Parallel Computing

Homework #3

Theoretical Part

Concurrency

Concurrency simply means that multiple computations are happening at the same time. For example, while working on a word processor we are also listening to music. In the background, we are downloading a movie and at the same time printing documents.
Software that can do such things is known as concurrent software.

The java programming language is designed to support concurrent programming with the help of class libraries and it also includes high-level concurrency APIs to make the life of a programmer easy.

Processes and Threads

In concurrent programming, there are two basic units of execution: processes and threads. However, in the Java programming language, concurrent programming is mostly concerned with threads and that's what we will focus on.

In the Java programming language to create *threads,* we need to use the Thread class. All of the threads that are executing are an instance of the Thread class. There are two basic ways of using the Thread

1. Directly control the creation and management of a thread i.e create a Thread each time the application needs to execute a task (asynchronous)
2. Abstract thread management i.e pass the application's tasks to an executor and the executor would create a thread and execute the task.

Threads in Java

Defining and starting a thread in java is really simple there is only one thing that a programmer must keep in mind and that is the application that is creating an instance of the Thread must also provide the code that would be executed on the thread. There are two ways to accomplish this

1. Using Runnable Object

   This simply means that the programmer implements the ***`Runnable`*** interface which contains a single method ***`run`*** which contains the code to be executed in the thread. The programmer just needs to pass the object od this class to the Thread constructor and voila. Example

```
public class ExampleRunnable implements Runnable {

    public void run() {
        System.out.println("Hello World! I'm a thread!");
    }

    public static void main(String args[]) {
        ExampleRunnable obj = new ExampleRunnable();
        Thread th = new Thread(obj);
        th.start();
    }
}
```

2. Subclass Thread

   The Thread class itself implements the Runnable.

```
public class ExampleThread implements Runnable {
    public void run() {
        System.out.println("Hello World! I'm a thread!");
    }
    public static void main(String args[]) {
        ExampleThread th = new ExampleThread();
        th.start();
    }
}
```

As we can see in both od the examples that we need to invoke the ***start()*** method to start the new thread. Similarly, the Thread class defines a number of methods for thread management.
One such method is `sleep()` which as the name suggests causes the current thread to suspend execution i.e go to sleep for a specific period of time. The ***sleep()*** method is also really useful in the cases when we have multiple threads and some thread require more time and processing to complete a task. In such cases, we can invoke the sleep method on the threads which doesn't require more processing time thus causing the other threads to get more time and complete the task faster.
Note: ***sleep()*** method throws an exception ***InterruptedException*** when another thread interrupts the current thread while it is sleeping. When working with ***sleep()*** method one needs to take care of this exception as well.

```
# we can ignore `throws InterruptedException` as we only have one
#thread working but it is considered good practice to handle
it.

public class ExampleSleep throws InterruptedException{

    String personal_info[] = {
        "My name is Yash Sharma",
        "I am a masters student at ETU LETI",
        "I am from New Delhi, India",
        "Nice to meet you."
    };

    public static void main(String args[]) {
        for(int i = 0; i < personal_info.length; i++) {
            System.out.println(personal_info[i]);
            Thread.sleep(2000);
        }
    }
}
```

Another important method in Thread class is `Thread.interrupt` which interrupts the process or thread. In layman's terms, an interrupt is a way to tell the thread that it must stop whatever it is doing and do something else for example if a thread is sleeping we can interrupt the sleep and tell the thread to wake up and start executing a task.
A thread sends an interrupt by invoking interrupt on the Thread object for the thread to be interrupted but for this to work, the interrupted thread must must support its own interruption i.e the interrupted thread must know what to do when it is interrupted and this is the task of the programmer to define exactly how to handle interrupts.
Let's update the previous example to handle what would thread do if some other thread interrupts it.

```
public class ExampleSleep{

    String personal_info[] = {
        “My name is Yash Sharma”,
        “I am a masters student at ETU LETI”,
        “I am from New Delhi, India”,
        “Nice to meet you.”
    };

    public static void main(String args[]) {
        for(int i = 0; i < personal_info.length; i++) {
            System.out.println(personal_info[i]);
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                return;
            }
        }
    }
}
```

What this code would do is print a statement from the personal_info string array and would try to sleep for 2 seconds. If during that 2 seconds of sleeping some other thread would interrupt this thread, it would enter the catch clause and would end the execution.

