PGCert IT: Programming for Industry

Lab 13: Concurrency & Multithreading

Exercise One: Simple Thread / Runnable

1. Write code that declares an anonymous `Runnable` object with the variable name `myRunnable`. The runnable should loop through all integers between 0 and 1 million, and print them out to the console.

2. Write code that creates a `Thread` which will run myRunnable, then starts that thread.

3. Write code that will request that your thread gracefully terminates, then wait for it to finish.

Exercise Two: Simple Concurrency

For this exercise, examine the following Java code:

```
public class ExerciseTwo {

	private int value = 0;

	private void start() throws InterruptedException {
		List<Thread> threads = new ArrayList<>();
		for (int i = 1; i <= 100; i++) {
			final int toAdd = i;
			Thread t = new Thread(new Runnable() {
				@Override
				public void run() {
					add(toAdd);
				}
			});
			t.start();
			threads.add(t);
		}

		for (Thread t : threads) {
			t.join();
		}

		System.out.println("value = " + value);
	}

	private void add(int i) {
		value = value + i;
	}

	public static void main(String[] args) throws InterruptedException {
		new ExerciseTwo().start();
	}

}
```

1. What would you expect the output of this program to be?

2. Is the output of the program guaranteed to be what you expect? Why / why not? If not, what is the name of the problem which may occur, and how could you change the program to mitigate this problem?

Exercise Three: A Banking Application

This exercise involves exploring concurrent access to shared objects in a simple banking application.

The following sources files have been provided in the materials inside the `ex04` package:

Class	Description
`BankAccount`	A simple class that represents a bank account. A `BankAccount` object maintains a balance and responds operations to perform deposits and withdrawals. Deposits are always successful, but withdrawals only succeed if the resulting balance is non-negative.
`Transaction`	`Transaction` objects represents banking transactions. A transaction can either be a deposit or a withdrawal, and is associated with an amount.
`TransactionGenerator`	This is a utility class that provides operations to generate and read transaction data. Method `writeDataFile()` creates a text file storing transactions, one per line. Method `readDataFile()` reads the text file and returns a List of Transaction objects.
`SerialBankingApp`	An application that applies a set of transactions to a single `BankAccount` object. The application uses `TransactionGenerator` to acquire a List of Transaction objects, and after applying each to the `BankAccount` object, prints the account's final balance.

Study the supplied files to become familiar with the classes.

Your task is to write a concurrent banking application in a new class named `ConcurrentBankingApp`. The application is similar to `SerialBankingApp`, but should use multiple threads to apply a set of transactions to a common `BankAccount` object.

The concurrent application should conform to the *Producer / Consumer* model, which is widely used in concurrent software systems. Essentially, producer threads generate data that are processed by consumer threads. In such systems, producers and consumers exchange data using a bounded buffer, which is a thread-safe entity that protects itself against concurrent access and which regulates the producers and consumers. When the buffer is full, producers are blocked from adding further data items to the buffer. Likewise, when empty, consumers are blocked and have to wait for items to be added to the buffer before they can continue.

The JDK includes a buffer interface knows as `BlockingQueue`. An implementation, `ArrayBlockingQueue`, also forms part of the JDK. An `ArrayBlockingQueue` has a finite size and supports operations for adding and removing data items to and from the queue. For example, to add an item to the queue, method `put()` can be used. If there is free space in the queue, `put()` adds the item and returns immediately, allowing the producer to continue to execute. In other cases, if the queue is full, `put()` blocks the caller until space becomes available. Similarly, to remove an item from a queue, the caller can be blocked when the queue is empty – using method `take()`. An additional method, `poll()`, returns null immediately if the queue is empty, allowing the consumer to attempt to retrieve a data item, but to continue to execute rather than block if there are no items in the queue. See the `java.util.concurrent` Javadoc pages for further details.

Your concurrent program should comprise one producer and two consumer threads. The producer is to retrieve a List of `Transaction` objects using the `TransactionGenerator` class. It should then add these to a `BlockingQueue` object, which should be initialized with a capacity of 10 `Transaction` items. During its operation, the producer will likely block because the buffer becomes full. To make space in the queue, the two consumers should retrieve `Transactions` from the queue and apply them to a common `BankAccount` object.
Within the Producer / Consumer program architecture, the producer and consumers should share a common `ArrayBlockingQueue` object – as this regulates execution of the producer and consumers. The two consumers should also share a reference to a common `BankAccount` object. The program should ensure that all transactions are executed exactly once in total.

The basic algorithm for ConcurrentBankingApp is as follows:

1. Create the three threads.
2. Start each thread.
3. Wait for the producer to finish putting `Transaction` items into the queue and terminate.
4. Once the producer has finished, send an interrupt request to the two consumers. The consumers should respond by terminating as soon as the queue becomes empty after receiving the interrupt request. Note that if the consumers terminate because the queue is empty before receiving the interrupt, there may be more data to arrive in the queue that wouldn't be processed.
5. Wait for the two consumers to terminate.
6. Print out the final balance of the shared `BankAccount` object.

Once you have completed the program, run it and compare the final balance with that of `SerialBankingApp` when run with the same set of transactions. What do you notice? What simple change do you need to make to class `BankAccount` to ensure correctness of the concurrent program?

Hints

- The most convenient way of implementing the single producer is through an anonymous inner class of type Runnable.

```
final BlockingQueue<Transaction> queue = new
ArrayBlockingQueue<>(10);

Thread producer = new Thread(new Runnable() {
      @Override
      public void run() {
         // TODO Stuff & things
      }
});
```

The producer code will need to access the shared `BlockingQueue` object – any code in an anonymous inner class can access variables that are declared final in the enclosing method.

