Java Threads - 3

Thread Interference

- Interference happens when two operations, running in different threads, but acting on the same data, *interleave*.
- This means that the two operations consist of multiple steps, and the sequences of steps overlap.

```
class Counter { private int c = 0;
        public void increment() {
         c++; }
        public void decrement() {
        c--; }
        public int value() {
        return c; } }
```

Suppose Thread A invokes increment at about the same time Thread B invokes decrement. If the initial value of c is 0, their interleaved actions might follow this sequence:

1. Thread A: Retrieve c.
2. Thread B: Retrieve c.
3. Thread A: Increment retrieved value; result is 1.
4. Thread B: Decrement retrieved value; result is -1.
5. Thread A: Store result in c; c is now 1.
6. Thread B: Store result in c; c is now -1.

Thread A's result is lost, overwritten by Thread B. Thread interference bugs can be difficult to detect and fix.

1. Retrieve the current value of c.
2. Increment the retrieved value by 1.
3. Store the incremented value back in c.

Race Condition Example

```
class incrementingThread extends
Thread{
private Counter counter1;
public incrementingThread(Counter
counter1){this.counter1 = counter1;}
public void run(){
this.counter1.increment();
}}
```

```
class decrementingThread extends
Thread{
private Counter counter1;
public decrementingThread(Counter
counter1){this.counter1 = counter1;
}
public void run(){
this.counter1.decrement();
}}
```

```
class Counter {
   private int c = 0;
   public void increment() {
   System.out.println("Entering Increment Method");
   for (int i=0; i<=100; i++){try {Thread.sleep(10);
} catch (InterruptedException e) {e.printStackTrace();}
   c++;   }
   System.out.println("Exiting Increment Method");   }
   public void decrement() {
   System.out.println("Entering decrement Method");
   for (int i=0; i<=100; i++){
   try {Thread.sleep(10);}
 catch (InterruptedException e) {e.printStackTrace();}
   c--;   }
   System.out.println("Exiting decrement Method");  }
   public int value() {   return c;   }}
```

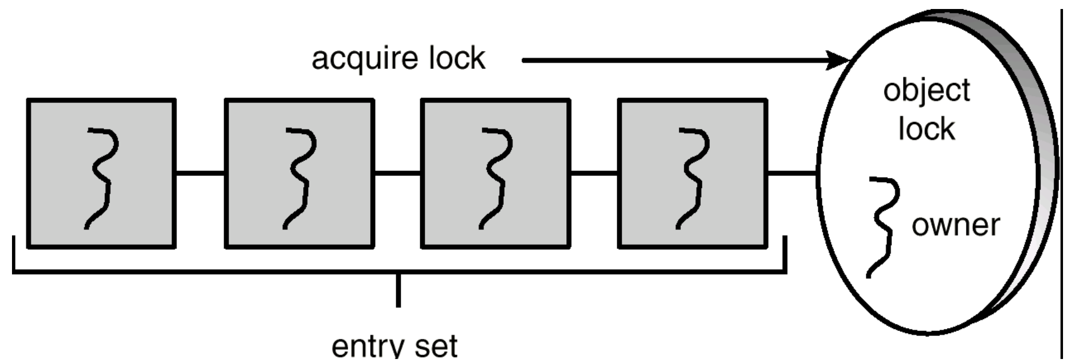
Race Condition Example-II

```
public class testCounter {
public static void main(String[] args) {
Counter c1 = new Counter();
incrementingThread t1 = new incrementingThread(c1);
decrementingThread t2 = new decrementingThread(c1);
t1.start();
t2.start();
try {
t1.join();
t2.join();
} catch (InterruptedException e) {
e.printStackTrace();
}
System.out.println(c1.value());
}
}
```

Java “Synchronized”

- Every object has a lock associated with it.
- Calling a synchronized method requires “owning” the lock.
- If a calling thread does not own the lock (another thread already owns it), the calling thread is placed in the entry set for the object’s lock.
- The lock is released when a thread exits the synchronized method.

```
public class SynchronizedCounter {
private int c = 0;
public synchronized void increment() {
 c++; }
public synchronized void decrement() {
c--; }
public synchronized int value() {
return c; } }
```



Synchronized Example

```
class incrementingThread extends
Thread{
private Counter counter1;
public incrementingThread(Counter
counter1){this.counter1 = counter1;}
public void run(){
this.counter1.increment();
}}
```

```
class decrementingThread extends
Thread{
private Counter counter1;
public decrementingThread(Counter
counter1){this.counter1 = counter1;
}
public void run(){
this.counter1.decrement();
}}
```

```
class Counter {
   private int c = 0;
   public synchronized  void increment() {
   System.out.println("Entering Increment Method");
   for (int i=0; i<=100; i++){try {Thread.sleep(10);
} catch (InterruptedException e) {e.printStackTrace();}
   c++;    }
   System.out.println("Exiting Increment Method");   }
   public synchronized  void decrement() {
   System.out.println("Entering decrement Method");
   for (int i=0; i<=100; i++){
   try {Thread.sleep(10);}
 catch (InterruptedException e) {e.printStackTrace();}
   c--;    }
   System.out.println("Exiting decrement Method");  }
   public synchronized  int value() {    return c;   }}
```

