IST 2011
Requested shutdown at Fri Mar 04 14:35:45 IST 2011
Trying to submit new tasks...
Executor refuses new tasks!
RunnableCounter: 0
RunnableCounter: 1
RunnableCounter: 2
RunnableCounter: 3
RunnableCounter: 4
RunnableCounter: 5
RunnableCounter: 6
RunnableCounter: 7
RunnableCounter: 8
RunnableCounter: 9
```

Going Beyond Concepts

Interesting Questions

Can we force the `ExecutorService` to shutdown immediately?

Yes. Instead of `shutdown()`, use `shutdownNow()`. The executor immediately issues the interrupt call to all the active threads. It also issues `stop` call to all the active as well as waiting threads. Finally, it returns the list of threads that never commenced their execution.

Thats little scary. Can't the executor wait at least till the active threads to terminate gracefully (not until completion of the task)?

It is possible. Use `awaitTerminate()` method to instruct the executor to wait till the timeout before shutting down. The active threads should terminate (before the time out expires) in response to the interrupt call.

Is there any way to instruct the executor to accept tasks in bulk?

It is not possible to do with `Runnable` tasks, but can be done through `Callable` tasks which we are going to see in the next section.

Coding Exercises

1. Re-write the above application using `SingleThreadExecutor` instead of Fixed Thread Pool.

2. Create an application that creates large number of threads. Run the application with and without thread pools and check the performance of the application.

Quiz

1. A cached thread pool reuses a thread. Yes or no?
2. Can a single application use synchronization primitives, locks and executors together?

Chapter 10

Receiving results from the tasks

- Creating and running Callable tasks
- Working Future results

Understanding the Concepts

So far, all the `Runnable` tasks that are being submitted to a `Thread` or an `Executor` simply run and vanish without returning any result back to the caller. Sometimes, the caller may want to receive some result by running a task. However, the caller do not want to wait till the task completes. If caller has to wait for the want of task result, then there is no point in running the task in a separate thread. The way out is by using `Callable` tasks.

Callable tasks

A task can be defined as callable task by implementing `Callable` interface instead of `Runnable` interface. This new interface has only one method `call()` to be implemented. So, naturally the task definition would go into this method. That way, its just like the `run()` of `Runnable`. The comparison ends here.

- The `call` method returns a computed result, `run()` method don't.
- The `call` method throws exceptions, `run` method don't.

These couple of variations go a long a way, in fact.

Running the Callable tasks

`Callable` tasks can be run only through submission to an `Executor` Service.

Receiving results from Callable tasks

Up on submission of a task, the `Executor` service returns a `Future` object which would be populated with the result at some point of time.

The Future object basically represents the task for querying its status and the end result.

One can check this object to see if the task is completed or canceled, at any future point of time. One can also cancel the task through the `Future`. And finally, the end result can be queried from the `Future` using the get method.

Applying the Concepts

Objective

Run a task as a separate thread and check the time when its completed.

Listing 11: CallableCounter.java

```
1. package com.glarimy.concurrency;
2.
3. import java.util.Date;
4. import java.util.concurrent.Callable;
5. import java.util.concurrent.ExecutorService;
6. import java.util.concurrent.Executors;
7. import java.util.concurrent.Future;
8.
9. public class CallableCounter implements Callable<Date> {
10.    private int limit;
11.
12.    public CallableCounter(int limit) {
13.       this.limit = limit;
14.    }
15.
16.    @Override
17.    public Date call() throws Exception {
```

```
18.        int number = 0;
19.        while (limit-- > 0) {
20.           Thread.sleep(1000);
21.           System.out.println("CallableCounter: " + number++);
22.        }
23.        return new Date();
24.     }
25.
26.     public static void main(String[] args)throws Exception {
27.        ExecutorService pool = Executors.newFixedThreadPool(3);
28.        System.out.println("Submitted the task at " + new Date());
29.        Future<Date> result = pool.submit(new CallableCounter(10));
30.        System.out.println("Requested shutdown at " + new Date());
31.        pool.shutdown();
32.        System.out.println("Tasks ends at " + result.get());
33.     }
34. }
```

Explanation

As expected, the task is implemented as a `Callable` task. The `call` method returns a `Date` object once it is done with its task.

The `Executor` service gives a `Future` object which is queried for the result.

Result

Running the above application gives the following result. It would vary in terms of exact time stamps on your machine.

