Chapter 9: Threads

Part I

A Programmer's Guide to Java Certification (Second Edition)

Khalid A. Mughal and Rolf W. Rasmussen
Addison-Wesley, 2nd. Edition, 2003.
ISBN 0-201-72828-1

E-mail: `pgjc2e@ii.uib.no`
Web site: `http://www.ii.uib.no/~khalid/pgjc2e/`

Modified: 8/3/06

9.1 Multitasking

- Multitasking allows several activities to occur concurrently on the computer.
- *Process-based multitasking*
  – allows processes (i.e., programs) to run concurrently on the computer.
- *Thread-based multitasking*
  – allows parts of the *same* program to run concurrently.
  – The sequence of code executed for each task defines an *independent* path of execution, called a *thread* (*of execution*).
- Some advantages of thread-based multitasking:
  – threads share the same address space
  – context switching between threads is usually less expensive than between processes
  – cost of communication between threads is relatively low
- Java supports thread-based multitasking and provides high-level facilities for multithreaded programming.
- *Thread safety*: design of classes to ensure that the state of their objects is always consistent, even when the objects are used concurrently by multiple threads.

9.2 Overview of Threads

- *A thread is an independent sequential path of execution within a program.*
- Threads exist in a common memory space, i.e share both data and code, and the process running the program (*lightweight threads*).
- Every thread in Java is created and controlled by a unique object of the `java.lang.Thread` class.
- Important aspects of multithreaded programming:
  - creating threads and providing the code that gets executed by a thread.
  - accessing common data and code through synchronization.
  - transitioning between thread states.

The Main Thread

- The runtime environment distinguishes between *user threads* and *daemon threads*.
  - As long as a *user thread* is alive, the JVM does not terminate.
  - A *daemon thread* is stopped if there are no more user threads running, thus terminating the program.
  - The status of the thread as either daemon or user can be set before the thread is *started*.
- The *main thread* is a user thread which is automatically created to execute the `main()` method.
  - C*hild* threads are spawned from the main thread, inheriting its user-thread status.
  - The `main()` method can finish, but the program will keep running until all the user threads have finished.
  - Marking all spawned threads as daemon threads ensures that the application terminates when the main thread dies.
- When a GUI application is started, a special user thread is automatically created to monitor the user–GUI interaction.
  - allows interaction between the user and the GUI, even though the main thread might have died after the `main()` method finished executing.

9.3 Thread Creation

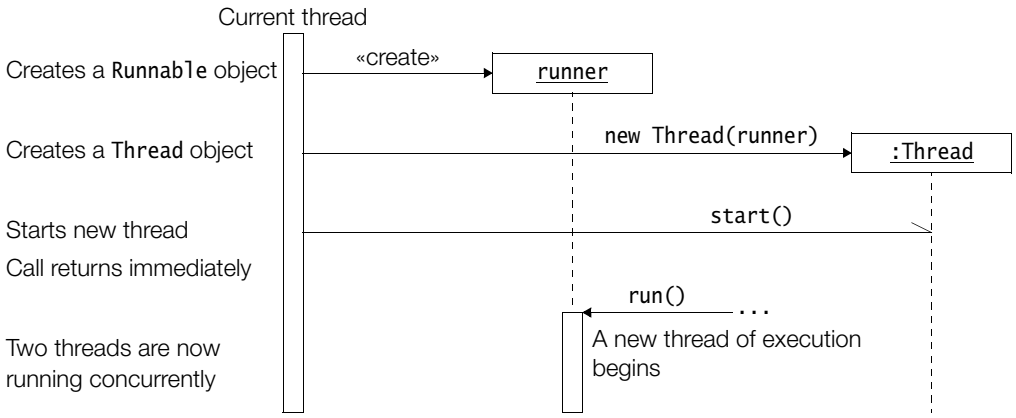
- A thread in Java is represented by an object of the `Thread` class.
- Implementing threads is achieved in one of two ways:
  – implementing the `java.lang.Runnable` interface
  – extending the `java.lang.Thread` class

Implementing the `Runnable` Interface

```
public interface Runnable {
    void run();
}
```

- A thread, which is created based on an object that implements the `Runnable` interface, will execute the code defined in the public method `run()` of this object.
- The `run()` method defines the entry and the exits for the thread.
- The thread ends when the `run()` method ends, either by normal completion or by throwing an uncaught exception.