Synchronized Example-II

```
public class testCounter {
public static void main(String[] args) {
Counter c1 = new Counter();
incrementingThread t1 = new incrementingThread(c1);
decrementingThread t2 = new decrementingThread(c1);
t1.start();
t2.start();
try {
t1.join();
t2.join();
} catch (InterruptedException e) {
e.printStackTrace();
}
System.out.println(c1.value());
}
}
```

Block Synchronization

- A mechanism where a block can be labeled as synchronized
- Blocks of code – rather than entire methods – may be declared as synchronized.
- This yields a lock scope that is typically smaller than a synchronized method.
- The synchronized keyword takes as a parameter an object whose lock the system needs to obtain before it can continue

Block Synchronization

- Synchronized methods are effectively implementable as

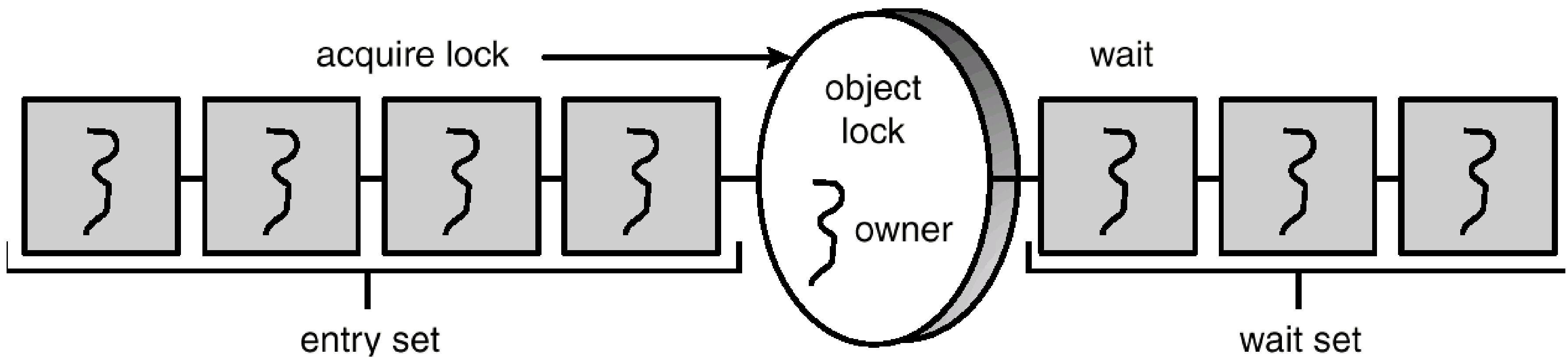
```
public int read() {
  synchronized(this) {
   return theData;
  }
}
```

```
Object O = new Object();
synchronized(O) {
   …
}
```

- `this` is the Java mechanism for obtaining the current object

Java wait/notify notifyall

- When a thread calls wait(), the following occurs:
  - the thread releases the object lock.
  - thread state is set to blocked.
  - thread is placed in the wait set.



Java notify/notifyall

- When a thread calls notify(), the following occurs:
  - selects an arbitrary thread *T* from the wait set.
  - moves *T* to the entry set.
  - sets *T* to Runnable.

T can now compete for the object's lock again.

- notify() selects an arbitrary thread from the wait set. *This may not be the thread that you want to be selected.
- Java does not allow you to specify the thread to be selected.
- notifyAll() removes ALL threads from the wait set and places them in the entry set. This allows the threads to decide among themselves who should proceed next.
- notifyAll() is a conservative strategy that works best when multiple threads may be in the wait set.

Static Data

- Static data is shared between all objects created from the class
- In Java, classes themselves are also objects and there is a lock associated with the class
- This lock may be accessed by either labeling a static method with the synchronized modifier or by identifying the class's object in a synchronized block statement
- The latter can be obtained from the `Object` class associated with the object
- Note that this class-wide lock is not obtained when synchronizing on the object

Assignment

- Producer/Consumer using wait notify.

Accessing Shared Data by Threads

1. Write a concurrent Java program which declares an array of 10 integers. Create two threads. One which writes the value 1 to all the elements in the array. However, in between writing each element of the array, the thread sleeps for one second. The second thread writes the values 7 to each element of the array. Again in between writing the elements it sleeps. When both threads have terminated, print out the array. Reverse the order of starting the threads, and repeat the experiment. Understand what is happening.
2. If 1. above produced (1,7,1,7,1,7,1,...) or (7,1,7,1,7,1,7,...), modify the program so that, even with the sleeps, the result at the end of the threads is (1,1,1,1,1,1,1,1,1,1) or (7,7,7,7,7,7,7,7,7,7). You will need to use a synchronized method or a synchronized statement to achieve this.
3. If 1. above produced (1,1,1,1,1,1,1,...) or (7,7,7,7,7,7,7,..), modify the program so that, even with the sleeps, the result at the end of the threads is (1,7,1,7,1,7,1,17,...) or (7,1,7,1,7,1,7,1,7,1,..). You will need to use a synchronized method or a synchronized statement to achieve this.

Daemon threads

- Threads that run for benefit of other threads
  - E.g., garbage collector
- Run in background
  - Use processor time that would otherwise go to waste
- Unlike normal threads, do not prevent a program from terminating - when only daemon threads remain, program exits
- Must designate a thread as daemon before **`start`** called:

  `void setDaemon( true );`
- Method **`boolean isDaemon()`**
- Returns **`true`** if thread is a daemon thread