- For the two consumers, a new class `Consumer` should be introduced. This class can be instantiated twice, once for each consumer. `Consumer` should implement `Runnable` so that a `Consumer` instance can be passed to a `Thread` object for execution. `Consumer`'s instance variables should include references to the shared `BlockingQueue` and `BankAccount` objects.
- From class `Thread`, methods `interrupt()` and `join()` are sufficient for this exercise.
- You should handle `InterruptedException` appropriately. Note that this exception can be thrown by methods that block the calling thread, e.g. `BlockingQueue`'s `put()` and `take()` methods and Thread's `join()` method. For `Consumer`, catching an `InterruptedException` from a `put()` or `take()` call on the `BlockingQueue` indicates that the consumer thread has been interrupted (e.g., by `ConcurrentBankingApp` having detected that that the producer has finished putting Transactions into the queue).

Exercise Four: Prime Factorization of N

This exercise involves developing a multithreaded command-line program to perform what can be a lengthy computation. The computation in this case is the prime factorization of number N. Fortunately, an algorithm for performing the computation has already been developed, and an implementation in Java is available! Familiarize yourself with the following webpage: http://introcs.cs.princeton.edu/java/13flow/Factors.java.html.

For this exercise, there is no starting point provided – you should create everything yourself, including an appropriate package for your java files.

This exercise is broken down into 3 steps.

Step one: `PrimeFactorsTask`

Implement a new class, PrimeFactorsTask, that implements the Runnable interface. This class will implement the prime factorization algorithm and should provide the following public methods.

- `PrimeFactorsTask(long n)`
- `void run()`
- `long n()`
- `List<Long> getPrimeFactors() throws IllegalStateException`
- `TaskState getState()`

In addition, `PrimeFactorsTask` should define a public enumerated type named `TaskState` with three values: `Initialized`, `Completed` and `Aborted`. The `Aborted` value won't be required until step 3, but it can be added now.

The constructor should create a `PrimeFactorsTask` object with a given value for N (the value for which prime factors are to be computed). Initializing the `PrimeFactorsTask` object also involves instantiating a `List<Long>` instance variable, which will later be used to store computed prime numbers, and an instance variable of type `TaskState`, assigned the value Initialized.

Method `run()` should implement the algorithm for computing prime factors, storing the computed primes in the `PrimeFactorTask`'s `List` instance variable. When the algorithm has completed, `run()` should set the `PrimeFactorsTask` object's `TaskState` instance variable to Completed.

`n()` simply returns the value of N for which the `PrimeFactorsTask` object has been created. The `getPrimeFactors()` method is intended to be called only after the `run()` method has completed. In cases where the algorithm has completed, this method should return the List of computed prime factors. In other cases, it should throw the standard Java run-time exception `IllegalStateException`.

Finally, method `getState()` returns the current state of the `PrimeFactorsTask` instance. Having implemented `PrimeFactorsTask`, you have a class whose objects are intended to be executed by `Thread` instances. When a `Thread` is created and passed a `PrimeFactorsTask` object, starting the Thread object will cause the new thread to execute the `PrimeFactorsTask`'s `run()` method. The `PrimeFactorsTask` object, like any object, has state that it modifies over time. Once the thread has finished executing, the associated `PrimeFactorsTask` object can be inspected for the computed prime factors.

Step two: A command-line program

Write a simple command-line program that allows a user to supply a value for N and which executes a `PrimeFactorsTask` object in a separate thread. Once the `PrimeFactorsTask` object's `run()` method has completed (which causes the associated `Thread` object to terminate), request the computed prime factors from the `PrimeFactorsTask` object and display them to the user.

Step three: Allowing the user to cancel

Enhance your program so that the user can cancel the `Thread` used to execute the prime factor computation. For large values of N, e.g. 9201111169755555649, the algorithm takes a few minutes to execute, and so the ability to abort the computation may be useful.

Hints

- For step 2, you simply need to create a `PrimeFactorsTask` object and a `Thread` instance. Upon creating the Thread, pass in the `PrimeFactorsTask` object (this is valid because `PrimeFactorsTask` implements the `Runnable` interface type expected by `Thread`).
- To wait for the `Thread` object (used to run the `PrimeFactorsTask`) to terminate – either because the associated `PrimeFactorsTask` object's `run()` method completed normally or was aborted, use `Thread`'s `join()` method.
- For step 3, the basic approach is to introduce an extra thread to handle user input indicating that the computation should be aborted. You would then have 3 threads in your program:
    1. The usual main thread that starts the other two threads and which waits for the computation thread to complete.
    2. The computation thread that is used to run the `PrimeFactorsTask`.
    3. The abort thread that is used to block on user input. When the user types a key, indicating that they want to abort the computation, this thread should interrupt the computation thread via an `interrupt()` call.
- Taking the above approach for step 3, `PrimeFactorsTask` needs to participate in the abortion process. Whenever the abort thread is interrupted, the `PrimeFactorsTask` needs to become aware that it should cease processing. A common pattern in such situations is to have a `Runnable` object's `run()` method periodically check its `Thread`'s interrupted status, and for the `run()` method to finish upon detecting the interrupt. For `PrimeFactorsTask`, you can add such a check inside the for loop of the prime factors algorithm. `Thread`'s class (as opposed to instance) method `interrupted()` returns true if the `Thread` in which the `PrimeFactorsTask` object is running has been interrupted.