Workbook 3

Introduction

In the previous two weeks you wrote, and then improved, a Java client which connected to a chat server. This week you will lay the groundwork for building your own Java chat server. You will explore concurrency in greater detail and design the core data structure, a first-in-first-out (FIFO) queue. Next week you will use the queue to hold messages between the threads inside your Java chat server.

Important

An on-line version of this guide is available at:

http://www.cl.cam.ac.uk/teaching/current/FJava

You should check this page regularly for announcements and errata. You might find it useful to refer to the on-line version of this guide in order to follow any provided web links or to cut 'n' paste example code.

Concurrency in Java

Your use of Java threads was somewhat limited in Workbooks 1 and 2. In those Workbooks there was no communication or coordination between threads, nor were there any shared data structures. This week you will explore how to share information between threads safely and how to coordinate thread activities correctly.

A common paradigm used when a program consists of multiple threads is the *producer-consumer* model. In this model one or more threads produce data, and one or more threads consume data. In its simplest form, there is one producer and one consumer. If the producer and consumer are going to share data, they need a shared data structure, and this is typically a FIFO queue. An interface specification for a simple queue can be defined as follows:

```
package uk.ac.cam.your-crsid.fjava.tick3;

public interface MessageQueue<T> {//A FIFO queue of items of type T
 public abstract void put(T msg); //place msg on back of queue
 public abstract T take(); //block until queue length >0; return head of queue
}
```

Notice that this interface takes a generic parameter `T`. You have used interfaces defined in this way before (for example the `java.util.List` interface). You can declare variables of type `MessageQueue` as you might expect:

```
MessageQueue<Integer> queue;
```

A (partial) implementation of the interface `MessageQueue`, called `UnsafeMessageQueue`, is shown below; note how the type parameter `T` is used in the body of the class. As the name implies, `UnsafeMessageQueue` is not *thread-safe*, in other words, the information stored in the queue may be corrupted if it is accessed concurrently by more than one thread. You'll see this in action later in the workbook. Most of the implementation of `UnsafeMessageQueue` should look familiar to you; it's a simple implementation of a linked list. The first two statements at the top of the `take` method may look odd however. Here the programmer is attempting to emulate the blocking nature of some Java library APIs (e.g. the `read` method of the `Socket` object) by using a `while` loop to poll the queue at regular intervals. The `while` loop calls `Thread.sleep` which pauses execution of the current thread for 100 ms. Polling and sleeping is repeated until an item is placed in the queue. The programmer has used sleep to reduce the CPU load of polling to an acceptable level. As you will see, this piece of code is not sensible!

A naive (partial) implementation of `UnsafeMessageQueue`:

```
package uk.ac.cam.your-crsid.fjava.tick3;

public class UnsafeMessageQueue<T> implements MessageQueue<T> {
 private static class Link<L> {
  L val;
  Link<L> next;
  Link(L val) { this.val = val; this.next = null; }
 }
 private Link<T> first = null;
 private Link<T> last = null;

 public void put(T val) {
  //TODO: given a new "val", create a new Link<T>
  //      element to contain it and update "first" and
  //      "last" as appropriate
 }

 public T take() {
  while(first == null) //use a loop to block thread until data is available
   try {Thread.sleep(100);} catch(InterruptedException ie) {}
  //TODO: retrieve "val" from "first", update "first" to refer
  //      to next element in list (if any). Return "val"
 }
}
```

Let's take a look at a simple piece of code which uses `UnsafeMessageQueue`:

```
package uk.ac.cam.your-crsid.fjava.tick3;

class ProducerConsumer {
 private MessageQueue<Character> m = new UnsafeMessageQueue<Character>();
 private class Producer implements Runnable {
  char[] cl = "Computer Laboratory".toCharArray();
  public void run() {
   for (int i = 0; i < cl.length; i++) {
    m.put(cl[i]);
    try {Thread.sleep(500);} catch (InterruptedException e) {
     e.printStackTrace();
    }
   }
  }
 }
 private class Consumer implements Runnable {
  public void run() {
   while (true) {
    System.out.print(m.take());
    System.out.flush();
   }
  }
 }
 void execute() {
  new Thread(new Producer()).start();
  new Thread(new Consumer()).start();
 }
 public static void main(String[] args) {
  new ProducerConsumer().execute();
 }
}
```

