Multi-Threaded Programming Design

CSCI 201

Principles of Software Development

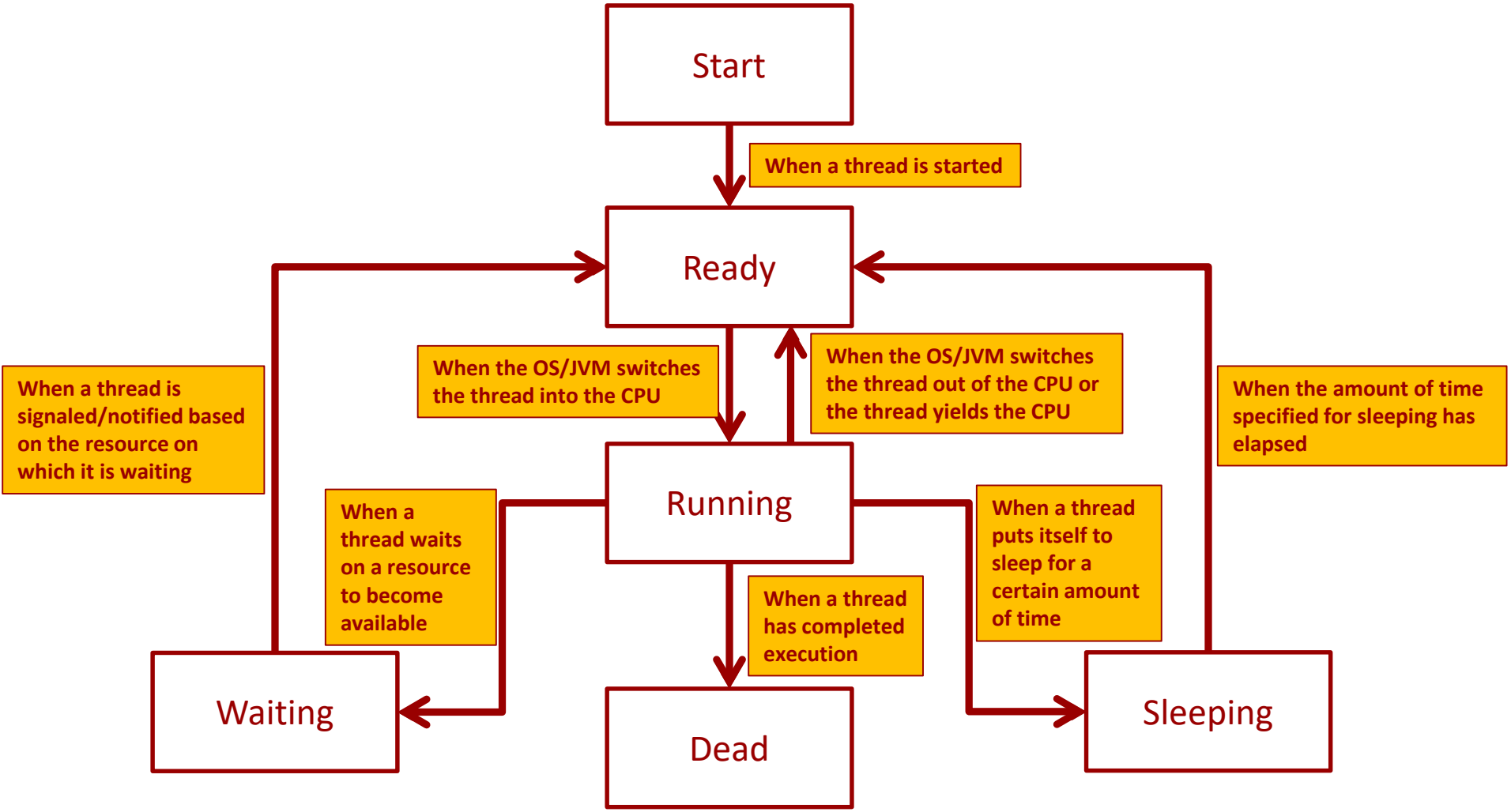
Jeffrey Miller, Ph.D.
jeffrey.miller@usc.edu

