Threads

Mārtiņš Leitass

Task Scheduling

Modern operating systems use preemptive multitasking to allocate CPU time to applications. There are two types of tasks that can be scheduled for execution:

- Processes: A process is an area of memory that contains both code and data. A process has a thread of execution that is scheduled to receive CPU time slices.
- Thread: A thread is a scheduled execution of a process. Concurrent threads are possible. All threads for a process share the same data memory but may be following different paths through a code section.

Concurrent programming

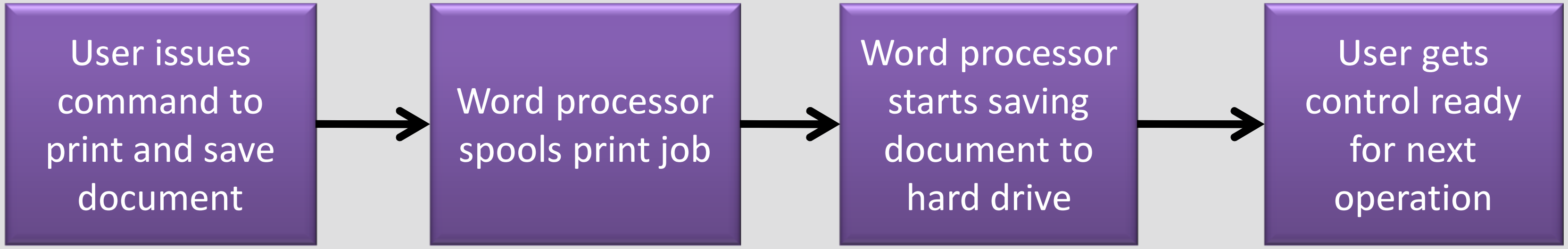
- with a single-processor CPU, only one thread is executing at any given time
- on multi-processor systems several threads are actually executing at the same time (physical concurrency)
- **multi-threading** occurs when concurrency exists among threads running in a single process (also referred to as multi-tasking)