Figure 9.1 *Spawning Threads Using a* `Runnable` *Object*



Constructors and methods of the java.lang.Thread class:

```
Thread(Runnable threadTarget)
Thread(Runnable threadTarget, String threadName)
```

The argument `threadTarget` is the object whose `run()` method will be executed when the thread is started.

```
static Thread currentThread()
```

This method returns a reference to the `Thread` object of the currently executing thread.

```
final String getName()
final void setName(String name)
```

The first method returns the name of the thread. The second one sets the thread's name.

```
void run()
```

Subclasses of the `Thread` class should override this method. If the current thread is created using a separate `Runnable` object, then the `Runnable` object's `run()` method is called.

```
final void setDaemon(boolean flag)
final boolean isDaemon()
```

Allow the status (user or deamon) to be set and read, respectively

```
void start()
```

This method *spawns* a new thread, that is, the new thread will begin execution as a child thread of the current thread. The call is asynchronous, i.e. returns immediately. It throws an `IllegalThreadStateException` if the thread was already started.

Example 9.1: Implementing the Runnable interface

- The constructor for the `Counter` class ensures:
  - that each object of the `Counter` class will create a new thread by passing the `Counter` instance to the `Thread` constructor.
  - that the thread is enabled for execution by the call to its `start()` method.
  - that the class defines the `run()` method that constitutes the code executed by the thread.
- The `Client` class creates an object of class `Counter` at (5) and retrieves its value in a loop at (6).
- Note that the main thread executing in the `Client` class sleeps for a longer time between iterations than the `Counter` thread.
  - The `Counter` thread is a *child* thread of the main thread.
  - It inherits the user-thread status from the main thread.
- If the code after statement at (5) in the `main()` method was removed, the main thread would finish executing before the child thread.
  - The program would continue running until the child thread completed.

Example 9.1 *Implementing the* `Runnable` *Interface*

```
class Counter implements Runnable {
    private int currentValue;
    private Thread worker;

    public Counter(String threadName) {
        currentValue = 0;
        worker = new Thread(this, threadName);   // (1) Create a new thread.
        System.out.println(worker);
        worker.start();                          // (2) Start the thread.
    }
    public int getValue() { return currentValue; }

    public void run() {                          // (3) Thread entry point
        try {
            while (currentValue < 5) {
                System.out.println(worker.getName() + ": " + (currentValue++));
                Thread.sleep(250);               // (4) Current thread sleeps.
            }
        } catch (InterruptedException e) {
            System.out.println(worker.getName() + " interrupted.");
        }
        System.out.println("Exit from thread: " + worker.getName());
    }
}
```

```
public class Client {
    public static void main(String[] args) {
        Counter counterA = new Counter("Counter A"); // (5) Create a thread.
        try {
            int val;
            do {
                val = counterA.getValue();       // (6) Access the counter value.
                System.out.println("Counter value read by main thread: " + val);
                Thread.sleep(1000);              // (7) Current thread sleeps.
            } while (val < 5);
        } catch (InterruptedException e) {
            System.out.println("main thread interrupted.");
        }
        System.out.println("Exit from main() method.");
    }
}
```


Possible output from the program:

```
Thread[Counter A,5,main]
Counter value read by main thread: 0
Counter A: 0
Counter A: 1
Counter A: 2
Counter A: 3
Counter value read by main thread: 4
Counter A: 4
Exit from thread: Counter A
Counter value read by main thread: 5
Exit from main() method.
```

