Session Overview

- Lab Solution – CountdownTimer
- Concurrent Access to Objects and Variables
- Member Variable Modifier: `volatile`
- Using `synchronized`
- Locking and Unlocking
- Using `synchronized` Instead of `volatile`
- Holding a Lock Between Method Calls
- Threads and Collections
- Threads and Swing
- Lab – RayDraw

Concurrent Access to Objects and Variables

- Things get tricky when more than one thread can interact with an object.
- Especially tricky is when two threads might be concurrently inside the methods of an object.
- Care must be taken to be sure that one thread sees the other's changes.
- Care must be taken to be sure that the threads don't collide and leave the object in an invalid (corrupt) state.
- Care must be taken to be sure that a thread isn't able to view an object while in an inconsistent state (for example while another thread is in the process of altering the object). [Sometimes this is a called a "dirty read".]

Member Variable Modifier: `volatile`

- A member variable can be declared as `volatile` to allow for multiple threads to see changes to its value when full synchronization isn't used (more on synchronization next).

- The `volatile` modifier is <u>*required*</u> on a variable if all of the following are true:
    - two or more threads interact with the variable, ***and***
    - one of the threads might change the value of the variable, ***and***
    - <u>not all</u> of the threads use synchronization to access/modify the variable.

- Without `volatile`, one thread might <u>not</u> see the changes that another thread has made!

- In many cases, `synchronized` should be used instead of `volatile`.

Using `synchronized`

- See Chapter 7 of "*Java Thread Programming*".

- The `synchronized` keyword can be used in three places:
  - non-`static` methods
  - blocks
  - `static` methods

- In each case, the `synchronized` keyword implies that exclusive access to a lock must be obtained before proceeding.

- There are two kinds of locks in Java:
  - Object-level
    - Each instance of a class has its own object-level lock.

  - Class-level
    - Every class has just one class-level lock.

- `static` methods require that the calling thread gain exclusive access to the *class-*level lock before entering the method:

```
public class ClassA //...
    //...
    public static synchronized void setValue(int value)
    //...
}
```

  for example, the thread executing this code:

```
ClassA.setValue(5);
```

  must get exclusive access to the single class-level lock before the Java VM lets the thread into `setValue()`.

- non-`static` methods require that the calling thread gain exclusive access to the *object*-level lock for the instance before entering the method.

```
public class ClassB //...
    //...
    public synchronized void setValue(int value)
    //...
}
```

for example, the thread executing this code:

```
ClassB b1 = //...
b1.setValue(5);
```

must get exclusive access to the object-level lock for `b1` before the Java VM lets the thread into `setValue()`. Each instance has its own object-level lock.

- `synchronized` blocks require that the calling thread gain exclusive access to the *object*-level lock for the object referenced before entering the block. For example, the thread executing this code:

```
ClassC ref = //...

synchronized ( ref ) {
    //...
}
```

must get exclusive access to the object-level lock for the object that `ref` is currently pointing to before the Java VM lets the thread into the `synchronized` block.

Locking and Unlocking

- In Java, threads don't directly acquire or release a lock, locking instead occurs automatically as the thread enters or leaves a `synchronized` block or method.

- If a thread can't immediately get exclusive access to the required lock, the thread *blocks* (potentially forever) until it finally acquires the lock.

- If a thread returns from a method either normally (`return`) or by throwing an exception, the lock is automatically released.

- As we'll see with the wait-notify mechanism later, a thread will temporarily release its lock while "waiting for notification" (however, a thread does not release its lock while inside `Thread.sleep()`!).

- If a thread calls another method while holding a lock, it does not release the lock but continues to hold it.

- If a thread is holding a lock and calls another method that requires that same lock, the thread is immediately allowed into the method as it is already has exclusive access to the required lock.

- Locking Example:

```
public class ClassA //...
    public synchronized void methodA() {
        methodB();
        methodC();
        //...
    }

    public synchronized void methodB() {
        //...
    }

    // not synchronized
    public void methodC() {
        //...
    }
}
```

A thread that calls `methodA()` will wait until it gets exclusive access to the object-level lock. It will then go into `methodA()`, call `methodB()` and `methodC()` all while continuing to hold the object-level lock. The lock is finally released when the thread returns from `methodA()`.

- Questions:
  - While a thread is inside `methodA()`, can another thread get inside `methodB()` on the same instance?
  - While a thread is inside `methodA()`, can another thread get inside `methodC()` on the same instance?