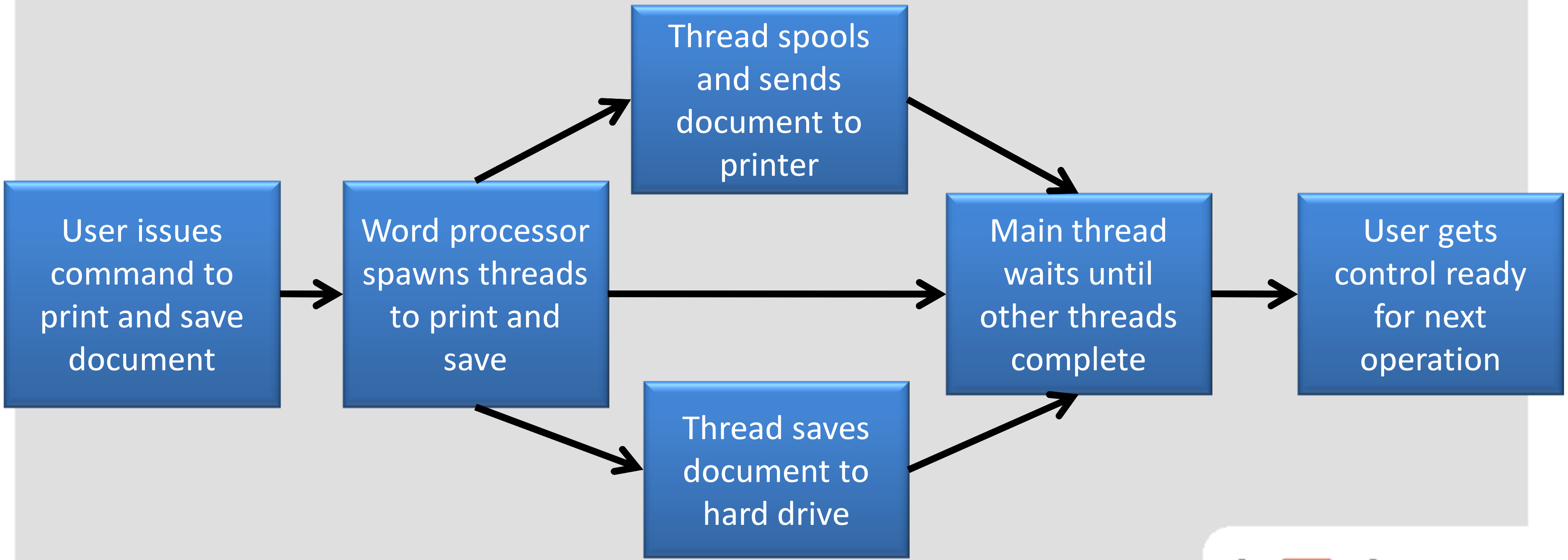
Why Threading Matters

To execute a program as quickly as possible, you must avoid performance bottlenecks. Some of these bottlenecks are:

- Resource Contention: Two or more tasks waiting for exclusive use of a resource
- Blocking I/O operations: Doing nothing while waiting for disk or network data transfers
- Underutilization of CPUs: A single-threaded application uses only a single CPU

Concurrent programming





Java Threads

- Java provides support for multi-threading as a part of the language
- support centers on the:
  - java.lang.Thread class
  - java.lang.Runnable interface
  - java.lang.Object methods wait(), notify(), and notifyAll
  - synchronized keyword
- every Java program has at least one thread which is executed when main() is invoked
- all user-level threads are explicitly constructed and started from the main thread or by a thread originally started from main()
- when the last **user** thread completes any **daemon** threads are stopped and the application stops
- a thread's default daemon status is the same as that of thread creating it
  - you can check the daemon status using **isDaemon()**
  - you can set the daemon status using **setDaemon()**.
- You cannot change a thread's status after it has been started

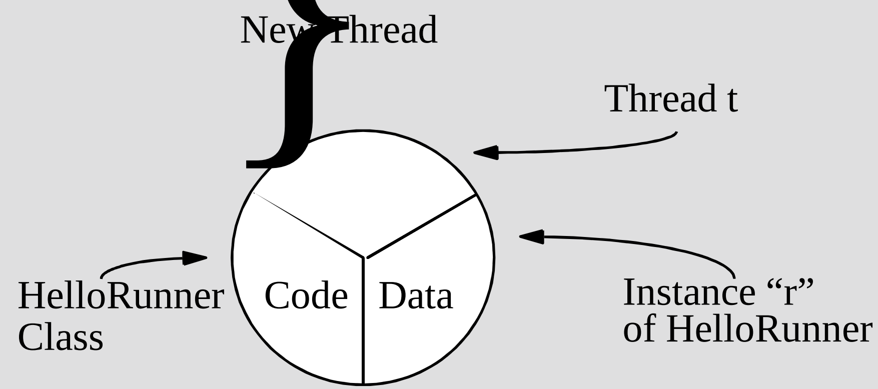
Creating the Thread

```
public class ThreadTester {
   public static void main(String args[]) {
      HelloRunner r = new HelloRunner();
      Thread t = new Thread(r);
      t.start();
   }
}
class HelloRunner implements Runnable {
   int i;
   public void run() {
      i = 0;
      while (true) {
         System.out.println("Hello " + i++);
         if ( i == 50 ) {
            break;
         }
      }
   }
}
```

Creating the Thread

- Multithreaded programming has these characteristics:
  - Multiple threads are from one Runnable instance.
  - Threads share the same data and code.
- For example:

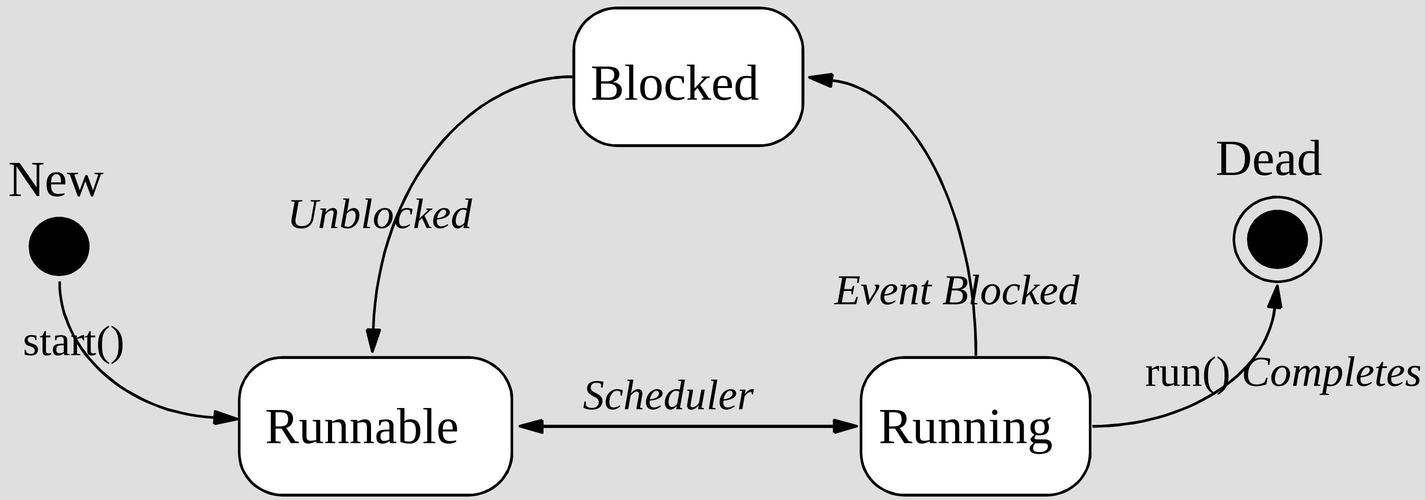
```
Thread t1 = new Thread(r);
Thread t2 = new Thread(r);
```



Starting the Thread

- Use the start method.
- Place the thread in a runnable state.

Thread Scheduling



Thread Scheduling Example

```
public class Runner implements Runnable {
   public void run() {
      while (true) {
         // do lots of interesting stuff
         // ...
         // Give other threads a chance
         try {
            Thread.sleep(10);
         } catch (InterruptedException e) {
            // This thread's sleep was interrupted
            // by another thread
         }
      }
   }
}
```

Terminating a Thread

```
public class Runner implements Runnable {
   private boolean timeToQuit=false;

   public void run() {
      while ( ! timeToQuit ) {
         // continue doing work
      }
      // clean up before run() ends
   }

   public void stopRunning() {
      timeToQuit=true;
   }
}
```

Terminating a Thread

```
public class ThreadController {
    private Runner r = new Runner();
    private Thread t = new Thread(r);

    public void startThread() {
        t.start();
    }

    public void stopThread() {
        // use specific instance of Runner
        r.stopRunning();
    }
}
```

Basic Control of Threads

- Test threads:

  isAlive()

- Access thread priority:

  getPriority()
  setPriority()

- Put threads on hold:

```
Thread.sleep()      // static method
join()
Thread.yield()      // static method
```

The join Method

```
public static void main(String[] args) {
   Thread t = new Thread(new Runner());
   t.start();
   ...
   // Do stuff in parallel with the other thread for a while
   ...
   // Wait here for the other thread to finish
   try {
      t.join();
   } catch (InterruptedException e) {
      // the other thread came back early
   }
   ...
   // Now continue in this thread
   ...
}
```

Other Ways to Create Threads

```
public class MyThread extends Thread {
    public void run() {
        while ( true ) {
            // do lots of interesting stuff
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                // sleep interrupted
            }
        }
    }

    public static void main(String args[]) {
        Thread t = new MyThread();
        t.start();
    }
}
```

Selecting a Way to Create Threads

- Implement Runnable:
  - Better object-oriented design
  - Single inheritance
  - Consistency
- Extend Thread:

  Simpler code

Out-of-Order Execution

- Operations performed in one thread may not appear to execute in order if you observe the results from another thread.
  - Code optimization may result in out-of-order operation.
  - Threads operate on cached copies of shared variables.
- To ensure consistent behavior in your threads, you must synchronize their actions.
  - You need a way to state that an action happens before another.
  - You need a way to flush changes to shared variables back to main memory.

