Chapter 4 Concurrency

4.1 Write code to define, instantiate, and start new threads using both java.lang.Thread and java.lang.Runnable

What is a thread?

Threads are lightweight processes that appear to run in parallel with your main program. Unlike a process a thread shares memory and data with the rest of the program. The word thread is a contraction of "thread of execution", you might like to imagine a rope from which you have frayed the end and taken one thread. It is still part of the main rope, but it can be separated from the main and manipulated on its own. Note that a program that runs with multiple threads is different from simply starting multiple instances of the same program, because a Threaded program will have access tot he same data within the program.

An example of where threads can be useful is in printing. When you click on a print button you probably don't want the main program to stop responding until printing has finished. What would be nice is that the printing process started running "in the background" and allowed you to continue using the main portion of the program.

It would also be useful if the main program would respond if the printing thread encountered a problem. A common example used to illustrate threads is to create a GUI application that launches a bouncing ball every time a button is clicked. Because of the speed of modern processors, by switching its time between each thread it appears that each ball has exclusive use of the processor and it will bounce around as if it was the only code running on the CPU. Unlike most language threading is embedded at the heart of the Java language, much of it at the level of the ultimate ancestor class called Object. With older languages like C/C++ there is no single standard for programming Threads.

When studying for the Java Programmers exam you need to understand the concept that when a program starts a new thread, the program no longer has a single path of execution. Just because one thread A starts running before thread B, it does not mean that thread A will finish executing before thread B, and it certainly does not mean that thread B will not start before thread A. Thus you could get a question that says something like "what is the most likely output of the following code". The exact output might depend on the underlying operating system or other programs running at the time.

Just because a Threaded program generates a certain output on your machine/operating system combination there may be no guarantee it will generate the same output on a different system. The people who set the exams questions know that it is easy to make unwarranted assumptions based on how it works on the more common platforms (read Windows) and will include questions that test your knowledge of the platform depending nature of Java threading.

Make a careful not of the exact thread objectives of the exam because it expects you to know a narrow range of topics really well, but there are many Thread related topics that the exam does not cover. Thus you do not need to know about thread groups, thread pooling thread priorities and many other thread topics.

The two ways of creating a thread

Of the two methods of creating a new thread the use of Runnable is probably more common, but you must know about both for the purpose of the exam. Here is an example of a class created with the Runnable interface.

```
class MyClass implements Runnable{
        public void run(){
                // Insert the code you want to run in a thread here
        }
}
```

Any class that implements an interface must create a method to match all of the methods in the interface. The methods need not do anything sensible, i.e. they may have blank bodies, but they must be there. Thus I include the method run even in this little example, because you must include a run method if you implement Runnable. Not including a run method will cause a compile time error.

The other method for creating a thread is to create a class that is descended from Thread. This is easy to do but it means you cannot inherit from any other class, as Java only supports single inheritance. Thus if you are creating a Button you cannot add threading via this method because a Button inherits from the AWT Button class and that uses your one shot at inheritance. There is some debate as to which way of creating a thread is more truly object oriented, but you do need to go into this for the purpose of the exam.

```
class MyThread extends Thread {
        public void run () {
                // Insert the code you want to run in a thread here
        }
}
```

Instantiating and starting a Thread

Although the code that runs in your thread is in a method called run, you do not call this method directly, instead you call the start method of the thread class. This is a really important point that is likely to come up on the exam. It can easily catch you out because it runs against the grain of most Java programming you do. Normally if you put code in a method, you cause that code to execute by calling the method. There are no rules against calling the run method directly, but it will then execute as an ordinary method rather than as part of the thread.

The Runnable interface does not contain a start method, so to get at this and the other useful methods for threads (sleep, suspend etc etc), you pass your class with the Runnable interface as the constructor to an instance of the Thread class.

Thus to cause the thread to execute from a class that implements Runnable you would call the following

```
MyClass mc = new MyClass();
Thread t = new Thread(mc);
t.start();
```

Although it is the run method code that executes, a thread is actually started via the start method

Again note that was a call to start, not a call to run, even though it is the code in the run method in your class that actually executes.

If you create your class as a sub class of Thread you can simply call the start method. The drawback of sub classing the Thread class is that due to only supporting single inheritance you cannot inherit the functionality of any other class.

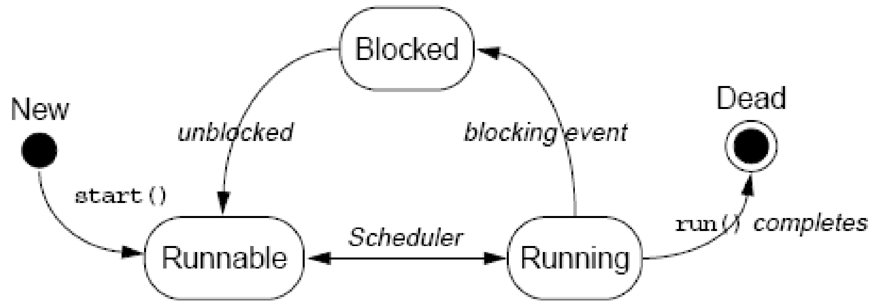
Creating two threads of execution

```
MyClass mc = new MyClass();
MyClass mc2 = new MyClass();
Thread t = new Thread(mc);
Thread t2 = new Thread(mc2);
t.start();
t2.start();
```

Note that that there are no guarantees that thread t will finish execution before thread t2. Of course with no code in the body of the run method it is highly likely that t will finish before t2, but no guarantee. Even if you run the code a thousand or so times on your computer and you get the same order of completion, you cannot be certain that on another operating system, or even with a different set of circumstances on your machine the order of completion will be the same.

