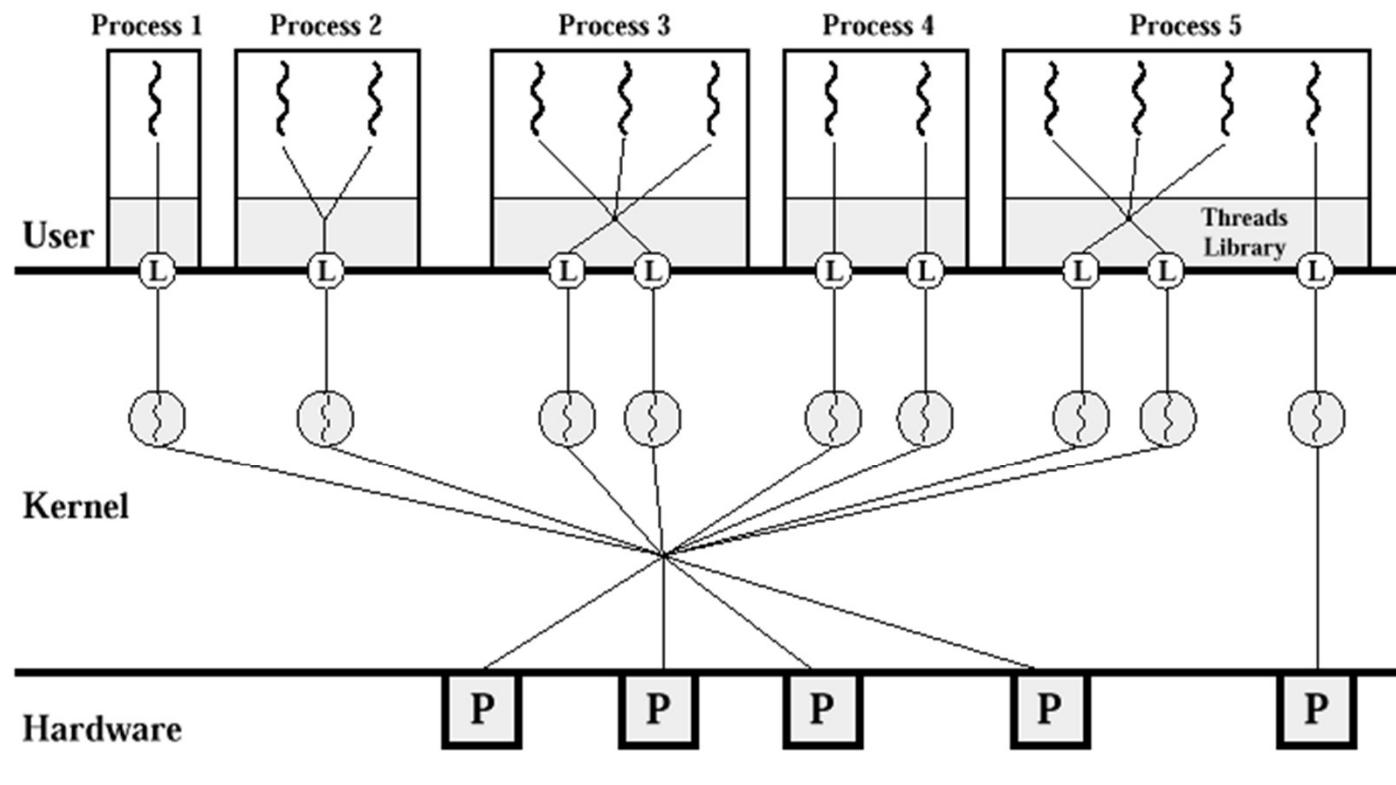




---

# Multithreading and Concurrency Control



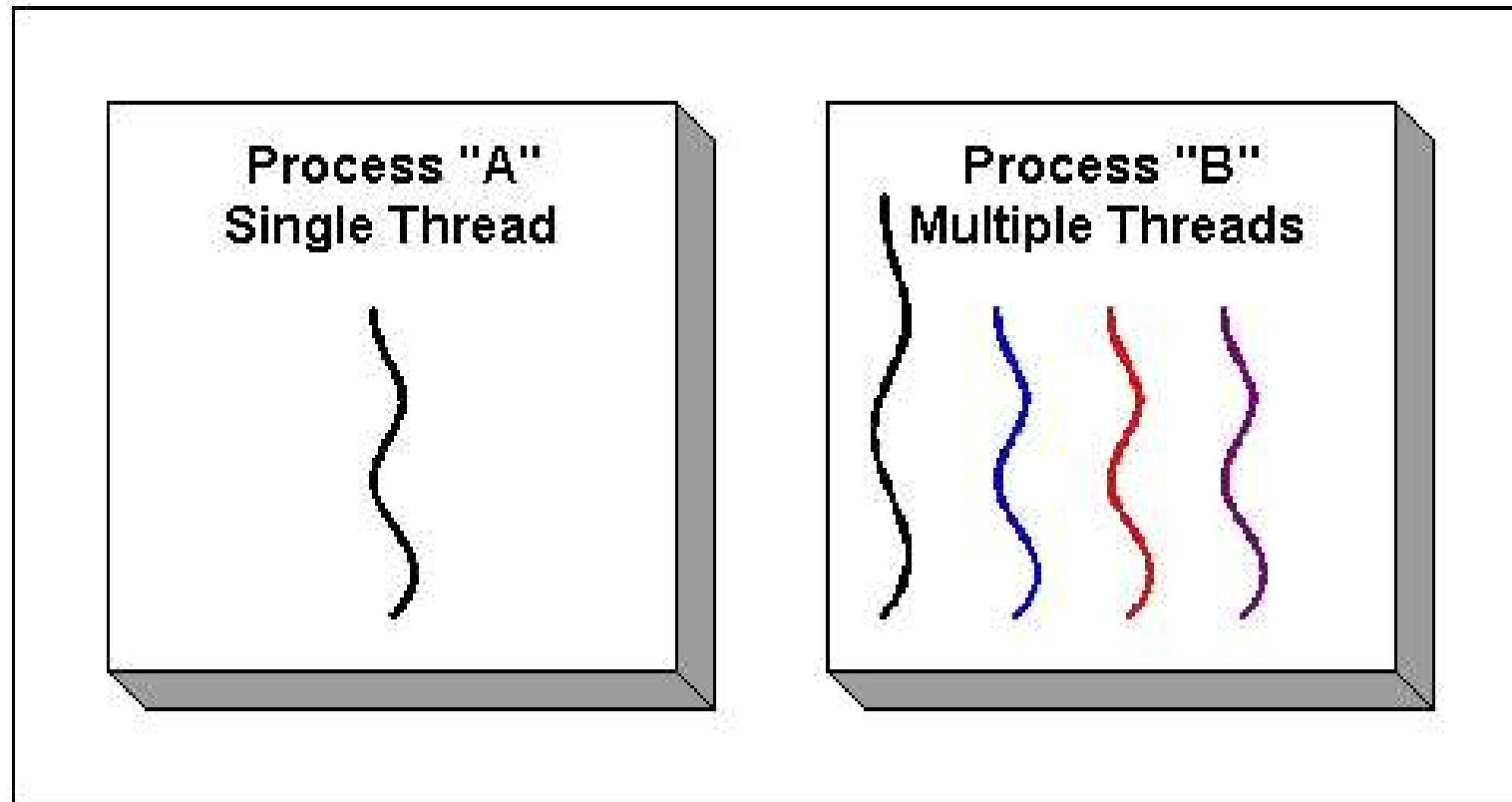


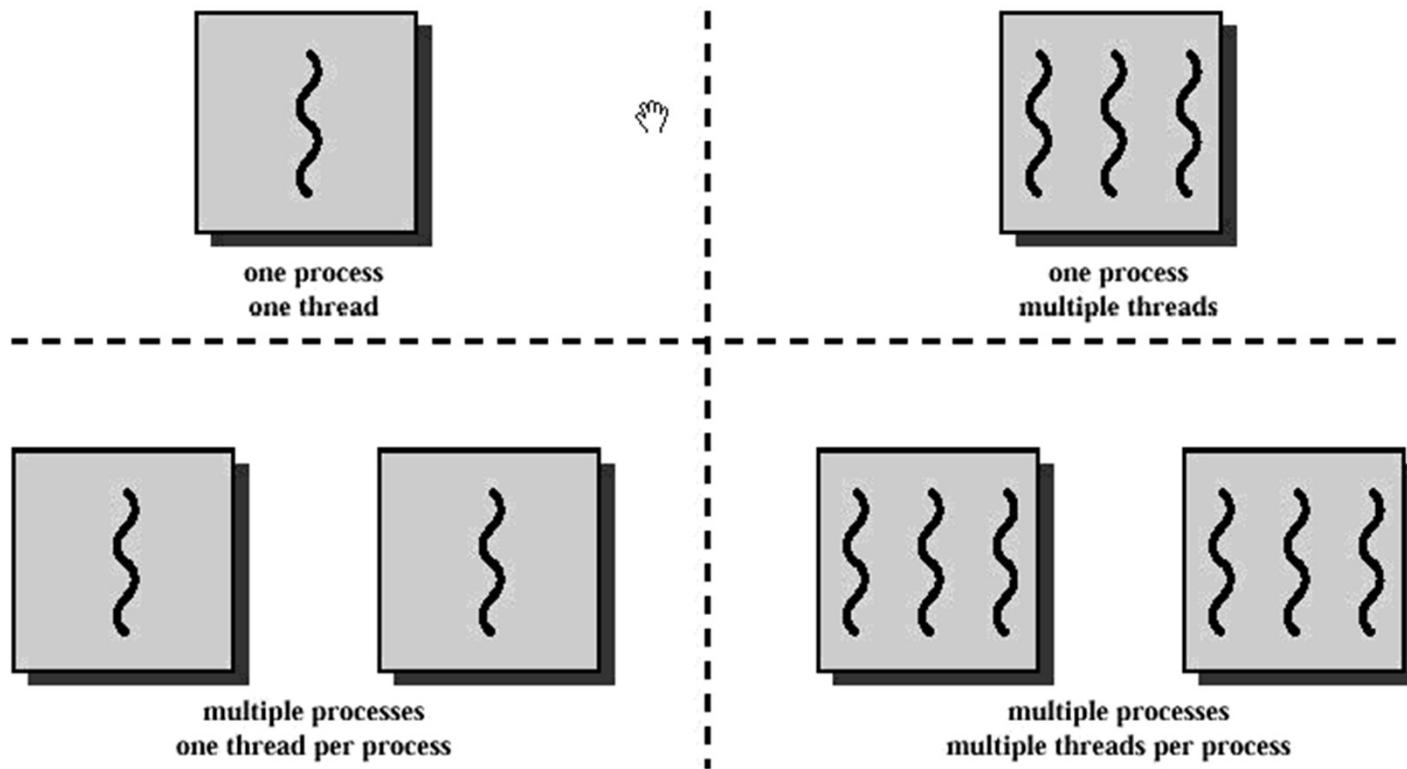
{ User-level thread

( $\circlearrowright$ ) Kernel-level thread

(L) Light-weight Process

P Processor





## Process vs Threads

### Process (JVM)

Each thread has its own stack memory

Thread 1

Stack

method1()

Thread 2

Stack

method1()

Thread 3

Stack

method1()

Heap

Object 1

Object 2

Single heap per process  
shared by all the threads

# Three Java APIs you must know for Multithreading

The screenshot shows a Mozilla Firefox browser window displaying the Java API documentation for the `Thread` class. The URL in the address bar is `docs.oracle.com/javase/7/docs/api/index.html?java/lang/Thread.html`. The browser title is "Thread (Java Platform SE 7) - Mozilla Firefox". The page header includes the Java™ Platform Standard Ed. 7 logo and navigation links for Overview, Package, Class (which is highlighted in orange), Use, Tree, Deprecated, Index, and Help. Below the header, there are links for Prev Class, Next Class, Frames, and No Frames, along with Summary and Detail sections for Nested, Field, Constr, and Method. The main content area is titled "java.lang" and "Class Thread". It lists the `Object` and `Thread` classes as superclasses. Under "All Implemented Interfaces:", it shows the `Runnable` interface. Under "Direct Known Subclasses:", it lists `ForkJoinWorkerThread`. A code snippet at the bottom defines the `Thread` class as extending `Object` and implementing `Runnable`. A descriptive paragraph explains that a `thread` is a thread of execution in a program. A note below states that every thread has a priority, and threads with higher priority are executed before lower priority threads. Threads can also be marked as daemons.