Using `UnsafeMessageQueue`

1. Start the Ticklet 3 project on Chime https://www.cl.cam.ac.uk/teaching/current/FJava/ticklet3 and clone a copy of your repository onto your local machine. You should see an implementation of `UnsafeMessageQueue` and `ProducerConsumer`.
2. Complete the implementation of `UnsafeMessageQueue` by filling in the `TODO` sections. Do *not* attempt to make your implementation thread-safe.
3. Test your implementation of `UnsafeMessageQueue` by executing the main method of `ProducerConsumer`.

You probably noticed that `ProducerConsumer` does not terminate. This is because the `Consumer` thread waits forever for a new item from the `Producer` thread, and no new items are forthcoming because the `Producer` thread has terminated. Clearly we need some way for these two threads to coordinate better; something we'll come to later in the workbook. It's also worth noting that the `ProducerConsumer` object accessed methods of the `UnsafeMessageQueue` concurrently, and despite this, the program appeared to behave correctly. This is all too common in multi-threaded programs; code which is not thread-safe works correctly *most of the time*. In fact you have to work quite hard to demonstrate that the implementation of `UnsafeMessageQueue` is not thread-safe.

Demonstrating `UnsafeMessageQueue` is not thread-safe

1. Read the source code of `QueueTest` in your repository and make sure you understand what it's doing and what it is likely to print as output. In particular you should make sure you understand what the method `join` does when invoked on a `Thread` object (read the Java documentation).
2. Compile and run `QueueTest` from within IntelliJ. Be prepared to discuss the functionality of `QueueTest.java` with your Assessor next week.

A safe queue

In the last section you discovered that your queue implementation did not always work correctly; when running `QueueTest` messages sometimes go missing! In general terms, the problem arises due to uncontrolled concurrent threads executing the methods `put` and `take`. This interleaving results in the linked-list data structure being modified in incorrect ways. This can be avoided by using mutual exclusion to prevent more than one thread from running either `put` or `take` simultaneously.

A second problem with `UnsafeMessageQueue` was the use of the `while` statement at the top of the `take` method. Here the programmer is attempting to pause the execution of the `take` method until at least one item of data is available on the queue. This approach is clumsy, since the thread sitting in the `while` loop must wake periodically regardless of whether there is any new data available on the queue, and if new data arrives just after the thread has gone to sleep, then the thread will not awake again until the sleep call has finished. A better way to handle this is to use the *wait-notify* paradigm. The wait-notify paradigm and the `sychronized` paradigm are strongly interconnected in Java. You will first explore `synchronized` paradigm first, and then see how this relates to wait-notify.

The statement `synchronized` is used to provide mutual exclusion in Java, and thus prevent the concurrent execution of multiple statements by more than one thread. A `synchronized` statement is written as follows:

```
synchronized (object) {
 //multiple Java statements
 //...
}
```

Notice that the statement `synchronized` takes an argument called `object` in the above example. In fact `object` can be an instance of any Java object. The Java runtime associates a lock, or *mutex*, with `object`; when a thread enters a `synchronized` collection of statements the lock must be acquired, and when the thread leaves the `synchronized` block, the lock is released. The execution of any other threads which attempt to run code inside a `synchronized` statement while the lock is held by another thread will block until the lock is released and the thread has acquired the lock.