Note that the Runnable method of creating a new thread requires an instance of the Thread class to be created, and have the Runnable class passed as a parameter to the constructor.

4.2 Recognize the states in which a thread can exist, and identify ways in which a thread can transition from one state to another



A thread can be in one of the following states:

- **Running**
- **Runnable** - Waiting for a chance to run on the CPU. All threads enter this state before running. Threads with higher priorities may be preventing the thread from getting a chance to run (this depends on how priorities have been implemented in the particular Java Virtual machine)
- **Blocked**
    - sleep() - Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers
    - yield - Causes the currently executing thread object to temporarily pause and allow other threads to execute
    - join() - Waits for this thread to die
- **Dead** - the run() method has completed (the thread can never be restarted)

4.3 Given a scenario, write code that makes appropriate use of object locking to protect static or instance variables from concurrent access problems

The use of these methods, and synchronized methods or code blocks, ensure only one thread at a time can access parts of a class, and control how and when threads get this access.

Using the synchronized keyword to require a thread of execution to obtain an object lock prior to proceeding

Using the synchronized keyword in the method declaration requires a thread obtain the lock for this object before it can execute the method.

```
synchronized void someMethod() { }
```

You can also make a block of code synchronized by using the synchronized keyword followed by the object or class, for which a thread must obtain the lock before it can execute this block of code:

```
// .. some code before the synchronized block
synchronized (someObject) {
        // synchronized code
}
// more code...
```

The wait() method

The wait() method must be used in synchronized code. The thread releases ownership of this monitor and waits until another thread notifies threads waiting on this object's monitor to wake up (i.e. move into Ready state) either through a call to the notify() method or the notifyAll() method. The thread then waits until it can re-obtain ownership of the monitor and resumes execution.

The notify() or notifyAll() methods of an object

notify() moves one thread, that is waiting the this objects monitor, into the Ready state. This could be any of the waiting threads; the choice of which thread is chosen is an implementation issue of the virtual machine.

notifyAll() moves all threads, waiting on this objects monitor, into the Ready state.

4.4 Given a scenario, write code that makes appropriate use of wait, notify, or notifyAll

The following code is an example of thread interaction that demonstrates the use of wait and notify methods to solve a classic producer-consumer problem.

Start by looking at the outline of the stack object and the details of the threads that access it. Then look at the details of the stack and the mechanisms used to protect the stack's data and to implement the thread communication based on the stack's state.

The example stack class, called SyncStack to distinguish it from the core class java.util.Stack, offers the following public API:

```
public synchronized void push(char c);
public synchronized char pop();
```

Producer

The producer thread runs the following method:

```
public void run() {
  char c;

  for (int i = 0; i < 200; i++) {
    c = (char)(Math.random() * 26 +'A');
    theStack.push(c);
    System.out.println("Producer" + num + ": " + c);
    try {
      Thread.sleep((int)(Math.random() * 300));
     } catch (InterruptedException e) {
       // ignore it
     }
   }
 }
```

This example generates 200 random uppercase characters and pushes them onto the stack with a random delay of 0 to 300 milliseconds between each push. Each pushed character is reported on the console, along with an identifier for which producer thread is executing.

Consumer

The consumer thread runs the following method:

```
public void run() {
  char c;
  for (int i = 0; i < 200; i++) {
    c = theStack.pop();
    System.out.println("Consumer" + num + ": " + c);

    try {
      Thread.sleep((int)(Math.random() * 300));
    } catch (InterruptedException e) { }

   }
}
```

This example collects 200 characters from the stack, with a random delay of 0 to 300 milliseconds between each attempt. Each popped character is reported on the console, along with an identifier to identify the consumer thread that is executing.

Now consider construction of the stack class. You are going to create a stack that has a seemingly limitless size, using the ArrayList class. With this design, your threads have only to communicate based on whether the stack is empty.

SyncStack Class

A newly constructed SyncStack object's buffer should be empty. You can use the following code to build your class:

```
public class SyncStack {
        private List buffer = new ArrayList(400);
        public synchronized char pop() {...}
        public synchronized void push(char c) {...}
}
```

There are no constructors. It is considered good style to include a constructor, but it has been omitted here for brevity.

Now consider the push and pop methods. They must be synchronized to protect the shared buffer. In addition, if the stack is empty in the pop method, the executing thread must wait. When the stack in the push method is no longer empty, waiting threads are notified.

