Java Threads

CSc 335

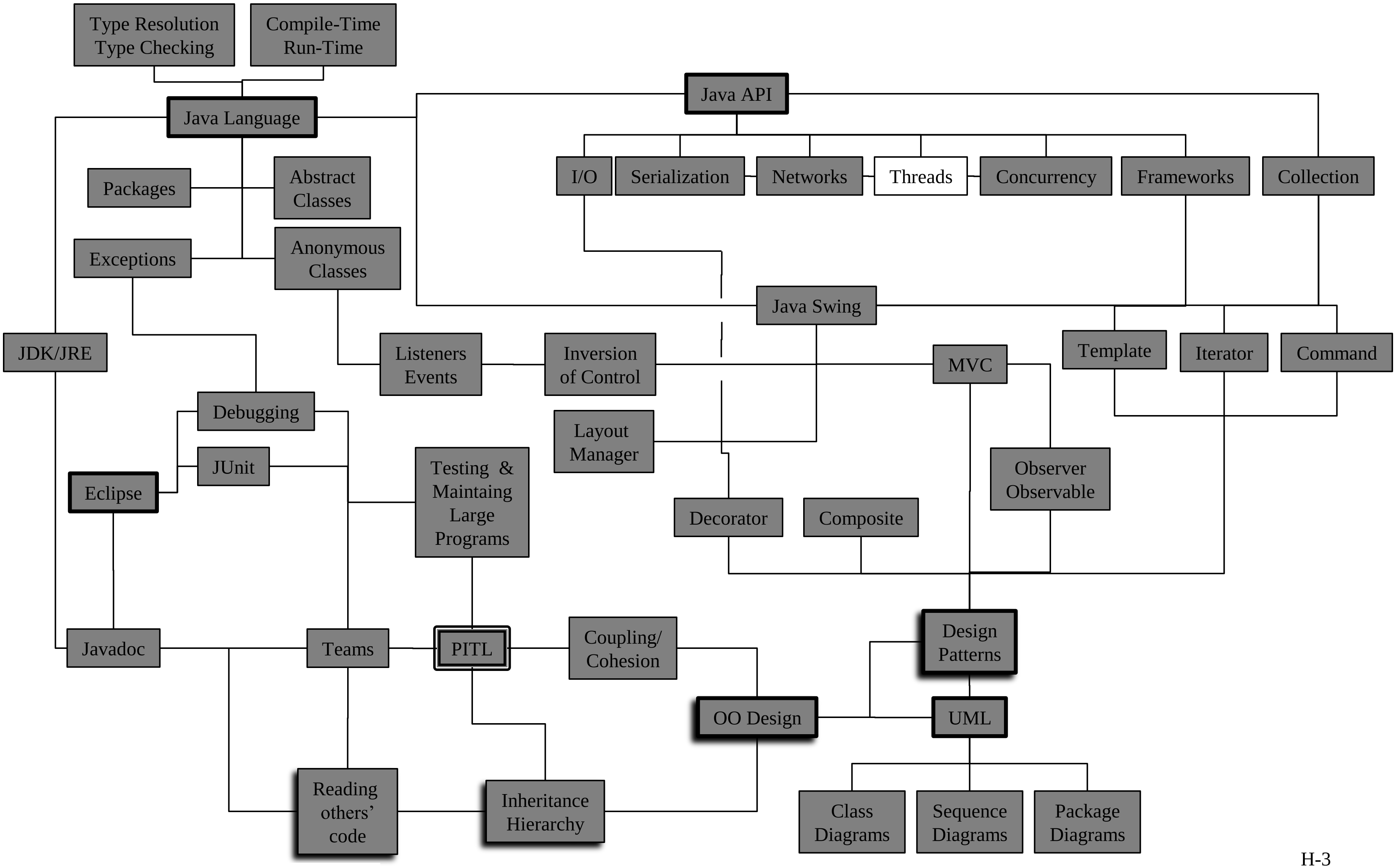
Object-Oriented Programming and Design

Spring 2009

Acknowledgements

- Some materials from the following texts was used:
  - **The Theory and Practice of Concurrency**, by A.W. Roscoe, Prentice Hall, 1997, ISBN 0-13-674409-5.
  - **Java In A Nutshell (5th Ed.)**, by David Flanagan, O'Reilly Media, 2005, ISBN 0-596-00773-6.
- Slides by Ivan Vazquez, with some help from Rick Snodgrass.