USC Viterbi
School of Engineering

Outline

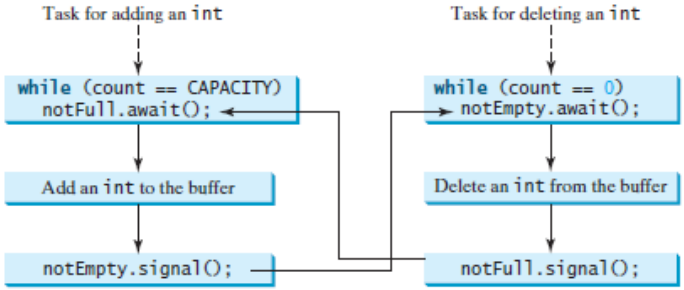
- Blocking Queues
- Multi-Threaded Programming Design

Thread States Review



Producer/Consumer Problem Review

- There are two classes – `Producer` and `Consumer`
- In a shared variable (`buffer` in the following example), the `Producer` increases the value and the `Consumer` decreases the value
- The shared variable has a maximum capacity that the value cannot exceed (`CAPACITY` in the following example)
  - If the `Producer` tries to add a value when the buffer has reached its capacity, it must wait for the `Consumer` (with the condition `notFull` in the following example)
- The shared variable has a minimum capacity that the value cannot pass (0 in the following example)
  - If the `Consumer` tries to decrease the value when the buffer has reached its minimum capacity, it must wait for the `Producer` (with the condition `notEmpty` in the following example)



Producer/Consumer Example with Monitors

```
1   import java.util.LinkedList;
2   import java.util.concurrent.ExecutorService;
3   import java.util.concurrent.Executors;
4
5   public class ProducerConsumerWithMonitors {
6     private static Buffer buffer = new Buffer();
7
8     public static void main(String [] args) {
9       ExecutorService executor = Executors.newFixedThreadPool(2);
10      executor.execute(new ProducerTask());
11      executor.execute(new ConsumerTask());
12      executor.shutdown();
13    }
14
15    private static class ProducerTask implements Runnable {
16      public void run() {
17        try {
18          int i = 1;
19          while (true) {
20            System.out.println("Producer writes: " + i);
21            buffer.write(i++);
22            Thread.sleep((int)(Math.random() * 1000));
23          }
24        } catch (InterruptedException ie) {
25          System.out.println("Producer IE: " + ie.getMessage());
26        }
27      }
28    }
29
30    private static class ConsumerTask implements Runnable {
31      public void run() {
32        try {
33          while (true) {
34            System.out.println("\t\tConsumer reads: " + buffer.read());
35            Thread.sleep((int)(Math.random() * 1000));
36          }
37        } catch (InterruptedException ie) {
38          System.out.println("Consumer IE: " + ie.getMessage());
39        }
40      }
41    }
```

```
42    private static class Buffer {
43      private static final int CAPACITY = 1;
44      private LinkedList<Integer> queue = new LinkedList<Integer>();
45      private static Object notEmpty = new Object();
46      private static Object notFull = new Object();
47      public void write(int value) {
48        synchronized(notFull) {
49          synchronized(notEmpty) {
50            try {
51              while (queue.size() == CAPACITY) {
52                System.out.println("Wait for notFull condition " + value);
53                notFull.wait();
54              }
55              queue.offer(value);
56              notEmpty.notify();
57            } catch (InterruptedException ie) {
58              System.out.println("Buffer.write IE: " + ie.getMessage());
59            }
60          }
61        }
62      }
63      public int read() {
64        int value = 0;
65         synchronized(notFull) {
66          synchronized(notEmpty) {
67            try {
68              while (queue.isEmpty()) {
69                System.out.println("\t\tWait for notEmpty condition");
70                notEmpty.wait();
71              }
72              value = queue.remove();
73              notFull.notify();
74            } catch (InterruptedException ie) {
75              System.out.println("Buffer.read IE: " + ie.getMessage());
76            }
77          }
78        }
79        return value;
80      }
81    } // ends class Buffer
82  } // ends class ProducerConsumerWithMonitors
```