The `synchronized` statement can be used to create a thread-safe version of `UnsafeMessageQueue` by wrapping all the statements inside the `put` and `take` methods inside a `synchronized` statement, and making sure that both the synchronized statements lock on the same object. (If the `put` and `take` methods lock on different objects, then the method bodies will be able to execute concurrently, and you'll have the same problems as before.) One question you might ask is which object should you lock on? In this case a good solution is to use the instance of `UnsafeMessageQueue` itself by presenting `this` as the name for `object`. For example, your implementation of `put` and becomes:

```
public void put(T val) {
 synchronized(this) {
  //... your code here
 }
}
```

It turns out that this is such a common thing to do, that Java provides a convenient alternative notation which is visually less cumbersome. If you want the whole method body to be executed only when holding a lock associated with the instance of an object on which the method is associated, you can prefix the method declaration with the keyword `synchronized`.[1] For example, your implementation of `put` now becomes:

```
public synchronized void put(T val) {
  //... your code here
}
```

In general it is best practice to use thread-safe data structures from the Java library. See, for example, the `java.util.concurrent` package. Nevertheless, writing a thread-safe queue is a good way to practice the mutual exclusion and the wait-notify paradigm in Java, so you will write your own implementation in this course.

First implementation of `SafeMessageQueue`

3. Copy your implementation of `UnsafeMessageQueue` into a new class called `SafeMessageQueue`.

4. Prefix both of your methods `put` and `take` in `SafeMessageQueue` with the keyword `synchronized` to ensure only one thread executes a single method body at any one point in time.

5. Uncomment the lines in the main method of `QueueTest` which test the class `SafeMessageQueue`. Run this program. What happens? Why do you think your implementation of `SafeMessageQueue` doesn't work?

The reason `QueueTest` never terminates is because the program enters a state of *deadlock* inside `SafeMessageQueue`. If `QueueTest` calls `take` when the queue is empty (i.e. `head == null`) then your code asks the thread to sleep for 100 ms in anticipation of more data arriving. Unfortunately this `sleep` method is executed whilst holding the lock (since it's inside the body of a `synchronized` state-

[1] As you discovered in the last Workbook, `java.lang.Class` is a Java object. Therefore the class itself can also be used if you need a single lock for *every* instance of a particular Java class and this is exactly what you get if you prefix a `static` method with the `synchronized` keyword.

ment); any method calls for `put` to add items to the list will be blocked waiting for the method call `take` to finish (and therefore release the lock). The program cannot make progress; the method call `put` is waiting for the method call `take` to complete, and the method call `take` is waiting for the method call `put` to complete.

This problem can be solved by using the wait-notify paradigm mentioned earlier. The paradigm works as follows: when thread A wants to wait for thread B to do something, thread A simply calls the method `wait` on a Java object for which it *already holds the lock*; thread A then releases the lock and pauses execution. Thread B may then acquire the lock that thread A has released, can complete any necessary work, and call `notify` to wake up thread A when any shared data structures are in a consistent state. When thread A reawakes after calling `wait` it first reacquires the lock it gave up, and after that, continues execution. If you attempt to call `wait` or `notify` on a Java object for which you don't hold the lock then you'll receive a `java.lang.IllegalMonitorStateException` at runtime.

The wait-notify paradigm

1. Replace your call to `Thread.sleep` with a call to `this.wait` inside the body of the `take` method of `SafeMessageQueue`.
2. Add a call to the method `this.notify` at the end of the method `put`.
3. Run your version of `QueueTest`. Your implementation of `SafeMessageQueue` should no longer deadlock and it should not lose any messages either!

It's possible for more than one thread to call `wait` on a Java object. In such cases, any subsequent call to `notify` only wakes up one thread, not all of them; if you want to wake up all threads waiting on an object, then you should call `notifyAll`. In general you should use `notifyAll` unless you are absolutely certain that the use of `notify` is correct.