Next, on the list of Thread methods is `join()` method which allows one thread to wait for the completion of another thread. The `join()` method puts the current thread on wait until the thread on which it's called is dead.

```
class MyRunnable implements Runnable{

    public void run() {
        System.out.println("Hello World! I'm a thread!");

        try {
            Thread.sleep(4000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Bye Bye);
    }

}

public class ExampleThreadJoin {

    public static void main(String[] args) {

        Thread t1 = new Thread(new MyRunnable(), "t1");
        Thread t2 = new Thread(new MyRunnable(), "t2");

        t1.start();

        try {
            t1.join(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        t2.start();

        try {
            t1.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("All threads are complete!");
    }

}
```

Synchronization

Threads communicate by sharing access to common objects and fields. This communication is efficient but error-prone such as

1. ***Thread Interference*** occurs when two operations, running on multiple threads but accessing the same data, interleave. This means that the two operations consist of multiple steps, and the sequences of steps overlap.
2. ***Memory Consistency Errors*** occur when different threads have inconsistent views of the shared memory but luckily programmer does not need a detailed understanding of these causes.
   The key to avoiding memory consistency errors in understanding the happens-before relationship. This relationship is simply a guarantee that memory writes by one specific statement is visible to another specific statement.

We talked about the problem or error that might occur, now let’s talk about the ways to synchronize the threads.
Java Programming language provides two ways:

1. Synchronized Methods
   This is the most simple way to synchronize, the programmer just needs to add the `synchronized` keyword to the declaration.

For example:

```
public synchronized void increment() {
    i++;
}

public synchronized void decrement() {
    i--;
}
```

2. Synchronized Statements
   Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

Liveness

A concurrent application's ability to execute in a timely manner is known as its *liveness*. The liveness problems which might occur while working with threads are:

1. ***Deadlocks***
   Assume an example where two trains are coming towards each other on the same track and there is only on the track, once they are in front of each other none of the trains can move. The same situation can occur in threads,

where two or more threads are trying to access the data at the same time but are blocked forever, waiting for each other to finish.

In simpler terms, a set of threads is deadlocked when each thread is waiting for a resource to be freed which is controlled by another process thus causing all the threads to be blocked till eternity.

2. ***Starvation***
   Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads.

High-Level Concurrency Objects

Till now all the methods and functions discussed were part of the low-level APIs in Java which is inbuilt into the language since the beginning but that doesn't mean that's all no the Java Programming language also have high-level concurrency features which were introduced in version 5.0 to make the life of programmers easier.

- **Lock Objects**
  Lock objects work very much like the implicit locks used by synchronized code. As with implicit locks, only one thread can own a Lock object at a time. Lock objects also support a wait/notify mechanism. But the biggest advantage of Lock object over implicit locks is their ability to back out of an attempt to acquire a lock. The tryLock method backs out if the lock is not available immediately or before a timeout expires.
  Lock Objects can help solve the Deadlock problem we discussed above.

- **Executors** define a high-level API for launching and managing threads. This is the second part of the strategy of using Thread in Java. Using Executors we can separate thread management and creation from the rest of the application completely. Now, we just need to pass the application task to the executor and forget about it. The executor would handle it.

One important concept in the Executors is ***Thread Pools*** which is basically a pool of worker threads that are ready to execute some task. This pool of worker threads is separate from the Runnable and Callable. The advantage of using this is that it minimizes the overhead due to thread creation. Thread objects use a significant amount of memory, and in a large-scale application, allocating and deallocating many thread objects creates a significant memory management overhead.

- **Concurrent Collections** make it easier to manage large collections of data, and can greatly reduce the need for synchronization.
    - `BlockingQueue` defines a first-in-first-out data structure that blocks or times out when you attempt to add to a full queue or retrieve from an empty queue.
    - `ConcurrentMap` is a subinterface of `java.util.Map` that defines useful atomic operations. These operations remove or replace a key-value pair only if the key is present, or add a key-value pair only if the key is absent. Making these operations atomic helps avoid synchronization. The standard general-purpose implementation of ConcurrentMap is ConcurrentHashMap, which is a concurrent analog of HashMap.
    - `ConcurrentNavigableMap` is a subinterface of ConcurrentMap that supports approximate matches. The standard general-purpose implementation of ConcurrentNavigableMap is ConcurrentSkipListMap, which is a concurrent analog of TreeMap.