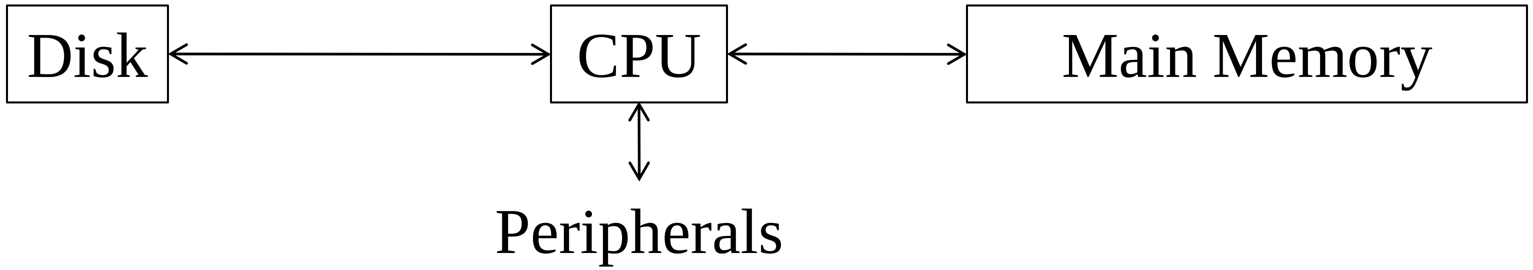
Java Threads



Outline

- Basic concepts
  - Processes
  - Threads
  - Java: `Thread` class
  - Java: `runnable` Interface
  - Single-threaded vs. Multi-Threads
  - Concurrent Programming
- Thread Safety
- Inter-Thread Control
- Caveats