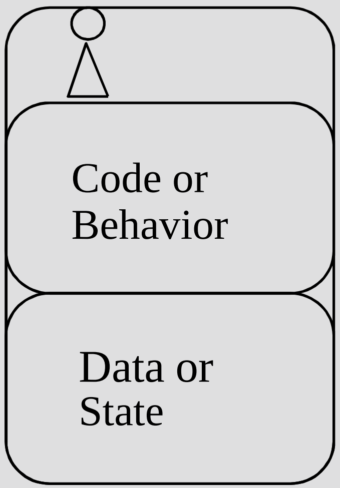
Using the synchronized Keyword

```
public class MyStack {

   int idx = 0;
   char [] data = new char[6];

   public void push(char c) {
      data[idx] = c;
      idx++;
   }

   public char pop() {
      idx--;
      return data[idx];
   }
}
```

The Object Lock Flag

- Every object has a flag that is a type of *lock flag*.
- The synchronized enables interaction with the lock flag.

Object this



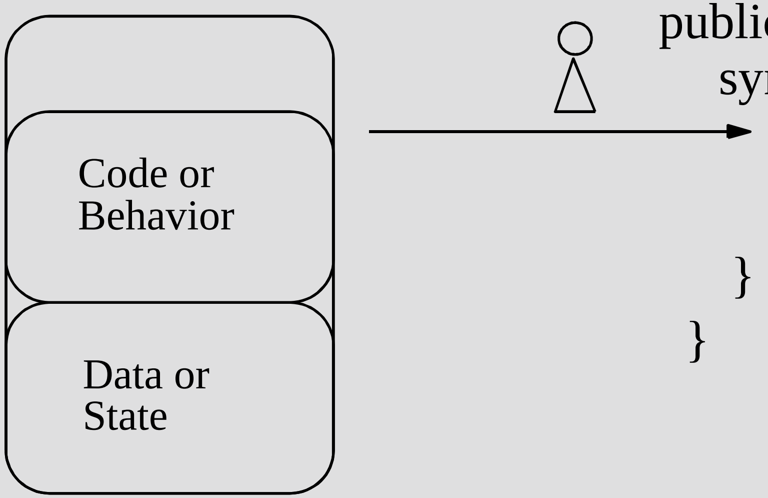
Thread before synchronized(this)

```
public  void  push(char  c)  {
  synchronized  (this)  {
      data[idx]  =  c;
      idx++;
    }
}
```

The Object Lock Flag

Object this

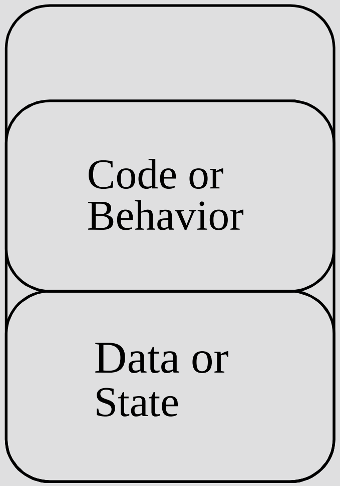
Thread after synchronized(this)




```
public void push(char c) {
  synchronized (this) {
    data[idx] = c;
    idx++;
  }
}
```

The Object Lock Flag

Object this
lock flag missing



Another thread, trying to
execute synchronized(this)

Waiting for
object lock

```
public char pop() {
  synchronized (this) {
      idx--;
      return data[idx];
   }
}
```

Releasing the Lock Flag

The lock flag is released in the following events:

- Released when the thread passes the end of the synchronized code block
- Released automatically when a break, return, or exception is thrown by the synchronized code block

Object Monitor Locking

- `synchronized` methods use the monitor for the this object.
- `static synchronized` methods use the classes' monitor.
- `synchronized` blocks must specify which object's monitor to lock or unlock.
- `synchronized` blocks can be nested.