Producer/Consumer Example with Locks/Conditions

```
1   import java.util.LinkedList;
2   import java.util.concurrent.*;
3   import java.util.concurrent.locks.*;
4
5   public class ProducerConsumerWithLocks {
6     private static Buffer buffer = new Buffer();
7
8     public static void main(String [] args) {
9       ExecutorService executor = Executors.newFixedThreadPool(2);
10      executor.execute(new ProducerTask());
11      executor.execute(new ConsumerTask());
12      executor.shutdown();
13    }
14    private static class ProducerTask implements Runnable {
15      public void run() {
16        try {
17          int i = 1;
18          while (true) {
19            System.out.println("Producer tries to write: " + i);
20            buffer.write(i++);
21            Thread.sleep((int)(Math.random() * 1000));
22          }
23        } catch (InterruptedException ie) {
24          System.out.println("Producer IE: " + ie.getMessage());
25        }
26      }
27    }
28    private static class ConsumerTask implements Runnable {
29      public void run() {
30        try {
31          while (true) {
32            System.out.println("\t\tConsumer reads: " + buffer.read());
33            Thread.sleep((int)(Math.random() * 1000));
34          }
35        } catch (InterruptedException ie) {
36          System.out.println("Consumer IE: " + ie.getMessage());
37        }
38      }
39    }
```

```
40    private static class Buffer {
41      private static final int CAPACITY = 1;
42      private LinkedList<Integer> queue = new LinkedList<Integer>();
43      private static Lock lock = new ReentrantLock();
44      private static Condition notEmpty = lock.newCondition();
45      private static Condition notFull = lock.newCondition();
46      public void write(int value) {
47        lock.lock();
48        try {
49          while (queue.size() == CAPACITY) {
50            System.out.println("Wait for notFull condition " + value);
51            notFull.await();
52          }
53          queue.offer(value);
54          notEmpty.signal();
55        } catch (InterruptedException ie) {
56          System.out.println("Buffer.write IE: " + ie.getMessage());
57        } finally {
58          lock.unlock();
59        }
60      }
61      public int read() {
62        int value = 0;
63        lock.lock();
64        try {
65          while (queue.isEmpty()) {
66            System.out.println("\t\tWait for notEmpty condition");
67            notEmpty.await();
68          }
69          value = queue.remove();
70          notFull.signal();
71        } catch (InterruptedException ie) {
72          System.out.println("Buffer.read IE: " + ie.getMessage());
73        } finally {
74          lock.unlock();
75          return value;
76        }
77      }
78    } // ends class Buffer
79  } // ends class ProducerConsumerWithLocks
```

Producer/Consumer Output

```
Producer writes: 1
                        Consumer reads: 1
                        Wait for notEmpty condition
Producer writes: 2
                        Consumer reads: 2
                        Wait for notEmpty condition
Producer writes: 3
                        Consumer reads: 3
Producer writes: 4
                        Consumer reads: 4
Producer writes: 5
                        Consumer reads: 5
Producer writes: 6
                        Consumer reads: 6
Producer writes: 7
                        Consumer reads: 7
Producer writes: 8
Producer writes: 9
Wait for notFull condition
                        Consumer reads: 8
Producer writes: 10
Wait for notFull condition
```