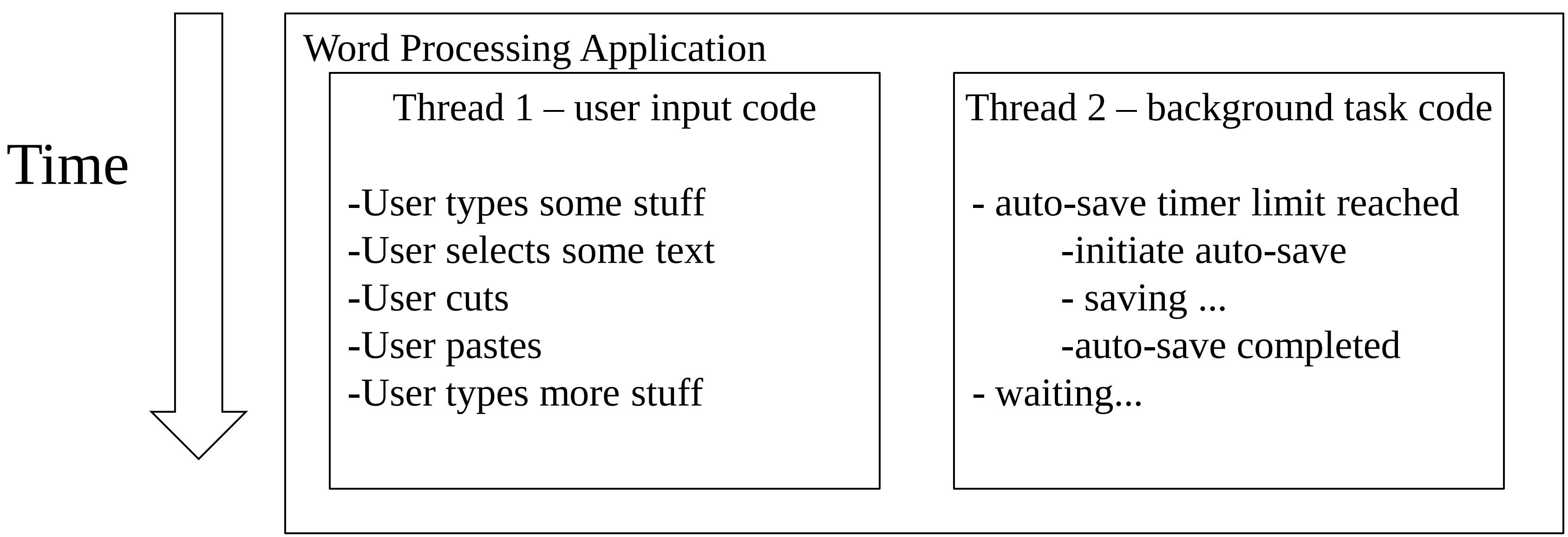
Processes



- Each *process* has
  - Program counter
  - Registers
  - Page map address (address space)
  - Open files, etc.
- CPU *context switches* between processes
  - Saves registers of prior process
  - Loads register of new process
  - Loads new page map
- A process is *heavy weight*.
  - Lot of state
  - Context switch takes time

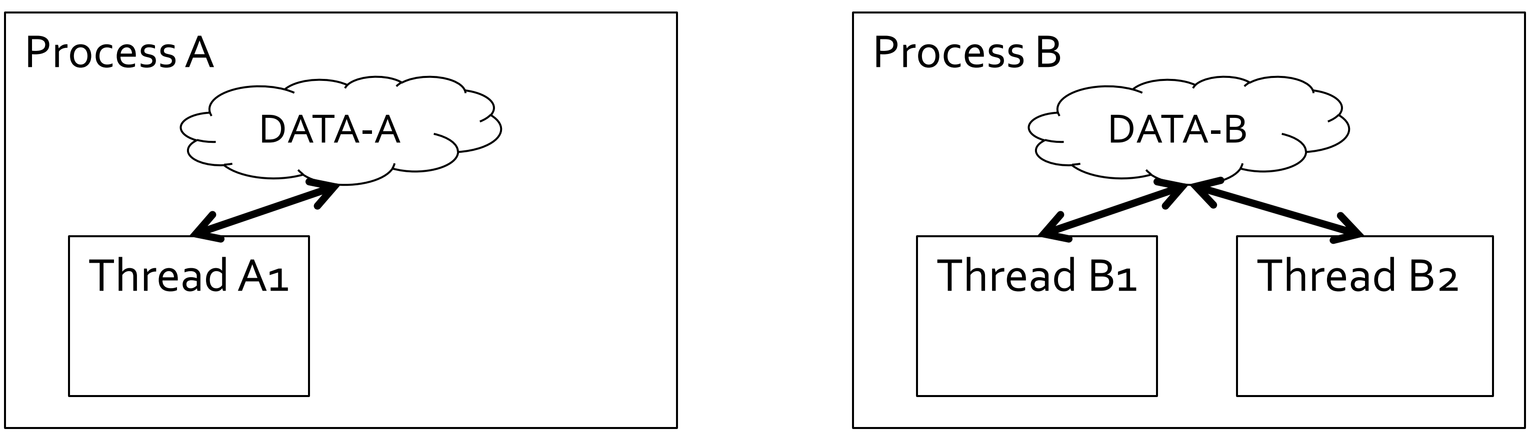
What Are Threads?

- As an example program using threads, a word processor should be able to accept input from the user and at the same time, auto-save the document.
- The word processing application contains two threads:
  - One to handle user-input
  - Another to process background tasks (like auto-saving).