Using `synchronized` Instead of `volatile`

- Synchronizing methods to access and change a variable can eliminate the need for **`volatile`** on that member variable:

```
public class ClassD extends Object {
    private double value;  // doesn't need to be volatile

    public synchronized double getValue() {
        return value;
    }

    public synchronized void setValue(double x) {
        value = x;
    }
}
```

- When more than one value is to be set at one time, **`volatile`** is not even an option, **`synchronized`** must be used:

```
public class ClassE extends Object {
    private double x;
    private double y;

    public synchronized void setLocation(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public synchronized Point2D getLocation() {
        //... return the current x, y pair ...
    }
}
```

Imagine the case were a thread executes the following code:

```
ClassE e = //...
e.setLocation(10, 10);
```

and another thread executes:

```
e.setLocation(20, 20);
```

If **`setLocation()`** was not **`synchronized`**, with another thread we might occasionally observe an x, y pair showing (10, 20) or some other corrupt combination.

Holding a Lock Between Method Calls

- Sometimes we need to sequentially invoke two synchronized methods on a object and want to be sure that another thread can't sneak in between those calls.
- Example:

```
public class ClassB extends Object {
    private double value;

    public synchronized double getValue() {
        return value;
    }

    public synchronized void setValue(double x) {
        value = x;
    }
}
```

and we have a thread executing this code:

```
ClassB b = //..

if ( b.getValue() < 10.0 ) {
    b.setValue(50.0);
}
```

Then there is a small chance that after we return from `getValue()` but before we call `setValue()`, another thread will sneak in and alter the value!

- To eliminate this race condition, we need to use a `synchronized` block so that the calling thread holds the lock the whole time that it is inside the block—not just while inside the methods:

```
ClassB b = //..

synchronized ( b ) {
    if ( b.getValue() < 10.0 ) {
        b.setValue(50.0);
    }
}
```

Threads and Collections

- See material starting on page 155 of "Java Thread Programming".
- The Collections API is not inherently thread-safe.
- For speed, the default implementations do not use **`synchronized`** methods to control concurrent access. For example:

```
List list = new ArrayList(); // not multithread-safe
```

  If multiple threads interact with **`list`**, there are dangerous race conditions. If only one thread interacts, then we're perfectly safe and benefit by not having the overhead of invoking **`synchronized`** methods.

- If a collection is going to be accessed/modified by multiple threads, it needs to be wrapped in synchronization:

```
public class Collections //...
    public static Collection synchronizedCollection(Collection raw) //...
    public static List synchronizedList(List raw) //...
    public static Map synchronizedMap(Map raw) //...
    public static Set synchronizedSet(Set raw) //...
    public static SortedMap synchronizedSortedMap(SortedMap raw) //...
    public static SortedSet synchronizedSortedSet(SortedSet raw) //...
    //...
}
```

  For example:

```
List list = Collections.synchronizedList(new ArrayList()); // safe
```

  allows **`list`** to be safely accessed by multiple threads. A new instance of a class that implements the **`List`** interface with all **`synchronized`** methods is returned from this method. Internally, the synchronized method turns and calls the unsynchronized method on the backing **`List`**—in this case the underlying **`ArrayList`**. You should not allow direct access to the underlying **`ArrayList`**.

- Example: **`ListHelper.java`**

ListHelper.java

```java
import java.util.*;

public class ListHelper extends Object {
    private List list;

    public ListHelper() {
        // Don't keep any reference to the raw, unsync'd list
        list = Collections.synchronizedList(new ArrayList());
    }

    // allow general access to the list
    public List getList() {
        return list;
    }

    public void add(Object obj) {
        list.add(obj);
    }

    public boolean addIfNotDuplicate(Object obj) {
        synchronized ( list ) {
            if ( list.contains(obj) == false ) {
                list.add(obj);
                return true;
            }

            return false;
        }
    }

    public void sort() {
        synchronized ( list ) {
            Collections.sort(list);
        }
    }

    public void printContents() {
        synchronized ( list ) {
            Iterator iter = list.iterator();
            while ( iter.hasNext() ) {
                System.out.println(iter.next());
            }
        }
    }

    public static void main(String[] args) {
        ListHelper lh = new ListHelper();
        lh.add("Pear");
        lh.add("Apple");
        lh.add("Banana");

        lh.addIfNotDuplicate("Pear");
        lh.printContents();
        System.out.println("-----------");

        lh.sort();
        lh.printContents();
    }
}
```

Output

```
Pear
Apple
Banana
-----------
Apple
Banana
Pear
```

Threads and Swing

- See Chapter 9 – "Threads and Swing" of "Java Thread Programming".