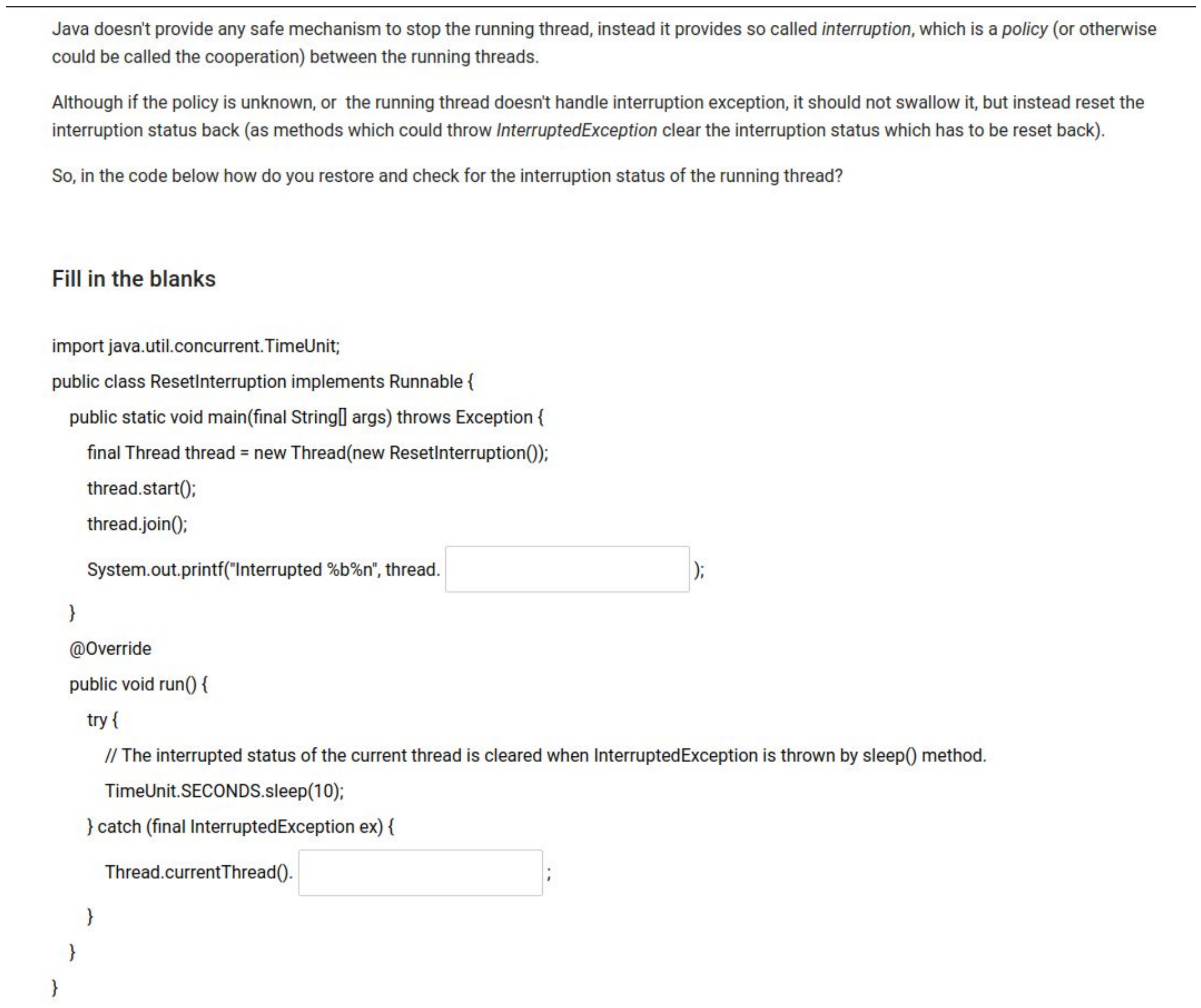
- **Atomic Variables**
  Defines classes that support atomic operations on single variables. All classes have `get` and `set` methods that work like reads and write on volatile variables.

Practical Part

```
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

public class ThreadSafe6Quiz {
    public static class ThreadSafe6 {
        private final Map< String, String > data = new
ConcurrentHashMap<>();
        public void putIfAbsent(final String key, final String
value) {
            data.putIfAbsent(key, value);
        }
    }
}
```

Problem Faces: In the beginning tried using ConcurrentMap but it gave an error after reading the documents found out that it is an interface not the implementation. ConcurrentHashMap is the implementation of it. Thus, reading doc helped solve the problem.


Java doesn't provide any safe mechanism to stop the running thread, instead it provides so called *interruption*, which is a *policy* (or otherwise could be called the cooperation) between the running threads.

Although if the policy is unknown, or the running thread doesn't handle interruption exception, it should not swallow it, but instead reset the interruption status back (as methods which could throw *InterruptedException* clear the interruption status which has to be reset back).

So, in the code below how do you restore and check for the interruption status of the running thread?

Fill in the blanks

```
import java.util.concurrent.TimeUnit;
public class ResetInterruption implements Runnable {
  public static void main(final String[] args) throws Exception {
    final Thread thread = new Thread(new ResetInterruption());
    thread.start();
    thread.join();
    System.out.printf("Interrupted %b%n", thread.                    );
  }
  @Override
  public void run() {
    try {
      // The interrupted status of the current thread is cleared when InterruptedException is thrown by sleep() method.
      TimeUnit.SECONDS.sleep(10);
    } catch (final InterruptedException ex) {
      Thread.currentThread().                    ;
    }
  }
}
```

Did not understand the question properly. I tried multiple ways but doesn't work.

I also faced a bit of problem in the tasks which were not covered in the theoretically part, for example, using other classes in java.utils.concurrent. I had to open documentation and read about all the available classes and their methods to understand how each works.
I also faced a problem with the comparison questions where the java was being compared with the go language (Research: What it is the analog in Java of the Go Once).