Programming Perspective

- The term *thread* is short for *thread of control*.
- A *thread* is a programming concept very similar to a process. But a process can contain multiple *threads*.
- *Threads* share the same data, while processes each have their own set of data: threads are *light-weight*.
- Note that your Java programs are being executed in a *thread* already (the "main" thread).



Single-Threaded Vs. Multi-Threaded

- A typical Java program is *single-threaded*. This means there is only one thread running.
- If more than one thread is running *concurrently* then a program is considered *multi-threaded*.
- The following example is *single-threaded*. (The only thread running the main thread.)

```
public class SingleThreadedExample {
    public static void main(String[] args) {
      for( int i = 0; i < 10; i++ ) {
        mySleep(250); // milliseconds
        System.out.println( "Main: " + i );
      }
    }
}
```

Output:

```
Main: 0
Main: 1
Main: 2
Main: 3
Main: 4

  ...
```

Using the `Thread` Class

- Java provides the `Thread` class to create and control Threads.
- To create a thread, one calls the constructor of a sub-class of the `Thread` class.
- The `run()` method of the new class serves as the body of the thread.
- A new instance of the sub-classed `Thread` is created in a running thread.
- The new thread (and its `run()` method) is started when `start()` is called on the `Thread` object.

```
public class ExampleThread extends Thread {
  public void run() {            // new thread's body
    ... // do stuff in the thread
  }
  public static void main(String[] args) {   // main thread's body
    Thread thread = new ExampleThread();
    thread.start();
    ...
```

- After the `thread.start()` call we have two threads active: the main thread and the newly started thread.

The `Runnable` Interface

- Another way of creating a `Thread` in Java is to pass the `Thread` constructor an object of type `Runnable`.
- The `Runnable` interface requires only the `run()` method, which serves as the body of the new thread. (`Thread` implements `Runnable`.)
- As before, the new thread (and its `run()` method) is started when `start()` is called on the `Thread` object.

new thread's body

main thread's body

```
public class ExRunnable implements Runnable {
  public void run() {
    ... // do stuff in the thread
  }
  public static void main(String[] args) {
    Thread thread
      = new Thread(new ExRunnable());
    thread.start();
    ...
```

Single-Threaded Vs. Multi-Threaded (contd.)

- Here we create and run two `CountThread` instances.

```
public class CountThread extends Thread {
  public CountThread(String s) { super(s); }
  public void run() {
    for( int i = 0; i < 10; i++ ) {
      mySleep(500);  // milliseconds
      System.out.println(this.getName()+ ":" + i );
    }

  public static void main(String[] args) {
    Thread t1 = new CountThread("t1");
    Thread t2 = new CountThread("t2");
    t1.start(); t2.start();
    ...
```

Output:

```
t1:0
        t2:0
t1:1
        t2:1
t1:2
        t2:2
t1:3
        t2:3
t1:4
        t2:4
t1: 5
        ...
```

- Threads t1 and t2 run simultaneously, each counting up to 10 in parallel.

Concurrent Programming

- *Concurrency* is a property of systems in which several threads are executing at the same time, and potentially interacting with each other.
- The biggest challenge in dealing with *concurrent* systems is in avoiding conflicts between threads.
- For example: what if our application wants to access the same data from two different threads at the same time?

Outline

- Basic concepts
- Thread Safety
  - Atomic actions
  - Synchronized modifier
  - Transient modifier
  - Concurrent atomic package
  - Concurrent collection
- Inter-Thread Control
- Caveats

Thread Safety

- If a class or method can be used by different threads *concurrently*, without chance of corrupting any data, then they are called *thread-safe*.
- Writing *thread-safe* code requires careful thought and design to avoid problems at run-time.
- It is important to document whether or not code is *thread-safe*. For example, much of the Java's Swing package is not *thread-safe*.