The pop method is as follows:

```
public synchronized char pop() {
  char c;
  while (buffer.size() == 0) {
    try {
      this.wait();
    } catch (InterruptedException e) {
      // ignore it...
    }
  }
  c = ((Character)buffer.remove(buffer.size()-1)).charValue();
  return c;
}
```

The wait call is made with respect to the stack object that shows how the rendezvous is being made with a particular object. Nothing can be popped from the stack when it is empty, so a thread trying to pop data from the stack must wait until the stack is no longer empty.

The wait call is placed in a try/catch block because an interrupt call can terminate the thread's waiting period. The wait must also be within a loop for this example. It must wait if its wait is interrupted and the stack is still empty.

The pop method for the stack is synchronized for two reasons. First, popping a character off of the stack affects the shared data buffer. Second, the call to this.wait() must be within a block that is synchronized on the stack object, which is represented by this.

The push method uses this.notify() to release a thread from the stack object's wait pool. After a thread is released, it can obtain the lock on the stack, and continue executing the pop method which removes a character from the stack's buffer.

Note – In pop, the wait method is called before any modifications are made to the stack's shared data. This is important, because the data must be in a consistent state before the object's lock is released and a thread's execution changes the stack's data.

You should also consider error checking. You might notice that there is no explicit code to prevent a stack underflow. This is not necessary because the only way to remove characters from the stack is through the pop method, and this method causes the executing thread to enter the wait state if no character is available. Therefore, error checking is unnecessary.

The push method is similar; it affects the shared buffer and must also be synchronized. In addition, because the push method adds a character to the buffer, it is responsible for notifying threads that are waiting for a non-empty stack. This notification is done with respect to the stack object.

The push method is as follows:

```
public synchronized void push(char c) {
        this.notify();
        Character charObj = new Character(c);
        buffer.addElement(charObj);
}
```

The call to this.notify() serves to release a single thread that called wait because the stack is empty. Calling notify before the shared data actually gets changed is of no consequence. The stack object's lock is released only upon exit from the synchronized block, so threads waiting for that lock can obtain it while the stack data are being changed by the pop method.

SyncStack Example

You must assemble the producer, consumer, and stack code into complete classes. A test harness is required to bring these pieces together. Pay particular attention to how SyncTest creates only one stack object that is shared by all threads.

The following is an example of the SyncTest.java application code:

```
package mod13;

public class SyncTest {

  public static void main(String[] args) {

    SyncStack stack = new SyncStack();

    Producer p1 = new Producer(stack);
    Thread prodT1 = new Thread (p1);
    prodT1.start();

    Producer p2 = new Producer(stack);
    Thread prodT2 = new Thread (p2);
    prodT2.start();

    Consumer c1 = new Consumer(stack);
    Thread consT1 = new Thread (c1);
    consT1.start();

    Consumer c2 = new Consumer(stack);
    Thread consT2 = new Thread (c2);
    consT2.start();
  }
}
```

The following is an example of the Producer.java application code:

```
package mod13;

public class Producer implements Runnable {
  private SyncStack theStack;
  private int num;
  private static int counter = 1;

  public Producer (SyncStack s) {
    theStack = s;
    num = counter++;
  }

  public void run() {
    char c;
    for (int i = 0; i < 200; i++) {
      c = (char)(Math.random() * 26 +'A');
      theStack.push(c);
      System.out.println("Producer" +num+ ": " +c);
      try {
        Thread.sleep((int)(Math.random() * 300));
      } catch (InterruptedException e) {
        // ignore it
      }
    }
  }
}
```

The following is an example of the Consumer.java application code:

```
package mod13;

public class Consumer implements Runnable {
  private SyncStack theStack;
  private int num;
  private static int counter = 1;

  public Consumer (SyncStack s) {
    theStack = s;
    num = counter++;
  }

  public void run() {
    char c;
    for (int i = 0; i < 200; i++) {
      c = theStack.pop();
      System.out.println("Consumer"+num+": " +c);

      try {
        Thread.sleep((int)(Math.random() * 300));
      } catch (InterruptedException e) { }

    }
  }
}
```

The following is an example of the SyncStack.java application code:

```java
package mod13;

import java.util.*;

public class SyncStack {
  private List buffer = new ArrayList(400);

  public synchronized char pop() {
    char c;
    while (buffer.size() == 0) {
      try {
        this.wait();
      } catch (InterruptedException e) {
        // ignore it...
      }
    }
    c = ((Character)buffer.remove(buffer.size()-1)).
        charValue();
    return c;
  }

  public synchronized void push(char c) {
    this.notify();
    Character charObj = new Character(c);
    buffer.addElement(charObj);
  }
}
```

The following is an example of the output from java mod13.SyncTest. Every time this thread code is run, the results vary.

```
Producer2: F
Consumer1: F
Producer2: K
Consumer2: K
Producer2: T
Producer1: N
Producer1: V
Consumer2: V
Consumer1: N
Producer2: V
Producer2: U
Consumer2: U
Consumer2: V
Producer1: F
Consumer1: F
Producer2: M
Consumer2: M
Consumer2: T
```