Thread (Java Platform SE 7) - Mozilla Firefox

File Edit View History Bookmarks Tools Help

Thread (Java Platform SE 7)

docs.oracle.com/javase/7/docs/api/index.html?java/lang/Thread.html

Java™ Platform Standard Ed. 7

Java™ Platform Standard Ed. 7

All Classes

Packages

java.applet

java.awt

All Classes

AbstractAction

AbstractAnnotationValueVisitor6

AbstractAnnotationValueVisitor7

AbstractBorder

AbstractButton

AbstractCellEditor

AbstractCollection

AbstractColorChooserPanel

AbstractDocument

AbstractDocument.AttributeConte

AbstractDocument.Content

AbstractDocument.ElementEdit

AbstractElementVisitor6

AbstractElementVisitor7

AbstractExecutorService

AbstractInterruptibleChannel

AbstractLayoutCache

AbstractLayoutCache.NodeDimen

AbstractList

AbstractListModel

Overview Package Class Use Tree Deprecated Index Help

Prev Class Next Class Frames No Frames

Summary: Nested | Field | Constr | Method      Detail: Field | Constr | Method

java.lang

## Class Thread

java.lang.Object  
java.lang.Thread

**All Implemented Interfaces:**

Runnable

**Direct Known Subclasses:**

ForkJoinWorkerThread

---

```
public class Thread
extends Object
implements Runnable
```

A `thread` is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. Each thread may or may not also be marked as a daemon. When code running in some thread creates a new `Thread` object, the new thread has its priority initially set equal to the priority of the creating thread, and is a daemon thread if and only if the creating thread is a daemon.

Secure Search

McAfee

# Three Java APIs you must know for Multithreading

The screenshot shows a Mozilla Firefox window displaying the Java Platform SE 7 documentation for the `Runnable` interface. The URL in the address bar is `docs.oracle.com/javase/7/docs/api/index.html?java/lang/Runnable.html`. The page title is "Runnable (Java Platform SE 7) - Mozilla Firefox". The left sidebar lists various Java packages, with "java.lang" currently selected. The main content area shows the `Runnable` interface definition and its implementation details.

**Java™ Platform Standard Ed. 7**

All Classes Packages

java.applet  
java.awt

All Classes

AbstractAction  
AbstractAnnotationValueVisitor6  
AbstractAnnotationValueVisitor7  
AbstractBorder  
AbstractButton  
AbstractCellEditor  
AbstractCollection  
AbstractColorChooserPanel  
AbstractDocument  
*AbstractDocument.AttributeConte*  
*AbstractDocument.Content*  
AbstractDocument.ElementEdit  
AbstractElementVisitor6  
AbstractElementVisitor7  
AbstractExecutorService  
AbstractInterruptibleChannel  
AbstractLayoutCache  
AbstractLayoutCache.NodeDimen  
AbstractList  
AbstractListModel

Overview Package **Class** Use Tree Deprecated Index Help

Prev Class Next Class Frames No Frames

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

`java.lang`

## Interface Runnable

**All Known Subinterfaces:**

`RunnableFuture<V>, RunnableScheduledFuture<V>`

**All Known Implementing Classes:**

`AsyncBoxView.ChildState, ForkJoinWorkerThread, FutureTask, RenderableImageProducer, SwingWorker, Thread, TimerTask`

---

**public interface Runnable**

The `Runnable` interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called `run`.

This interface is designed to provide a common protocol for objects that wish to execute code while they are active. For example, `Runnable` is implemented by class `Thread`. Being active simply means that a thread has been started and has not yet been stopped.

In addition, `Runnable` provides the means for a class to be active while not subclassing `Thread`. A class that implements `Runnable` can run without subclassing `Thread` by instantiating a `Thread` instance and passing itself in as the target. In most cases, the `Runnable` interface should be used if you are only planning to override the `run()` method and no other `Thread` methods. This is important because classes should not be subclassed unless the programmer intends on modifying or enhancing the fundamental behavior of the class.

# Three Java APIs you must know for Multithreading

The screenshot shows a Mozilla Firefox browser window displaying the Java API documentation for the `Object` class. The URL in the address bar is `docs.oracle.com/javase/7/docs/api/index.html?java/lang/Object.html`. The browser title is "Object (Java Platform SE 7) - Mozilla Firefox".

The page header includes the Java™ Platform Standard Ed. 7 logo and navigation links: Overview, Package, Class (which is highlighted in orange), Use, Tree, Deprecated, Index, and Help.