Java And Thread-Safety

- Java provides a number of powerful tools to make it relatively easily to implement *thread-safe* code.
  - Atomic actions
  - The `synchronized` modifier
  - The `transient` modifier
  - The `concurrent.atomic` package
  - The concurrent and synchronized collections

Atomic Actions

- An *atomic* action is one that cannot be subdivided and hence cannot be interrupted by another thread.
- Reads and writes are *atomic* for all reference variables and for most primitive variables (except `long` and `double` as they are 64 bits).
- This means that a thread can execute an *atomic* action without fear of interruption by another thread.

Thread Safety 101: Race Conditions

- Using atomic operations doesn't solve all concurrency problems.
- Look at the following constructor which assigns a serial number to an object.

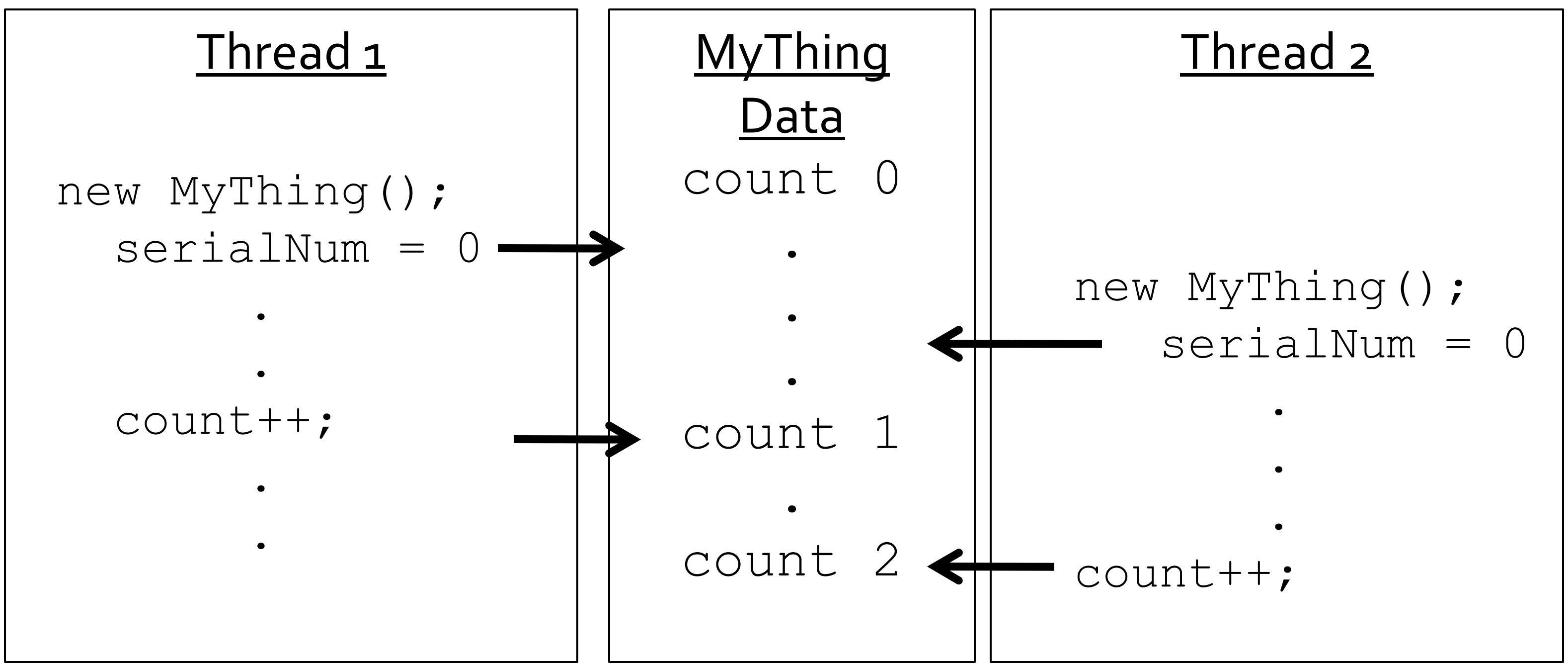
```
// i r Threadsafe ?
public class MyThing {
  static int count = 0;
  private int serialNum;

  public MyThing() {
      serialNum = count;
      count++;
  } ...
```

- What's the problem?
- Two threads could be assigning the same `count` to two different `MyThing` objects in parallel threads.
- This is a *race condition.*

Race Conditions

```
// i r Threadsafe
public class MyThing {
  static private int
  count = 0;
  private int serialNum;
}
```

```
public MyThing() {
    serialNum = count;
    count++;
}
```



Race Conditions (cont.)