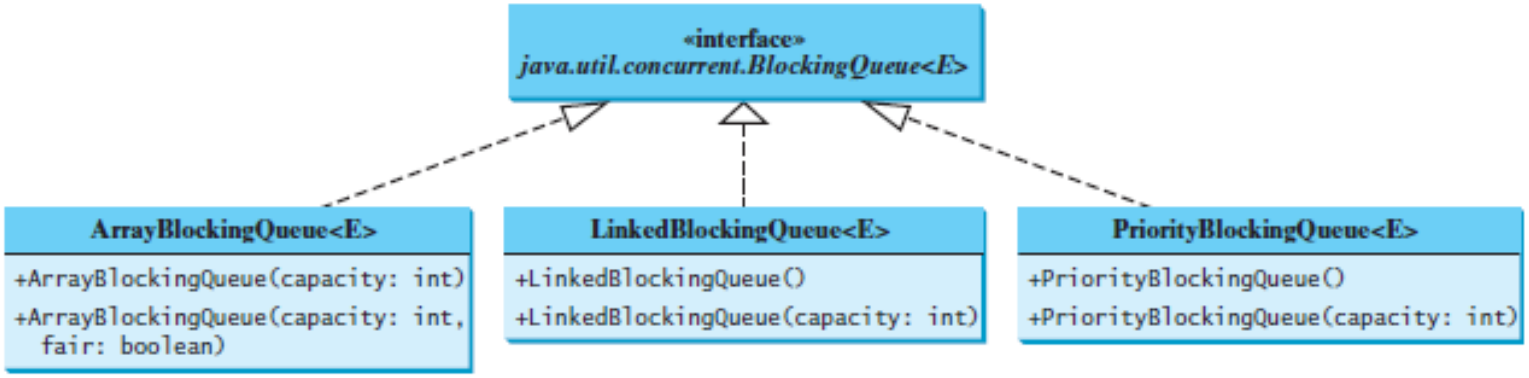
```
Producer writes: 1
                        Wait for notEmpty condition
                        Consumer reads: 1
Producer writes: 2
                        Consumer reads: 2
                        Wait for notEmpty condition
Producer writes: 3
                        Consumer reads: 3
                        Wait for notEmpty condition
Producer writes: 4
                        Consumer reads: 4
                        Wait for notEmpty condition
Producer writes: 5
                        Consumer reads: 5
                        Wait for notEmpty condition
Producer writes: 6
                        Consumer reads: 6
Producer writes: 7
Producer writes: 8
Wait for notFull condition
                        Consumer reads: 7
                        Consumer reads: 8
Producer writes: 9
```

```
Producer writes: 1
                        Wait for notEmpty condition
                        Consumer reads: 1
                        Wait for notEmpty condition
Producer writes: 2
                        Consumer reads: 2
                        Wait for notEmpty condition
Producer writes: 3
                        Consumer reads: 3
                        Wait for notEmpty condition
Producer writes: 4
                        Consumer reads: 4
```

```
Producer writes: 1
                        Consumer reads: 1
Producer writes: 2
                        Consumer reads: 2
Producer writes: 3
Producer writes: 4
Wait for notFull condition
                        Consumer reads: 3
                        Consumer reads: 4
Producer writes: 5
```

Blocking Queues

- A blocking queue causes a thread to block (i.e. move to the waiting state) when you try to add an element to a full queue or to remove an element from an empty queue
  - It will remain there until the queue is no longer full or no longer empty
  - There are three blocking queues in Java: `ArrayBlockingQueue`, `LinkedBlockingQueue`, and `PriorityBlockingQueue`