The left sidebar lists categories: All Classes, Packages, java.applet, and java.awt. Under All Classes, a scrollable list includes: AbstractAction, AbstractAnnotationValueVisitor6, AbstractAnnotationValueVisitor7, AbstractBorder, AbstractButton, AbstractCellEditor, AbstractCollection, AbstractColorChooserPanel, AbstractDocument, AbstractDocument.AttributeConte, AbstractDocument.Content, AbstractDocument.ElementEdit, AbstractElementVisitor6, AbstractElementVisitor7, AbstractExecutorService, AbstractInterruptibleChannel, AbstractLayoutCache, AbstractLayoutCache.NodeDimen, AbstractList, and AbstractListModel.

The main content area shows the `java.lang` package and the **Class Object**. It provides a brief description: "Class Object is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class." It also indicates that the class was introduced in JDK 1.0 and lists a "See Also" link to the `Class` class.

A "Constructor Summary" section is present at the bottom, with a "Constructors" button.

# And you must know this package in Java: java.util.concurrent

The screenshot shows a Mozilla Firefox browser window displaying the Java API documentation for the `ConcurrentHashMap` class. The URL in the address bar is `docs.oracle.com/javase/7/docs/api/index.html?java/lang/Runnable.html`. The browser interface includes a menu bar (File, Edit, View, History, Bookmarks, Tools, Help), a toolbar with icons for back, forward, search, and refresh, and a status bar at the bottom.

The main content area shows the `java.util.concurrent` package. The `ConcurrentHashMap<K,V>` class is highlighted. The page provides the following details:

- Inheritance:** `java.lang.Object` → `java.util.AbstractMap<K,V>` → `java.util.concurrent.ConcurrentHashMap<K,V>`
- Type Parameters:**
  - `K` - the type of keys maintained by this map
  - `V` - the type of mapped values
- All Implemented Interfaces:**
  - `Serializable, ConcurrentMap<K,V>, Map<K,V>`
- Source Code:**

```
public class ConcurrentHashMap<K,V>
extends AbstractMap<K,V>
implements ConcurrentMap<K,V>, Serializable
```
- Description:**

A hash table supporting full concurrency of retrievals and adjustable expected concurrency for updates. This class obeys the same functional specification as `Hashtable`, and includes versions of methods corresponding to each method of `Hashtable`. However, even though all operations are thread-safe, retrieval operations do *not* entail locking, and there is *not* any support for locking the entire table in a way that

## Concurrent Processes

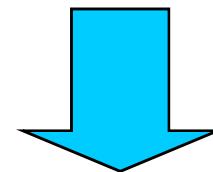
---

We structure complex systems as sets of simpler activities, each represented as a **sequential process**. Processes can overlap or be concurrent, so as to reflect the concurrency inherent in the physical world, or to offload time-consuming tasks, or to manage communications or other devices.

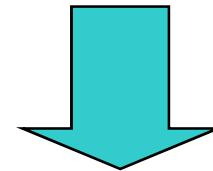
Designing concurrent software can be complex and error prone.

A rigorous engineering approach is essential.

*Concept of a process as a sequence of actions.*



*Model processes as finite state machines.*



*Program processes as threads in Java.*

## Definitions

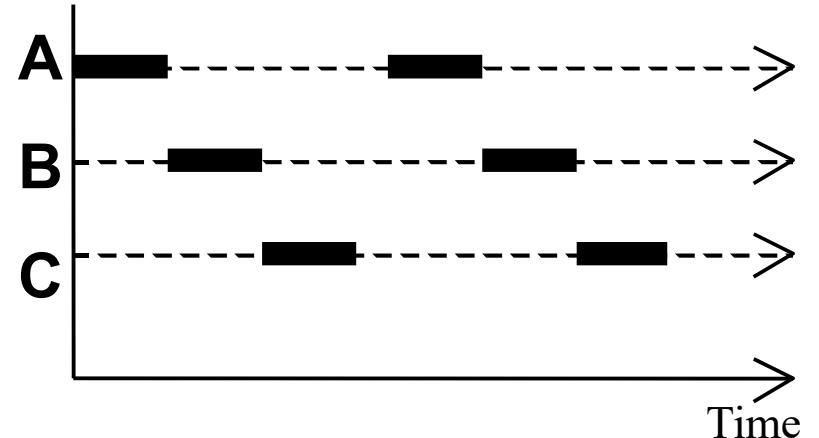
---

- **Concurrency**

- Logically simultaneous processing.
- Does not imply multiple processing elements (PEs). Requires interleaved execution on a single PE.