- What about increment? `serialNumber = count++;`
- Still doesn't work, because multiple low-level operations are involved:
  - Read `count` into a register
  - Increment that register
  - Store register value in `count` variable
  - Store register value also in `serialNumber`
- What about postincrement? `serialNumber = ++count;`
- Same problem…

Using `synchronized`

- To make this truly thread-safe, we can use Java's `synchronized` keyword.
- `synchronized` means that a thread must obtain a lock on an object (in this case the `MyThing` class object) before it can execute any of its `synchronized` methods on that object.

```
public class MyThing {
  static private int count=0;
  private int serialNum;

  public MyThing() {
       serialNum = getSN();
  }
```

```
  private static synchronized
         int getSN() {
    int newCount = count;

    count++;
    return newCount;
  }
} // End of class MyThing
```

Using `synchronized` (contd.)

- In the previous example, the `synchronized` method is `static`.
- Here it is used on an *instance* method which increments the `instanceCount` variable each time it is called.
- This method increments the `instanceCount` variable in a thread-safe way

```
public class MyThing {
   private int instanceCount;

   public MyThing() { ...
         instanceCount = 0;
   }
  public synchronized int
         incInstCount() {
    this.instanceCount++;
   }
} // End of class MyThing
```

Using `volatile`

- There is one other problem: the JVM permits threads to cache the value of variables in local memory (i.e., a machine register).
- This means the value read could be out of date. To avoid this, we use the `volatile` keyword on fields that are referenced by multiple threads.

`public class MyThing {`
`  private` **`volatile`** `int`
`      instanceCount;`
 	
`  public MyThing() { ...`
`      instanceCount = 0;`
`  }`
	`// thread safe`
	`  public` **`synchronized`** `int`
	`      incInstCount() {`
	`    return`
	`      this.instanceCount++;`
	`  }`
	`}`

Using `synchronized` Blocks

- It is possible to use finer-grained locking mechanisms that minimize the chance of lock conflicts.
- Here we lock only the `instanceCount` variable, so we do not lock the entire object.
- Note that we had to make `instanceCount` be an object (an `Integer`) to be able to use this mechanism.

<pre>public class MyThing {
  private **volatile Integer**
              instanceCount;
 	
  public MyThing() { ...
       instanceCount = 0;
  }</pre>
	<pre>// Also thread-safe
	  public int incInstCount() {
	    synchronized(instanceCount) {
	      return
	         this.instanceCount++;
	    }
	}</pre>

Using `concurrent.atomic`

- The `java.util.concurrent.atomic` package contains utility classes that permit *atomic* operations on objects without locking.

- These classes definine `get()` and `set()` accessor methods as well as compound operations, such as `incrementAndGet( )`.

```
public class MyThing {
   private AtomicInteger
                  instanceCount;

   public MyThing() { ...
       instanceCount =
        new AtomicInteger(0);
   }
```

```
 // Also thread-safe
 public int incInstCount() {
   return
    instanceCount.incrementAndGet();
  }
}
```

Concurrent And Synchronized Collections

- Java provides some concurrent *thread-safe* collections.
  - `BlockingQueue` – a FIFO that blocks when you attempt to add to a full queue, or retrieve from an empty queue
  - `ConcurrentMap` – Maintains a set of key-value pairs in a thread-safe manner.