Fine-grained locking

In the last section you used `synchronized` to prevent different threads from simultaneously executing multiple methods on an object. This worked by taking out the lock on a `SafeMessageQueue` object. At first glance this appears to be a little excessive; most of the time the methods `put` and `take` can safely run concurrently; a problem only arises when there are only zero, one or two items in the queue. It is possible to write an implementation of `SafeMessageQueue` which locks instances of `Link` instead. Such an approach is called *fine-grained locking* and can be useful as it has the potential to speed up the execution of multi-threaded programs, especially in a computer which contains more than one processor. Unfortunately it turns out that it's rather tricky to do correctly! So instead, you will explore a simpler example of fine-grained locking from the world of banking; Ticklet 3* explores fine-grained (and no-lock) variants of `SafeMessageQueue`.

A programmer might choose to represent a bank account as an object, and implement a simple method, `transferTo` to express the transfer of funds from one bank account to another. For example:

```
class BankAccount {
 private int balance;
 private int account;
 public void transferTo(BankAccount b, int amount) {
  this.balance -= amount;
  b.balance += amount;
 }
```

Note that `x -= y` is convenient short-hand for `int tmp = x - y; x = tmp;`. Therefore the code the computer will execute actually looks something more like the following:

```
class BankAccount {
 private int balance;
 private int account;
 public void transferTo(BankAccount b, int amount) {
  int tmp1 = this.balance - amount;  // (a)
  this.balance = tmp1;               // (b)
  int tmp2 = b.balance + amount;     // (c)
  b.balance = tmp2;                  // (d)
 }
```

The statements have been labelled (a), (b), (c) and (d). If the program invoking the method `transferTo` has a single thread, then this code will function correctly and whenever `transferTo` is invoked, `amount` is moved from one bank account object to another.

Unfortunately, the method `transferTo` will not function correctly if there are several concurrent transactions taking place from multiple threads. Let us consider what might happen in the case where there are two instances of BankAccount, a and b, and there are two threads running S and T. Thread S executes `a.transferTo(b,10)` and thread T executes `b.transferTo(a,20)`.

To keep the analysis simple, let's assume that both these threads are executed on a computer with a single CPU. On a single CPU computer, only one of the threads can actually run at the same time, and which one it is at any point in time is determined by the scheduler. In general, the scheduler may choose to halt the execution of one thread and start another at any point in time; for example, one thread may have executed only statement (a) of the method `transferTo` before it is taken off the CPU and another thread is scheduled.

Table 1, “An execution trace of `transferTo` by two threads” provides a timeline of two threads S and T, where thread S executes statement (a) before thread T takes control of the CPU and runs statements (a), (b), (c) and (d); finally, thread S regains control of the CPU and executes statements (b), (c) and (d). The instances of `a.bal`[ance] and `b.bal`[ance] are shared between the two threads, whereas the values of `tmp1` and `tmp2` are not shared since they are local variables in the method `transferTo` and therefore exist separately on the stack associated with the thread which invoked the method.

	Thread S			Shared		Thread T		
	`a.transferTo(b,10)`	`tmp1`	`tmp2`	`a.bal`	`b.bal`	`b.transferTo(a,20)`	`tmp1`	`tmp2`
1				100	100			
2	`tmp1 = a.bal-10`	90		100	100			
3		90		100	100	`tmp1 = b.bal-20`	80	
4		90		100	80	`b.bal = tmp1`	80	
5		90		100	80	`tmp2 = a.bal+20`	80	120
6		90		120	80	`a.bal = tmp2`	80	120
7	`a.bal = tmp1`	90		90	80			
8	`tmp2 = b.bal+10`	90	90	90	80			
9	`b.bal = tmp2`	90	90	90	90			
10				90	90			

Table 1. An execution trace of `transferTo` by two threads

At CPU cycle 1 the value of `a.balance` and `b.balance` are both £100. At CPU cycle 10, both threads have finished executing and the value of both `a.balance` and `b.balance` is £90. This doesn't make sense;transfering money between bank accounts may leave different amounts in each bank account, but the total of all accounts should remain constant. In this case the sum total of all accounts has dropped from £200 to £180! What's gone wrong? It turns out that the value of `tmp1`, whilst correctly set by thread S at the end of CPU cycle 2 is incorrect by the time it is used by thread S in CPU cycle 7 since the value of `a.balance` has been changed by the thread T in the meantime.

