CIS 36B – 19th Class / Lab Assignment – **10 Points-**
Student Name______________________________ **Student ID**________________________ **Point Total**

Section 1: Define

Pg. Chapter 12, pg. 435 - Java Programming *A comprehensive Introduction*

getName()

Returns this thread's name.

getPriority

Returns this thread's priority.

synchronized-

The Java programming language provides two basic synchronization idioms: *synchronized methods* and*synchronized statements*. The more complex of the two, synchronized statements, are described in the next section. This section is about synchronized methods.

To make a method synchronized, simply add the `synchronized` keyword to its declaration:

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

If `count` is an instance of `SynchronizedCounter`, then making these methods synchronized has two effects:

- First, it is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.

- Second, when a synchronized method exits, it automatically establishes a happens-before relationship with *any subsequent invocation* of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

Note that constructors cannot be synchronized — using the `synchronized` keyword with a constructor is a syntax error. Synchronizing constructors doesn't make sense, because only the thread that creates an object should have access to it while it is being constructed.

Warning: When constructing an object that will be shared between threads, be very careful that a reference to the object does not "leak" prematurely. For example, suppose you want to maintain a `List` called `instances` containing every instance of class. You might be tempted to add the following line to your constructor:

```
instances.add(this);
```

But then other threads can use `instances` to access the object before construction of the object is complete.

Synchronized methods enable a simple strategy for preventing thread interference and memory consistency errors: if an object is visible to more than one thread, all reads or writes to that object's variables are done through `synchronized` methods. (An important exception: `final` fields, which cannot be modified after the object is constructed, can be safely read through non-synchronized methods, once the object is constructed) This strategy is effective, but can present problems with liveness, as we'll see later in this lesson.

lock()-

Intrinsic Locks and Synchronization

Synchronization is built around an internal entity known as the *intrinsic lock* or *monitor lock*. (The API specification often refers to this entity simply as a "monitor.") Intrinsic locks play a role in both aspects of synchronization: enforcing exclusive access to an object's state and establishing happens-before relationships that are essential to visibility.

Every object has an intrinsic lock associated with it. By convention, a thread that needs exclusive and consistent access to an object's fields has to *acquire* the object's intrinsic lock before accessing them, and then *release* the intrinsic lock when it's done with them. A thread is said to *own* the intrinsic lock between the time it has acquired the lock and released the lock. As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock.

When a thread releases an intrinsic lock, a happens-before relationship is established between that action and any subsequent acquisition of the same lock.

Locks In Synchronized Methods

CIS 36B – 19th Class / Lab Assignment – **10 Points-**

Student Name________________________________ **Student ID**________________________________ **Point Total**

When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns. The lock release occurs even if the return was caused by an uncaught exception.

You might wonder what happens when a static synchronized method is invoked, since a static method is associated with a class, not an object. In this case, the thread acquires the intrinsic lock for the `Class` object associated with the class. Thus access to class's static fields is controlled by a lock that's distinct from the lock for any instance of the class.

Task 1:

USE OBJECT ORIENTATED PROGRAM DESIGN TO SOLVE PROBLEM

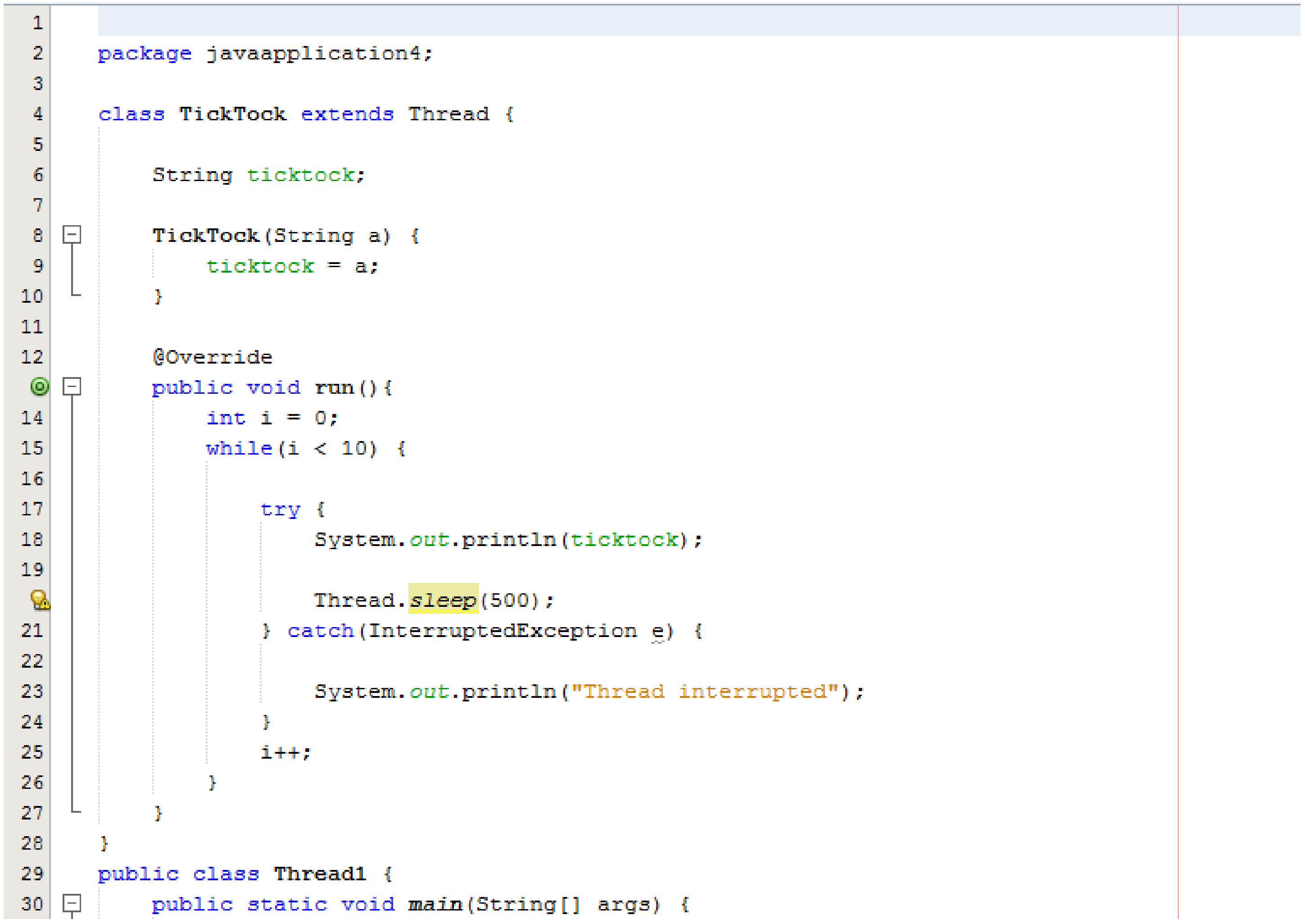
Create a class called TickTock so that it keeps time in seconds.

Have each tick take one half second, and each took take one half second. Each Tick-Tock will take one second.

CIS 36B – 19th Class / Lab Assignment – **10 Points-**

Student Name____________________________ **Student ID**____________________________ **Point Total**

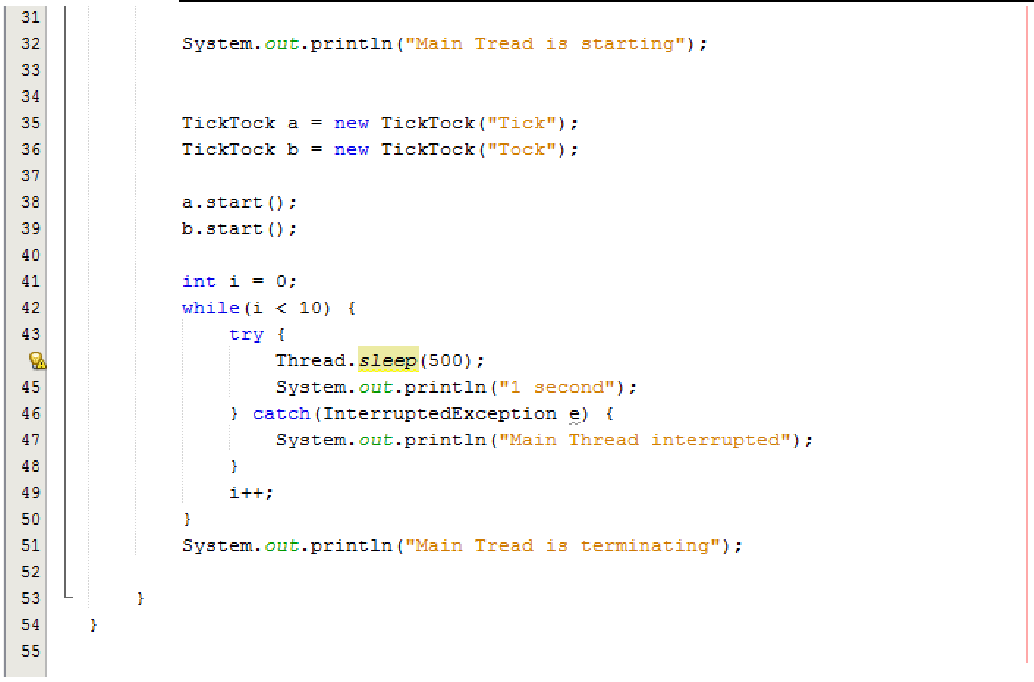
•



```java

package javaapplication4;

class TickTock extends Thread {

    String ticktock;

    TickTock(String a) {
        ticktock = a;
    }

    @Override
    public void run(){
        int i = 0;
        while(i < 10) {

            try {
                System.out.println(ticktock);

                Thread.sleep(500);
            } catch(InterruptedException e) {

                System.out.println("Thread interrupted");
            }
            i++;
        }
    }
}
public class Thread1 {
    public static void main(String[] args) {
```