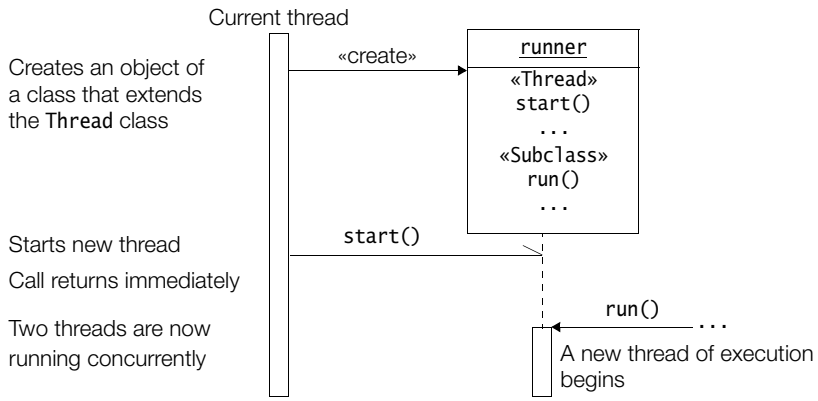
Thread's name (`Counter A`), its priority (`5`), and its parent thread (`main`).

The output from the main thread and the `Counter A` thread is interleaved.

Extending the Thread Class

- A class can also extend the `Thread` class to create a thread:
  1. A class extending the `Thread` class overrides the `run()` method from the `Thread` class to define the code executed by the thread.
  2. This subclass may call a `Thread` constructor explicitly in its constructors to initialize the thread, using the `super()` constructor call.
  3. The `start()` method inherited from the `Thread` class is invoked on the object of the class to make the thread eligible for running.

Figure 9.2 *Spawning Threads—Extending the* `Thread` *Class*



Example 9.2: Extending the Thread class

- Note the call to the constructor of the superclass `Thread` at (1) and the invocation of the inherited `start()` method at (2) in the constructor of the `Counter` class.
- The program output shows that the program continues running until the child threads have completed.
  - The two child threads are independent, each having its own counter and executing its own `run()` method.
- Note that the `Thread` class implements the `Runnable` interface: the roles of the `Runnable` object and the `Thread` object are combined in a single object of the subclass.
- The name of the current thread can be obtained as follows:

```
Thread.currentThread().getName());   // (5) Current thread
```

- Inserting the following code before (2) in Example 9.2:

```
setDaemon(true);
```

illustrates the daemon nature of threads.

Example 9.2 *Extending the* `Thread` *Class*

```
class Counter extends Thread {

    private int currentValue;

    public Counter(String threadName) {
        super(threadName);                          // (1) Initialize thread.
        currentValue = 0;
        System.out.println(this);
        start();                                    // (2) Start this thread.
    }
    public int getValue() { return currentValue; }

    public void run() {                             // (3) Override from superclass.
        try {
            while (currentValue < 5) {
                System.out.println(getName() + ": " + (currentValue++));
                Thread.sleep(250);                  // (4) Current thread sleeps.
            }
        } catch (InterruptedException e) {
            System.out.println(getName() + " interrupted.");
        }
        System.out.println("Exit from thread: " + getName());
    }
}
```

```
public class Client {
    public static void main(String[] args) {
        System.out.println("Method main() runs in thread " +
                Thread.currentThread().getName());   // (5) Current thread
        Counter counterA = new Counter("Counter A"); // (6) Create a thread.
        Counter counterB = new Counter("Counter B"); // (7) Create a thread.
        System.out.println("Exit from main() method.");
    }
}
```

Possible output from the program:

```
Method main() runs in thread main
Thread[Counter A,5,main]
Thread[Counter B,5,main]
Exit from main() method.
Counter A: 0
Counter B: 0
Counter A: 1
Counter B: 1
Counter A: 2
Counter B: 2
Counter A: 3
Counter B: 3
Counter A: 4
Counter B: 4
Exit from thread: Counter A
Exit from thread: Counter B
```