Using synchronized– Putting It Together

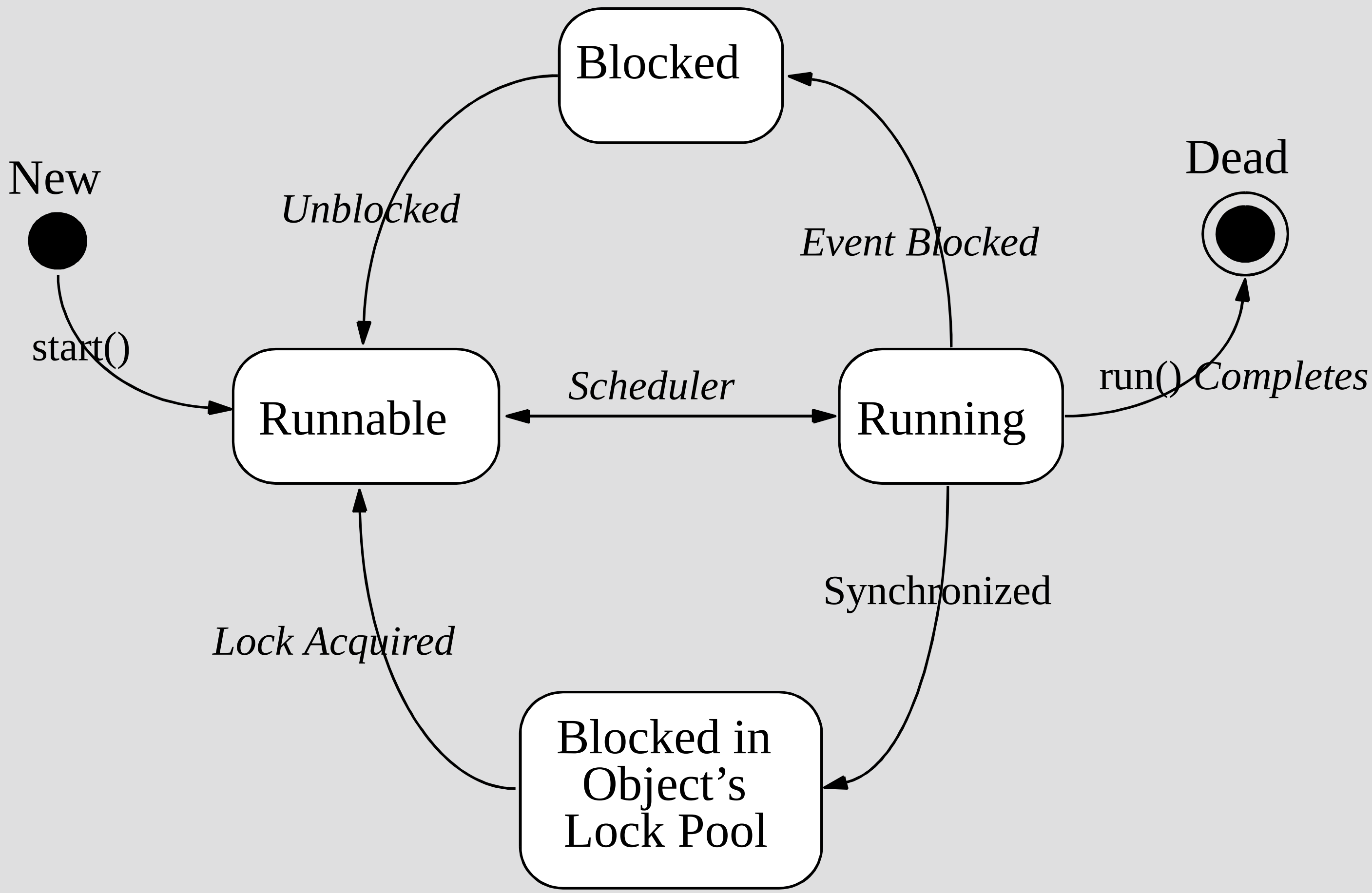
- *All* access to delicate data should be synchronized.
- Delicate data protected by synchronized should be private.