CIS 36B – 19th Class / Lab Assignment – **10 Points-**

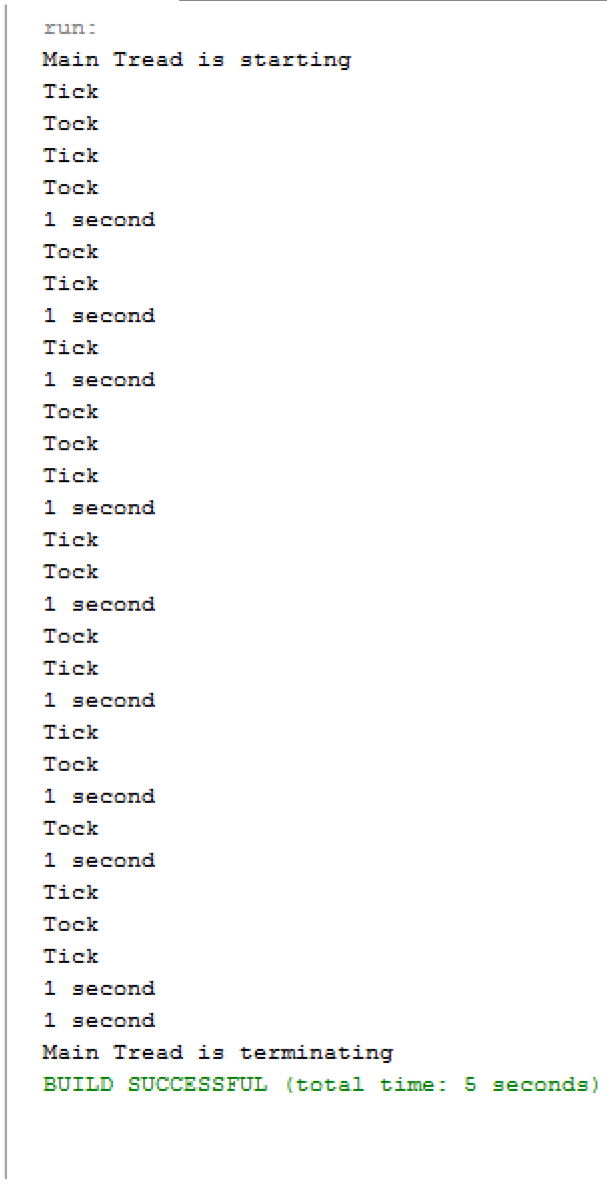
Student Name____ **Student ID**______________________________________ **Point Total**



```
        System.out.println("Main Tread is starting");


        TickTock a = new TickTock("Tick");
        TickTock b = new TickTock("Tock");

        a.start();
        b.start();

        int i = 0;
        while(i < 10) {
            try {
                Thread.sleep(500);
                System.out.println("1 second");
            } catch(InterruptedException e) {
                System.out.println("Main Thread interrupted");
            }
            i++;
        }
        System.out.println("Main Tread is terminating");

    }
}
```

CIS 36B – 19th Class / Lab Assignment – **10 Points-**

Student Name____________________________________**Student ID**____________________________________**Point Total**



```
run:
Main Tread is starting
Tick
Tock
Tick
Tock
1 second
Tock
Tick
1 second
Tick
1 second
Tock
Tock
Tick
1 second
Tick
Tock
1 second
Tock
Tick
1 second
Tick
Tock
1 second
Tock
1 second
Tick
Tock
Tick
1 second
1 second
Main Tread is terminating
BUILD SUCCESSFUL (total time: 5 seconds)
```