- Implementing the `Runnable` interface may be preferable to extending the `Thread` class:
  - The class implementing the `Runnable` interface is not prevented from extending another class.
  - The class implementing the `Runnable` interface can avoid the full overhead of the `Thread` class which can be excessive.
- Alternate ways of starting a thread:
  - The call to the `start()` method can be factored out of the constructor.
  - The `start()` method can be called later on the thread object.
- Inner classes are useful for implementing threads that do simple tasks:

```
(   new Thread() {
        public void run() {
            for(;;) System.out.println("Stop the world!");
        }
    }
).start();
```

9.4 Synchronization

- Threads share the same memory space, that is, they can share resources.
- Certain situations may require that only one thread at a time has access to a shared resource whose integrity must be maintained.
- Java provides high-level concepts for *synchronization* in order to control access to shared resources.

Locks

- A *lock* (a.k.a. *monitor*) is a mechanism to *synchronize* access to a shared resource.
- At any given time, at the most one thread can hold the lock (i.e., own the monitor) and thereby have access to the shared resource, implementing what is called *mutual exclusion* (a.k.a. *mutex*).
- In Java, *all* objects have a lock—including arrays.
- By associating a shared resource with a Java object and its lock, the object can act as a *guard*, ensuring synchronized access to the resource.
- The *object lock mechanism* enforces the following rules of synchronization:
  - A thread must *acquire* the object lock associated with a shared resource, before it can *enter* the shared resource.
    - No other thread can enter a shared resource if another thread already holds the object lock associated with the shared resource.
    - If a thread cannot immediately acquire the object lock, it is *blocked*, that is, it must wait for the lock to become available.
  - When a thread *exits* a shared resource, the object lock is also relinquished.
    - If another thread is waiting for this object lock, it can proceed to acquire the lock in order to gain access to the shared resource.

- Classes also have a class-specific lock that is analogous to the object lock.
- The keyword `synchronized` and the lock form the basis for implementing synchronized execution of code.
- There are two ways in which execution of code can be synchronized:
  - synchronized methods
  - synchronized blocks

Synchronized Methods

- If the methods of an object should only be executed by one thread at a time, then the declaration of all such methods should be specified with the keyword `synchronized`.
- A thread wishing to execute a `synchronized` method must first obtain the object's lock (i.e., hold the lock) before it can enter the object to execute the method.
  - A lock is achieved by calling the method.
  - If the lock is already held by another thread, the calling thread waits.
  - A thread relinquishes the lock simply by returning from the `synchronized` method.
- Synchronized methods are useful in situations where methods can manipulate the state of an object in ways that can corrupt the state if executed concurrently.
- While a thread is inside a `synchronized` method of an object, all other threads that wish to execute this `synchronized` method or any other `synchronized` method of the object will have to wait.
  - This restriction does not apply to the thread that already has the lock and is executing a `synchronized` method of the object.
  - Such a method can invoke other `synchronized` methods of the object without being blocked.

- The non-`synchronized` methods of the object can of course be called at any time by any thread.
- Static methods synchronize on the class lock.
  - Note that static, non-synchronized methods can be invoked at any time.
  - Synchronization of static methods in a class is independent from the synchronization of instance methods on objects of the class.
- Also note that a subclass decides whether the new definition of an inherited `synchronized` method will remain `synchronized` in the subclass.

Example: Synchronized Methods

- Pushing and popping of stack elements should be mutually exclusive operations.
- A *race condition* occurs when two or more threads simultaneously update the same value, and as a consequence, leave the value in an undefined or inconsistent state.
- In Example 9.3, two threads continually push and pop the same stack.
- *Non-synchronized* scenario:
  - The non-`synchronized` `push()` and `pop()` methods at (2a) and (4a) intentionally sleep at (3) and (5), respectively, between an update and the use of the value in the field `topOfStack`.
  - A race condition occurs, resulting in an uncaught exception terminating the Pusher thread, but the Popper thread continues.
- *Synchronized* scenario:
  - The `synchronized` version of the `push()` and `pop()` methods at (2b) and (4b), respectively, avoids the race condition.
  - The method `sleep`() does not relinquish any lock that the thread might have on the current object.
  - The lock is only relinquished when the `synchronized` method exits, guaranteeing mutually exclusive push-and-pop operations on the stack.