Using synchronized– Putting It Together

The following two code segments are equivalent:

```
public void push(char c) {
   synchronized(this) {
      // The push method code
   }
}

public synchronized void push(char c) {
   // The push method code
}
```

Thread State Diagram With Synchronization



latitelecom

The volatile Keyword

A field may have the volatile modifier applied to it:

- Reading or writing a volatile field will cause a thread to synchronize its working memory with main memory.
- volatile does not mean atomic.
  - If i is volatile, i++ is still not a thread-safe operation.

```
public volatile int i;
```

Deadlock

A deadlock has the following characteristics:

- It is two threads, each waiting for a lock from the other.
- It is not detected or avoided.
- Deadlock can be avoided by:
  - Deciding on the order to obtain locks
  - Adhering to this order throughout
  - Releasing locks in reverse order

Thread Interaction – wait and notify

- Scenario:

  Consider yourself and a cab driver as two threads.
- The problem:

  How do you determine when you are at your destination?
- The solution:
  - You notify the cab driver of your destination and relax.
  - The driver drives and notifies you upon arrival at your destination.

Thread Interaction

Thread interactions include:

- The wait and notify methods
- The pools:
  - Wait pool
  - Lock pool

Monitor Model for Synchronization

- Leave shared data in a consistent state.
- Ensure programs cannot deadlock.
- Do not put threads expecting different notifications in the same wait pool.

The Producer Class

```
package mod13;

public class Producer implements Runnable {
    private SyncStack theStack;
    private int num;
    private static int counter = 1;

    public Producer (SyncStack s) {
        theStack = s;
        num = counter++;
    }

```

The Producer Class

```
public void run() {
    char c;

    for (int i = 0; i < 200; i++) {
        c = (char)(Math.random() * 26 +’A’);
        theStack.push(c);
        System.out.println(“Producer” + num + “: “ + c);
        try {
            Thread.sleep((int)(Math.random() * 300));
        } catch (InterruptedException e) {
            // ignore it
        }
    }
  } // END run method

} // END Producer class
```

The Consumer Class

```
package mod13;

public class Consumer implements Runnable {
   private SyncStack theStack;
   private int num;
   private static int counter = 1;

   public Consumer (SyncStack s) {
      theStack = s;
      num = counter++;
   }

```

The Consumer Class

```
public void run() {
        char c;
        for (int i = 0; i < 200; i++) {
            c = theStack.pop();
            System.out.println(“Consumer” + num + “: “ + c);

            try {
                Thread.sleep((int)(Math.random() * 300));
            } catch (InterruptedException e) {
                // ignore it
            }
        }
    } // END run method
```

The SyncStackClass

This is a sketch of the SyncStack class:

```
public class SyncStack {

   private List<Character> buffer = new ArrayList<Character>(400);

   public synchronized char pop() {
      // pop code here
   }

   public synchronized void push(char c) {
      // push code here
   }
}
```

The popMethod

```
public synchronized char pop() {
        char c;
        while (buffer.size() == 0) {
            try {
                this.wait();
            } catch (InterruptedException e) {
                // ignore it...
            }
        }
        c = buffer.remove(buffer.size()-1);
        return c;
    }
```

The pushMethod

```
public synchronized void push(char c) {
        this.notify();
        buffer.add(c);
    }
```

The SyncTestClass

```
package mod13;
public class SyncTest {
   public static void main(String[] args) {
      SyncStack stack = new SyncStack();
      Producer p1 = new Producer(stack);
      Thread prodT1 = new Thread (p1);
      prodT1.start();
      Producer p2 = new Producer(stack);
      Thread prodT2 = new Thread (p2);
      prodT2.start();

      Consumer c1 = new Consumer(stack);
      Thread consT1 = new Thread (c1);
      consT1.start();
      Consumer c2 = new Consumer(stack);
      Thread consT2 = new Thread (c2);
      consT2.start();
   }
}
```

The SyncTestClass

Producer2:	F
Consumer1:	F
Producer2:	K
Consumer2:	K
Producer2:	T
Producer1:	N
Producer1:	V
Consumer2:	V
Consumer1:	N
Producer2:	V
Producer2:	U
Consumer2:	U
Consumer2:	V
Producer1:	F
Consumer1:	F
Producer2:	M
Consumer2:	M
Consumer2:	T

Methods to Avoid

Some Thread methods should be avoided:

- setPriority(int) and getPriority()
  - Might not have any impact or may cause problems
- The following methods are deprecated and should never be used:
  - destroy()
  - resume()
  - suspend()
  - stop(