Producer/Consumer with Blocking Queues

```
1   import java.util.concurrent.*;
2   public class ProducerConsumer {
3     private static ArrayBlockingQueue<Integer> buffer = new ArrayBlockingQueue<Integer>(2);
4     public static void main(String [] args) {
5       ExecutorService executor = Executors.newFixedThreadPool(2);
6       executor.execute(new ProducerTask());
7       executor.execute(new ConsumerTask());
8       executor.shutdown();
9     }
10    private static class ProducerTask implements Runnable {
11      public void run() {
12        try {
13          int i = 1;
14          while (true) {
15            System.out.println("Producer writes: " + i);
16            buffer.put(i++);
17            Thread.sleep((int)(Math.random() * 10000));
18          }
19        } catch (InterruptedException ie) {
20          System.out.println("Producer IE: " + ie.getMessage());
21        }
22      }
23    }
24    private static class ConsumerTask implements Runnable {
25      public void run() {
26        try {
27          while (true) {
28            System.out.println("\t\t\tConsumer reads: " + buffer.take());
29            Thread.sleep((int)(Math.random() * 10000));
30          }
31        } catch (InterruptedException ie) {
32          System.out.println("Consumer IE: " + ie.getMessage());
33        }
34      }
35    }
36  }
```

```
Producer writes: 1
                        Consumer reads: 1
Producer writes: 2
Producer writes: 3
                        Consumer reads: 2
Producer writes: 4
                        Consumer reads: 3
                        Consumer reads: 4
Producer writes: 5
                        Consumer reads: 5
Producer writes: 6
                        Consumer reads: 6
Producer writes: 7
                        Consumer reads: 7
```

```
Producer writes: 1
                        Consumer reads: 1
Producer writes: 2
                        Consumer reads: 2
Producer writes: 3
                        Consumer reads: 3
Producer writes: 4
                        Consumer reads: 4
Producer writes: 5
                        Consumer reads: 5
Producer writes: 6
                        Consumer reads: 6
Producer writes: 7
Producer writes: 8
Producer writes: 9
                        Consumer reads: 7
```

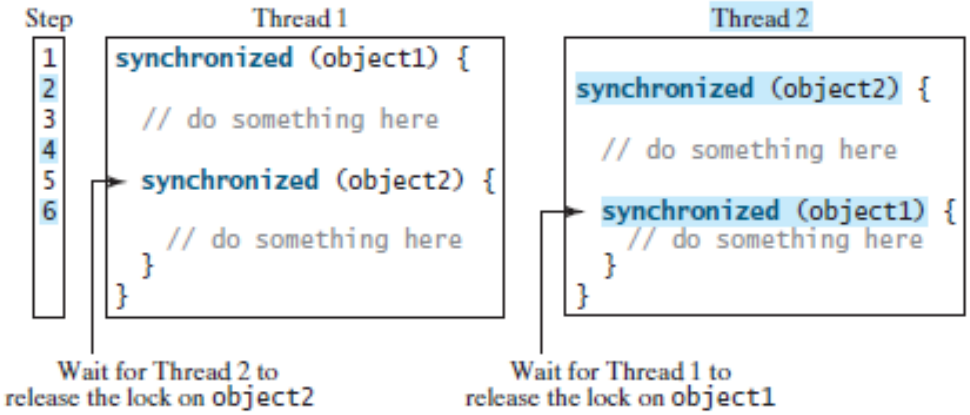
USC Viterbi School of Engineering

Outline

- Blocking Queues
- Multi-Threaded Programming Design

Avoiding Deadlock

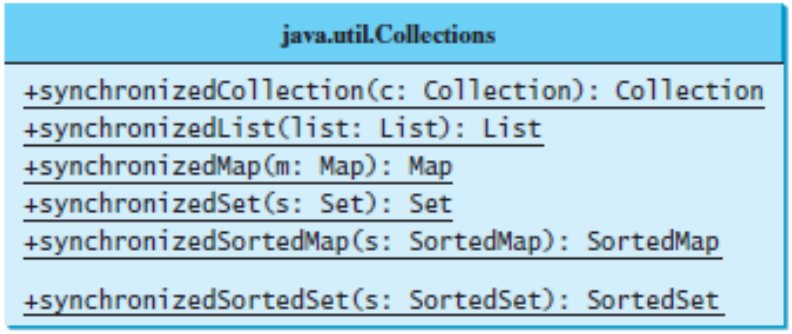
- Deadlock can occur when two threads are both waiting on locks the other thread has