- Swing was designed for speed and Swing components do not safely support access by any thread other than the "event handling" thread after they have been rendered.
    - This helped to keep Swing simpler and speeds runtime execution.

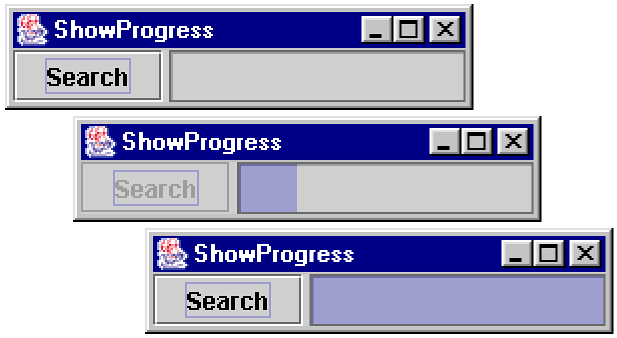
- Swing does provide a mechanism for sending work units to the event thread for safe component modification.
- These two `static` methods on `SwingUtilities` allow for work units to be handed off to the event thread:

```
public static void invokeAndWait(Runnable synchronousTask)
        throws InterruptedExcption, InvocationTargetException

public static void invokeLater(Runnable asynchronousTask)
```

- the `SwingUtilities.invokeLater()` method is generally preferred

- Example: `ShowProgress.java`



ShowProgress.java

```
// For more on using threads and Swing, see Chapter 9 -
// "Threads and Swing" of "Java Thread Programming"
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ShowProgress extends JPanel {
    private JButton searchB;
    private JProgressBar progressBar;
    private Runnable runWorkRunnable;
    private Runnable enableSearchButtonTask;
    private ProgressTask progressTask;

    public ShowProgress() {
        // create this just once and use over and over
        runWorkRunnable = new Runnable() {
                public void run() {
                    runWork();
                }
            };

        // Use the same runnable over and over to
        // save object construction costs. This instance
        // of Runnable is used at the end of runWork().
        enableSearchButtonTask = new Runnable() {
                public void run() {
                    searchB.setEnabled(true);
                }
            };

        searchB = new JButton("Search");
        searchB.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    searchB.setEnabled(false);
                    launchHelper();
                }
            });

        progressBar = new JProgressBar(0, 100);
        progressTask = new ProgressTask(progressBar);

        setLayout(new BorderLayout(3, 3));
        add(searchB, BorderLayout.WEST);
        add(progressBar, BorderLayout.CENTER);
    }

```

```java
    private void launchHelper() {
        Thread t = new Thread(runWorkRunnable, "ShowProgress helper");
        t.start();
    }

    private void runWork() {
        for ( int i = 0; i < 20; i++ ) {
            // Update to the latest value and (re)submit the task
            progressTask.setProgressValue(( i + 1 ) * 5);

            // Safely update the progress
            SwingUtilities.invokeLater(progressTask);

            try {
                Thread.sleep(200);
            } catch ( InterruptedException x ) {
                return;
            }
        }

        // Safely re-enable the Search button and change
        // the cursor back to normal using the event
        // handling thread--not this helper thread!
        SwingUtilities.invokeLater(enableSearchButtonTask);
    }

    private static class ProgressTask extends Object implements Runnable {
        private JProgressBar bar;
        private int progressValue;

        public ProgressTask(JProgressBar bar) {
            this.bar = bar;
            this.progressValue = 0;
        }

        // callable by any thread
        public synchronized void setProgressValue(int newVal) {
            this.progressValue = newVal;
        }

        // callable by any thread
        private synchronized int getProgressValue() {
            return progressValue;
        }

        // callable only by the event handling thread
        public void run() {
            bar.setValue(getProgressValue());
        }
    }

    public static void main(String[] args) {
        ShowProgress sp = new ShowProgress();

        JFrame f = new JFrame("ShowProgress");
        f.setContentPane(sp);
        f.pack();
        f.setVisible(true);
        f.addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
                    System.exit(0);
                }
            });
    }
}
```