- **Parallelism**

- Physically simultaneous processing.
- Involves multiple PEs and/or independent device operations.



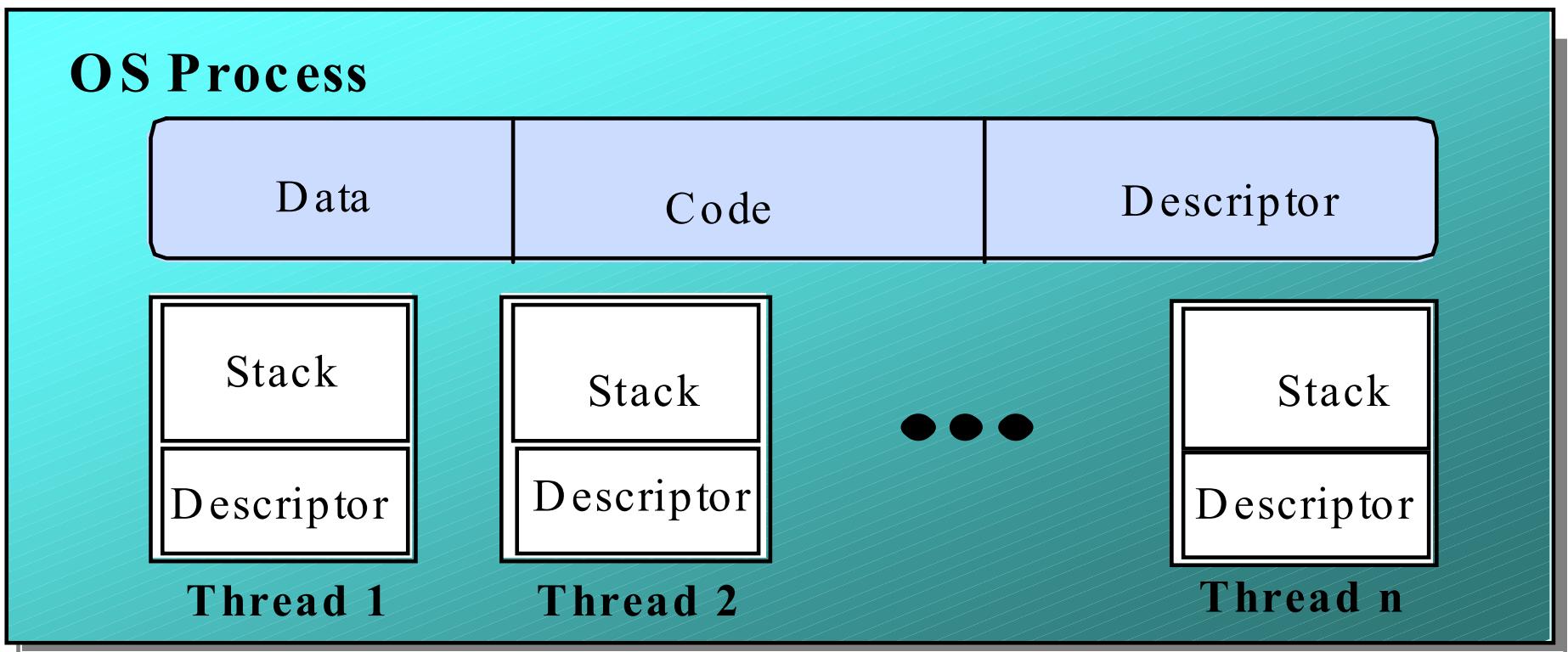
Both concurrency and parallelism require controlled access to shared resources. We use the terms parallel and concurrent interchangeably and generally do not distinguish between real and pseudo-parallel execution.