- This can be avoided if locks are obtained in the same order every time
  - If the lock on `object1` is always obtained before the lock on `object2`, deadlock will be avoided in the above example
  - NOTE: If Thread1 waits on `object1` inside the `object2` synchronization, deadlock can still occur (which is what we did in the ProducerConsumer example with monitors)

Java Collections Synchronization

- The classes in the Java Collections framework are not thread-safe
  - `Vector`, `Stack`, and `Hashtable` are thread-safe, but they are older objects (since version 1.0)
  - They have been replaced by `ArrayList`, `LinkedList`, and `Map` (since version 1.2)
- There are methods in the `Collections` class that can be used for obtaining thread-safe versions of any of the Collection objects
  - A synchronized collection object has synchronized versions of all the methods that access and update the original collection
  - Note: There are many more methods in the `Collections` class



Iterators and Synchronized Collections

- Even though we can get a synchronized `Collections` object, the iterator is fail-fast (not synchronized)
  - If the collection being iterated over is modified by another thread, the iterator will throw a `java.util.ConcurrentModificationException`
  - We can avoid this by obtaining a lock on the object over which we are iterating before we begin iterating

Synchronized Collections Example #1

```
1   import java.util.Collections;
2   import java.util.HashSet;
3   import java.util.Iterator;
4   import java.util.Set;
5
6   public class CollectionsTest {
7
8     public static void main(String [] args) {
9       for (int i=0; i < 100; i++) {
10        MyThread mt = new MyThread(i);
11        mt.start();
12      }
13    }
14  }
15
16  class MyThread extends Thread {
17    private static Set<Integer> hashSet = Collections.synchronizedSet(new HashSet<Integer>());
18    private int num;
19    public MyThread(int num) {
20      this.num = num;
21      hashSet.add(num);
22    }
23    public void run() {
24      System.out.print("thread " + num + ": ");
25      Iterator<Integer> iterator = hashSet.iterator();
26      while (iterator.hasNext()) {
27        System.out.print(iterator.next() + " ");
28      }
29      System.out.println();
30    }
31  }
```

```
thread 0: thread 1: 0 1 2 3 4 0 1 2 3 5 4 6 5 7 6 8 7 9 8 thread 3: 0 1 Exception in thread "Thread-0" Exception in thread "Thread
0       at java.util.HashMap$HashIterator.nextEntry(Unknown Source)
        at java.util.HashMap$KeyIterator.next(Unknown Source)
        at MyThread.run(Test.java:33)
Exception in thread "Thread-2" thread 5: java.util.ConcurrentModificationException0
1 2 3   at java.util.HashMap$HashIterator.nextEntry(Unknown Source)4 5 6
7 8     at java.util.HashMap$KeyIterator.next(Unknown Source)9
10 11 12        at MyThread.run(Test.java:33)13
14 15 17 16 java.util.ConcurrentModificationException19
18 21 20 23 22
        at java.util.HashMap$HashIterator.nextEntry(Unknown Source)
        at java.util.HashMap$KeyIterator.next(Unknown Source)
        at MyThread.run(Test.java:33)
thread 13: 0 java.util.ConcurrentModificationException
        at java.util.HashMap$HashIterator.nextEntry(Unknown Source)
        at java.util.HashMap$KeyIterator.next(Unknown Source)
        at MyThread.run(Test.java:33)
Exception in thread "Thread-13" java.util.ConcurrentModificationException
        at java.util.HashMap$HashIterator.nextEntry(Unknown Source)
        at java.util.HashMap$KeyIterator.next(Unknown Source)
```