Another possible execution trace for bank transfers

Table 1, “An execution trace of `transferTo` by two threads” shows only one possible execution trace for two threads S and T. There are many other possible combinations.

4. Look at Table 2 at the end of this workbook. Note that in this table thread S executes statements (a), (b), (c) and (d) in CPU cycles 2, 5, 6, and 7, and thread T executes statements (a), (b), (c) and (d) in CPU cycles 3, 4, 8 and 9 respectively. Is the final result correct or incorrect? Why? Discuss this with your supervisor.

Your final task today is to fix the implementation of `BankAccount` to make use of fine-grained locks inside the method `transferTo` to ensure that money is moved correctly between accounts. In other words, make sure that the `transferTo` method is able to function as part of an Atomic, Consistent, Isolated and Durable[2] (ACID) transaction. Inside the method `transferTo` you will need two nested `synchronized` statements to acquire two locks: one on `this` and the other on b. (The `synchronized` statements should be nested to ensure that the total capital held by the bank appears constant even in the presence of other transactions which are not related to balance transfers.) The order in which you acquire these locks is important if you want to avoid deadlock. (Why?) One way to avoid deadlock is to always acquire the lock on the account with the smallest account number first.

5. Run the `BankSimulator` class in your repository. You should find that the implementation is currently broken; the capital held by the bank at the start and the end of the day is different!

6. By aquiring and releasing appropriate locks inside the `transferTo` statement, fix the implementation so that the bank reliably finishes the day with the same amount of capital it started with.

An alternative to aquiring locks on individual bank accounts is to use a lock on the `BankSimulator` object. This would be simpler to implement and reason about, but would mean that only one thread could invoke the `transferTo` method at a time, even if all the bank accounts involved in a large set of transfers were different. Consequently there is a trade-off between complexity and performance when it comes to devising a locking strategy.

[2] You are not expected to make the bank transfer durable in the sense that the transaction will maintain the state of the transaction even in the presense of a power outage on your computer. Clearly a real banking system needs to ensure that the movement of any funds between bank accounts is recorded onto permanent storage before succeeding.

Ticklet 3

You have now completed all the necessary code to gain your third ticklet. Your repository should contain the following source files:

```
src/uk/ac/cam/your-crsid/fjava/tick3/BankSimulator.java
src/uk/ac/cam/your-crsid/fjava/tick3/MessageQueue.java
src/uk/ac/cam/your-crsid/fjava/tick3/ProducerConsumer.java
src/uk/ac/cam/your-crsid/fjava/tick3/QueueTest.java
src/uk/ac/cam/your-crsid/fjava/tick3/SafeMessageQueue.java
src/uk/ac/cam/your-crsid/fjava/tick3/UnsafeMessageQueue.java
```

When you are satisfied you have completed everything, you should commit all outstanding changes and push these to the Chime server. On the Chime server, check that the latest version of your files are in the repository, and once you are happy schedule your code for testing. You can resubmit as many times as you like and there is no penalty for re-submission. If, after waiting one hour, you have not received a final response you should notify `ticks1b-admin@cl.cam.ac.uk` of the problem. You should submit a version of your code which successfully passes the automated checks by the deadline, so don't leave it to the last minute!

	Thread S			Shared		Thread T		
	`a.transferTo(b,10)`	`tmp1`	`tmp2`	`a.bal`	`b.bal`	`b.transferTo(a,20)`	`tmp1`	`tmp2`
1				100	100			
2	`tmp1 = a.bal-10`							
3						`tmp1 = b.bal-20`		
4						`b.bal = tmp1`		
5	`a.bal = tmp1`							
6	`tmp2 = b.bal+10`							
7	`b.bal = tmp2`							
8						`tmp2 = a.bal+20`		
9						`a.bal = tmp2`		
10								

Table 2. An alternative execution trace of `transferTo` by two threads