Example 9.3 *Mutual Exclusion*

```
class StackImpl {                                       // (1)
    private Object[] stackArray;
    private int topOfStack;
    public StackImpl(int capacity) {
        stackArray = new Object[capacity];
        topOfStack = -1;
    }
    public boolean push(Object element) {               // (2a) non-synchronized
//  public synchronized boolean push(Object element) { // (2b) synchronized
        if (isFull()) return false;
        ++topOfStack;                                   // Note update before use.
        try { Thread.sleep(1000); } catch (Exception ex) { } // (3) Sleep a little.
        stackArray[topOfStack] = element;
        return true;
    }
    public Object pop() {                               // (4a) non-synchronized
//  public synchronized Object pop() {                  // (4b) synchronized
        if (isEmpty()) return null;
        Object obj = stackArray[topOfStack];
        stackArray[topOfStack] = null;
        try { Thread.sleep(1000); } catch (Exception ex) { } // (5) Sleep a little.
        topOfStack--;
        return obj;
    }
```



```
    public boolean isEmpty() { return topOfStack < 0; }
    public boolean isFull()  { return topOfStack >= stackArray.length - 1; }
}

public class Mutex {
    public static void main(String[] args) {
        final StackImpl stack = new StackImpl(20);  // (6) Shared by the threads.

        (new Thread("Pusher") {                     // (7) Thread no. 1
            public void run() {
                for(;;) {
                    System.out.println("Pushed: " +
                        stack.push(new Integer(2003)));  // Pushing
        }}}).start();

        (new Thread("Popper") {                     // (8) Thread no. 2
            public void run() {
                for(;;) {
                    System.out.println("Popped: " + stack.pop()); // Popping
        }}}).start();

        System.out.println("Exit from main().");
    }
}
```

Possible output from the program (non-sychronized scenario):

```
Exit from main().
...
Pushed: true
Popped: 2003
Popped: 2003
Popped: null
...
Popped: null
java.lang.ArrayIndexOutOfBoundsException: -1
        at StackImpl.push(Mutex.java:15)
        at Mutex$1.run(Mutex.java:41)
Popped: null
Popped: null
...
```


Synchronized Blocks

- A `synchronized` block allows execution of arbitrary code to be synchronized on the lock of an arbitrary object:

  `synchronized (`*<object reference expression>*`) { `*<code block>*` }`

  – The *<object reference expression>* and the braces of the block are mandatory.

- A `synchronized` method is equivalent to the following:

```
public Object pop() {
    synchronized (this) {           // Synchronized block on current object
        // ...
    }
}
```

- Mutual exclusion proceeds analagous to that in a synchronized method.

```
class SmartClient {
    BankAccount account;
    // ...
    public void updateTransaction() {
        synchronized (account) {       // (1) synchronized block
            account.update();          // (2)
        }
    }
}
```

Synchronized blocks in Inner Classes

- An inner object might need to synchronize on its associated outer object, in order to ensure integrity of data in the latter.

```
class Outer {                           // (1) Top-level Class
    private double myPi;                // (2)

    protected class Inner {             // (3) Non-static member Class
        public void setPi() {           // (4)
            synchronized(Outer.this) { // (5) Synchronized block on outer object
                myPi = Math.PI;         // (6)
            }
        }
    }
}
```


Synchronized blocks on a class lock

`synchronized (`*<class name>*`.class) { `*<code block>*` }`

- The block synchronizes on the lock of the object denoted by the reference *<class name>*`.class`.
- A `static synchronized` method `classAction()` in class `A` is equivalent to the following declaration:

```
static void classAction() {
    synchronized (A.class) {       // Synchronized block on class A
        // ...
    }
}
```

- In summary, a thread can hold a lock on an object
  - by executing a synchronized instance method of the object
  - by executing the body of a synchronized block that synchronizes on the object
  - by executing a synchronized static method of a class