```
Submitted the task at Fri Mar 04 14:55:04 IST 2011
Requested shutdown at Fri Mar 04 14:55:04 IST 2011
CallableCounter: 0
CallableCounter: 1
CallableCounter: 2
CallableCounter: 3
CallableCounter: 4
CallableCounter: 5
CallableCounter: 6
CallableCounter: 7
CallableCounter: 8
CallableCounter: 9
Task ends at Fri Mar 04 14:55:14 IST 2011
```

Going Beyond the Concepts

Interesting Questions

What sort of result the `get()` method of `Future` object returns if the task is yet to complete?

The `get()` method is a blocking call. Once you call the `get()` method, it return only when the task is completed. So, care must be taken. Better to check if the task is already completed by using the method `isDone()` before calling the `get()` method. Or at least use other version of the `get()` method that takes timeout as a parameter.

What happens if we try to cancel a completed task using the `cancel()` method of its `Future` object?

Nothing. The `cancel()` method simply returns `false`. In fact, `false` is returned when cancellation fails for any reason. The reasons could be that the task is already completed or canceled. Or the task can not be canceled for whatever reasons.

Is it possible to submit just only `Runnable` tasks to the executor but still get a `Future` object?

Yes. In fact, every submission to the `ExecutorService` irrespective of weather the task is `Callable` or `Runnable`, it returns a `Future` object. In case of `Runnable`, this `Future` may be useful only to cancel it or to check the status as there would be no result returned.

Coding Exercises

1. Enhance the application to check if the task is completed before fetching the result.
2. Enhance the application to fetch the result without blocking forever.
3. Rewrite the application using `Runnable` task to see what is possible and what's not.

Quiz

1. `Callable` extends `Runnable` interface. Yes / no?
2. Who receives the exceptions from `call` method? `Executor` service or the `Future` object?

Chapter 11

Scheduling tasks

- Working with time bound tasks
- Choosing appropriate executor
- Various scheduling methods

Understanding the Concepts

A time bound task is simply a task that must be invoked at some point of time in future. The task itself do not know anything about the timing. The invoker should handle it. So any `Runnable` or `Callable` task can be a time bound task.

Choosing appropriate executor

As executors comes in handy to invoke the tasks through submissions, lets see how we can make the tasks time-bound tasks.

The `Executors` factory gives, apart from other pools, an interesting pool called Scheduled Thread Pool. A `ScheduledExecutorService` can work with is pool. This service can take care of timing of the thread invocation.

Scheduling Methods

The scheduled executor service can invoke a `Runnable` or `Callable` task after specified delay. It also can invoke the task repeatedly at a fixed rate or delay.

Applying the Concepts

Objective

To run a task for every 10 seconds.

Listing 12: ScheduledCounter

```
1. package com.glarimy.concurrency;
2.
3. import java.util.Date;
4. import java.util.concurrent.Executors;
5. import java.util.concurrent.ScheduledExecutorService;
6. import java.util.concurrent.TimeUnit;
7.
8. public class ScheduledCounter implements Runnable {
9.     int number = 0;
10.
11.    @Override
12.    public void run() {
13.        System.out.println("CallableCounter: " + number++);
14.
15.    }
16.
17.    public static void main(String[] args) throws Exception {
18.        ScheduledExecutorService pool
19.            = Executors.newScheduledThreadPool(2);
20.        System.out.println("Submitted the task at " + new Date());
21.        pool.scheduleAtFixedRate(new ScheduledCounter(), 1, 10,
22.                TimeUnit.SECONDS);
23.        System.out.println("Requested shutdown at " + new Date());
24.        // pool.shutdown();
25.    }
26.}
```

Explanation

ScheduleCounter is just a normal `Runnable`. It prints a number. However, it is submitted to a `ScheduledExecutorService` as a repeated task. The initial delay is 1 second and the frequency is for every 10 seconds, after the initial delay.

Result

Here is a part of a typical result by running this application. Observe that the task is being executed repeatedly. The task itself do not have any loop to print the numbers.

```
Submitted the task at Fri Mar 04 15:27:02 IST 2011
Requested shutdown at Fri Mar 04 15:27:03 IST 2011
CallableCounter: 0
CallableCounter: 1
CallableCounter: 2
CallableCounter: 3
```

Beyond the Concepts

Interesting Questions

What if the repeated task throws an exception? It will still be repeated at that rate?

No. Once the task throws an `Exception`, subsequent invocations would be suppressed.

What happens in case the period between two subsequent invocations of a task is less that the time task actually takes?

The subsequent invocation would be delayed if a task is still running.

Is there any way to specify number of iterations or time after which no repetitions are required?

No. The only way to stop a repeated task is by canceling it. By terminating the executor, all the scheduled tasks submitted to that executor would be terminated.

Coding Exercises

1. Modify the application to check the result of each of the invocations using `Future` objects.
2. Modify the application to stop the task after 20 iterations.

A QUICK LOOK AT

JAVA THREADS

First Edition

Covers Threads and Concurrency API of JDK 1.6

- Thread
- Runnable
- Callable
- Thread API
- Synchronization
- Inter-thread communication
- Locks
- Executors
- Scheduling

With complete code illustrations.