Synchronized Collections Example #2

```
1   import java.util.*;
2   import java.util.concurrent.locks.*;
3
4   public class CollectionsTest {
5     public static void main(String [] args) {
6       for (int i=0; i < 100; i++) {
7         MyThread mt = new MyThread(i);
8         mt.start();
9       }
10    }
11  }
12  class MyThread extends Thread {
13    private static Set<Integer> hashSet = Collections.synchronizedSet(new HashSet<Integer>());
14    private static Lock lock = new ReentrantLock();
15    private int num;
16    public MyThread(int num) {
17      this.num = num;
18      hashSet.add(num);
19    }
20    public void run() {
21      lock.lock();
22      try {
23        System.out.print("thread " + num + ": ");
24        Iterator<Integer> iterator = hashSet.iterator();
25        while (iterator.hasNext()) {
26          System.out.print(iterator.next() + " ");
27        }
28        System.out.println();
29      } finally {
30        lock.unlock();
31      }
32    }
33  }
```

```
thread 0: 0 1 2 3 4 5 6 7 8 9 10 11 12
thread 2: 0 1 2 3 4 5 6 7 8 9 10 11 12
thread 5: 0 Exception in thread "Thread-5" java.util.ConcurrentModificationException
        at java.util.HashMap$HashIterator.nextEntry(Unknown Source)
        at java.util.HashMap$KeyIterator.next(Unknown Source)
        at MyThread.run(Test.java:34)
thread 7: 0 Exception in thread "Thread-7" java.util.ConcurrentModificationException
        at java.util.HashMap$HashIterator.nextEntry(Unknown Source)
        at java.util.HashMap$KeyIterator.next(Unknown Source)
        at MyThread.run(Test.java:34)
thread 6: 0 1 2 3 Exception in thread "Thread-6" java.util.ConcurrentModificationException
        at java.util.HashMap$HashIterator.nextEntry(Unknown Source)
        at java.util.HashMap$KeyIterator.next(Unknown Source)
        at MyThread.run(Test.java:34)
thread 9: 0 1 2 Exception in thread "Thread-9" java.util.ConcurrentModificationException
        at java.util.HashMap$HashIterator.nextEntry(Unknown Source)
        at java.util.HashMap$KeyIterator.next(Unknown Source)
        at MyThread.run(Test.java:34)
thread 1: 0 Exception in thread "Thread-1" java.util.ConcurrentModificationException
        at java.util.HashMap$HashIterator.nextEntry(Unknown Source)
        at java.util.HashMap$KeyIterator.next(Unknown Source)
```

Synchronized Collections Example #3

```
1   import java.util.*;
2   import java.util.concurrent.locks.*;
3
4   public class CollectionsTest {
5     public static void main(String [] args) {
6       for (int i=0; i < 100; i++) {
7         MyThread mt = new MyThread(i);
8         mt.start();
9       }
10    }
11  }
12  class MyThread extends Thread {
13    private static Set<Integer> hashSet = Collections.synchronizedSet(new HashSet<Integer>());
14    private static Lock lock = new ReentrantLock();
15    private int num;
16    public MyThread(int num) {
17      this.num = num;
18      lock.lock();
19      try {
20        hashSet.add(num);
21      } finally {
22        lock.unlock();
23      }
24    }
25    public void run() {
26      lock.lock();
27      try {
28        System.out.print("thread " + num + ": ");
29        Iterator<Integer> iterator = hashSet.iterator();
30        while (iterator.hasNext()) {
31          System.out.print(iterator.next() + " ");
32        }
33        System.out.println();
34      } finally {
35        lock.unlock();
36      }
37    }
38  }
```

```
thread 0: 0 1 2
thread 1: 0 1 2
thread 2: 0 1 2 3
thread 3: 0 1 2 3 4 5
thread 4: 0 1 2 3 4 5 6
thread 5: 0 1 2 3 4 5 6 7 8
thread 6: 0 1 2 3 4 5 6 7 8
thread 7: 0 1 2 3 4 5 6 7 8 9
thread 8: 0 1 2 3 4 5 6 7 8 9 10
thread 10: 0 1 2 3 4 5 6 7 8 9 10
thread 9: 0 1 2 3 4 5 6 7 8 9 10 11
thread 11: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
thread 12: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
thread 13: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
thread 14: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

Synchronized Collections Example #4