- Java also provides the *synchronized collection* wrapper classes, which pass through all method calls to the wrapped collection after adding any necessary synchronization.
  - `Collections.synchronized{Collection, Map, Set, List, SortedMap}`

Outline

- Basic concepts
- Thread Safety
- Inter-Thread Control
  - Stopping a thread
  - Waiting for a thread to finish
  - Passing data between threads
  - `BlockingQueue`
- Caveats

Stopping a Thread

- One of the simplest ways to stop a thread is to use a flag variable which can tell a thread to stop executing.
- Here's a flawed implementation of such a beast.

```
public class MyThread extends Thread {
  volatile Boolean done = false;
  public void run() {
    synchronized(done) {
      while(! done) {
         ... // do stuff
      } } }
```

```
  public void stop() {
   synchronized(done) {
      done = true;
    }
  }
```

- The problem is that the `done` variable is locked *outside* of the while loop in the `run()` method, which means it keeps the lock forever during the `while` loop.
- The `stop()` method can never get the lock.
- This is called *lock starvation*.

Stopping a Thread (II)

- To fix this we need to apply one of our basic rules: Hold locks for as short a time as possible.
- Here's a corrected implementation.

```
public class MyThread
      extends Thread {
  volatile Boolean done = false;
  public void run() {
      while(true) {
        synchronized(done) {
         if( done ) {
            break;
         }
        }  // end synchronized block
        ... // do regular loop stuff
      } } }
```

```
  public void stop() {
   synchronized(done) {
      done = true;
    }
  }
```

Waiting for a Thread to Finish

- Java terminates all threads (except for the Swing threads) when the `main()` method exits.
- Sometimes it is necessary to wait for a thread to finish. For example, we might be writing out a file in a thread which we don't want to be terminated partway through its write.
- The `join()` method of the `Thread` class permits us to wait until a thread is finished.

```
public static void main(...) {
       Thread fileWriterThread
              = new FWT();
       fileWriterThread.start();
       ... // time to exit
       // wait for fileWriter to finish.
       fileWriterThread.join();
}
```

Passing Data Between Threads

- One powerful design pattern that is readily applied to multi-threaded applications is the *Producer-Consumer* pattern.
- This is a way of synchronizing between two threads.
- One thread *produces* data, and puts it into a shared buffer or queue, and the other thread *consumes* the data (usually by processing it).
- An example use of this is a printer queue system where print jobs are received by a thread which takes the job and *produces* an entry in a print queue. The *consumer* thread takes the top entry in the queue and prints it. This avoids the confusion of having one thread attempt two jobs at once.
- Java's `BlockingQueue` interface provides methods for such queues that are *thread-safe*.

BlockingQueue

- `BlockingQueue<E>` is an interface with the following methods

	Throws exception	***Special value***	***Blocks***	***Times out***
Insert	`add(e)`	`offer(e)`	`put(e)`	`offer(e,time,unit)`
Remove	`remove()`	`poll()`	`take()`	`poll(time,unit)`
Examine	`element()`	`peek()`	*N/A*	*N/A*

- If a queue method *blocks* then it stop execution of the thread until the method returns. If a method *times out* then the method s*blocks* until the time specified is reached, then the method returns.
- Important implementations of `BlockingQueue` are

  `ArrayBlockingQueue`, `LinkedBlockingQueue`, `PriorityBlockingQueue` and `SynchronousQueue`.

Outline

- Basic concepts
- Thread Safety
- Inter-Thread Control
  - Stopping a thread
  - Waiting for a thread to finish
  - Passing data between threads
  - `BlockingQueue`
- Caveats

Concurrent Programming Caveats

- In general, multi-threaded programming is confusing and difficult to debug. When threading conflicts do occur, they don't always happen in the same way each time.
- When a thread acquires a lock on an object, no other thread can acquire the same lock until the first thread releases the lock. This can lead to a situation where multiple threads are *deadlocked* waiting for a lock to be released.
- Always release locks as quickly as possible.
- Keep your thread-safe code to a minimum and scrutinize it carefully.
- Review your design with someone who can play the devil's advocate and see if they can break your code.