```
1   import java.util.*;
2
3   public class CollectionsTest {
4     public static void main(String [] args) {
5       for (int i=0; i < 100; i++) {
6         MyThread mt = new MyThread(i);
7         mt.start();
8       }
9     }
10  }
11  class MyThread extends Thread {
12    private static Set<Integer> hashSet = Collections.synchronizedSet(new HashSet<Integer>());
13    private int num;
14    public MyThread(int num) {
15      this.num = num;
16      synchronized(hashSet) {
17        hashSet.add(num);
18      }
19    }
20    public void run() {
21      synchronized(hashSet) {
22        System.out.print("thread " + num + ": ");
23        Iterator<Integer> iterator = hashSet.iterator();
24        while (iterator.hasNext()) {
25          System.out.print(iterator.next() + " ");
26        }
27        System.out.println();
28      }
29    }
30  }
```

```
thread 0: 0 1 2
thread 1: 0 1 2
thread 2: 0 1 2 3
thread 3: 0 1 2 3 4 5
thread 4: 0 1 2 3 4 5 6
thread 5: 0 1 2 3 4 5 6 7 8
thread 6: 0 1 2 3 4 5 6 7 8
thread 7: 0 1 2 3 4 5 6 7 8 9
thread 8: 0 1 2 3 4 5 6 7 8 9 10
thread 10: 0 1 2 3 4 5 6 7 8 9 10
thread 9: 0 1 2 3 4 5 6 7 8 9 10 11
thread 11: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
thread 12: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
thread 13: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
thread 14: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

Synchronized Collections Example #5

```
1   import java.util.*;
2
3   public class CollectionsTest {
4     public static void main(String [] args) {
5       for (int i=0; i < 100; i++) {
6         MyThread mt = new MyThread(i);
7         mt.start();
8       }
9     }
10  }
11  class MyThread extends Thread {
12    private static Set<Integer> hashSet = Collections.synchronizedSet(new HashSet<Integer>());
13    private int num;
14    public MyThread(int num) {
15      this.num = num;
16      hashSet.add(num);
17    }
18    public void run() {
19      synchronized (hashSet) {
20        System.out.print("thread " + num + ": ");
21        Iterator<Integer> iterator = hashSet.iterator();
22        while (iterator.hasNext()) {
23          System.out.print(iterator.next() + " ");
24        }
25        System.out.println();
26      }
27    }
28  }
```

```
thread 0: 0 1 2 3
thread 3: 0 1 2 3
thread 2: 0 1 2 3
thread 1: 0 1 2 3
thread 4: 0 1 2 3 4 5
thread 5: 0 1 2 3 4 5
thread 6: 0 1 2 3 4 5 6 7 8
thread 8: 0 1 2 3 4 5 6 7 8
thread 7: 0 1 2 3 4 5 6 7 8 9
thread 9: 0 1 2 3 4 5 6 7 8 9 10
thread 10: 0 1 2 3 4 5 6 7 8 9 10
thread 11: 0 1 2 3 4 5 6 7 8 9 10 11 12 13
thread 12: 0 1 2 3 4 5 6 7 8 9 10 11 12 13
thread 13: 0 1 2 3 4 5 6 7 8 9 10 11 12 13
thread 14: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
thread 15: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
thread 16: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
thread 17: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 17 16 18
thread 18: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 17 16 18
thread 19: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 17 16 19 18 20
```

Multi-Threaded Programming Rules

- Use one lock for one resource
- Always acquire locks in the same order in different threads
- Always release locks in the opposite order they were acquired
  - If acquiring multiple locks, never release the outer lock without releasing the inner lock first
- Always synchronize iterating through a data structure if the variable is shared across multiple threads
- Use the `Collections` framework for creating thread-safe data structures instead of `Vector`, `Stack`, and `Hashtable`