## Modelling Concurrency

---

- How should we model process execution speed?
  - arbitrary speed
- How do we model concurrency?
  - arbitrary relative order of actions from different processes  
**(interleaving but preservation of each process order )**

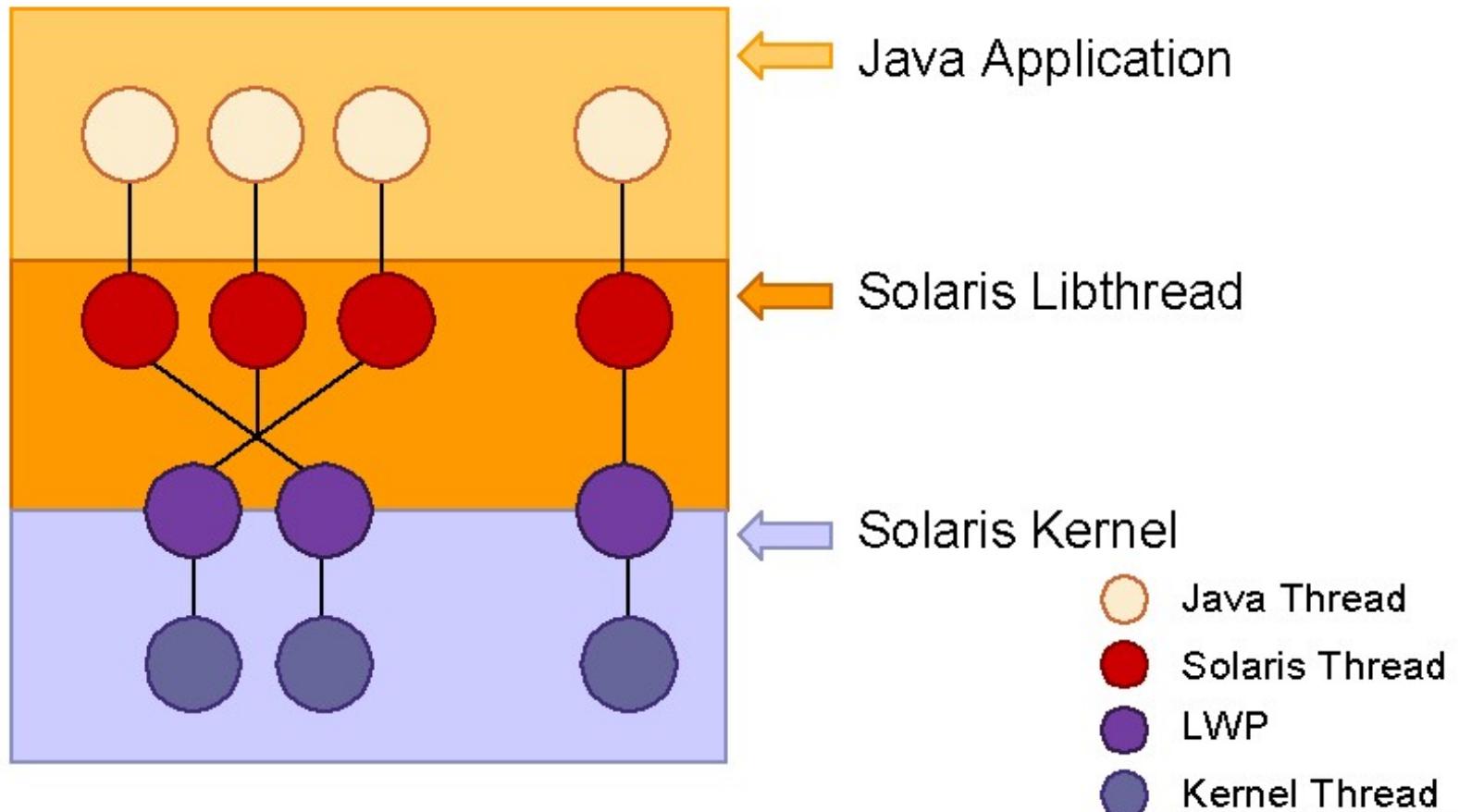
## Implementing processes - the OS view



A (heavyweight) process in an operating system is represented by its code, data and the state of the machine registers, given in a descriptor. In order to support multiple (lightweight) **threads of control**, it has multiple stacks, one for each thread.

# Solaris Threading Models

## Standard N-M Thread Model



- See [http://java.sun.com/docs/books/jvms/second\\_edition/html/Concepts.doc.html#33308](http://java.sun.com/docs/books/jvms/second_edition/html/Concepts.doc.html#33308)
- See the URL <http://www.oracle.com/technetwork/java/threads-140302.html> for some performance measurements for Solaris 7, 8, and 9
- See <http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html#PerformanceTuning>
- See [http://www.oracle.com/technetwork/java/hotspotfaq-138619.html#threads\\_general](http://www.oracle.com/technetwork/java/hotspotfaq-138619.html#threads_general)

# Threads and Synchronization in Java

---

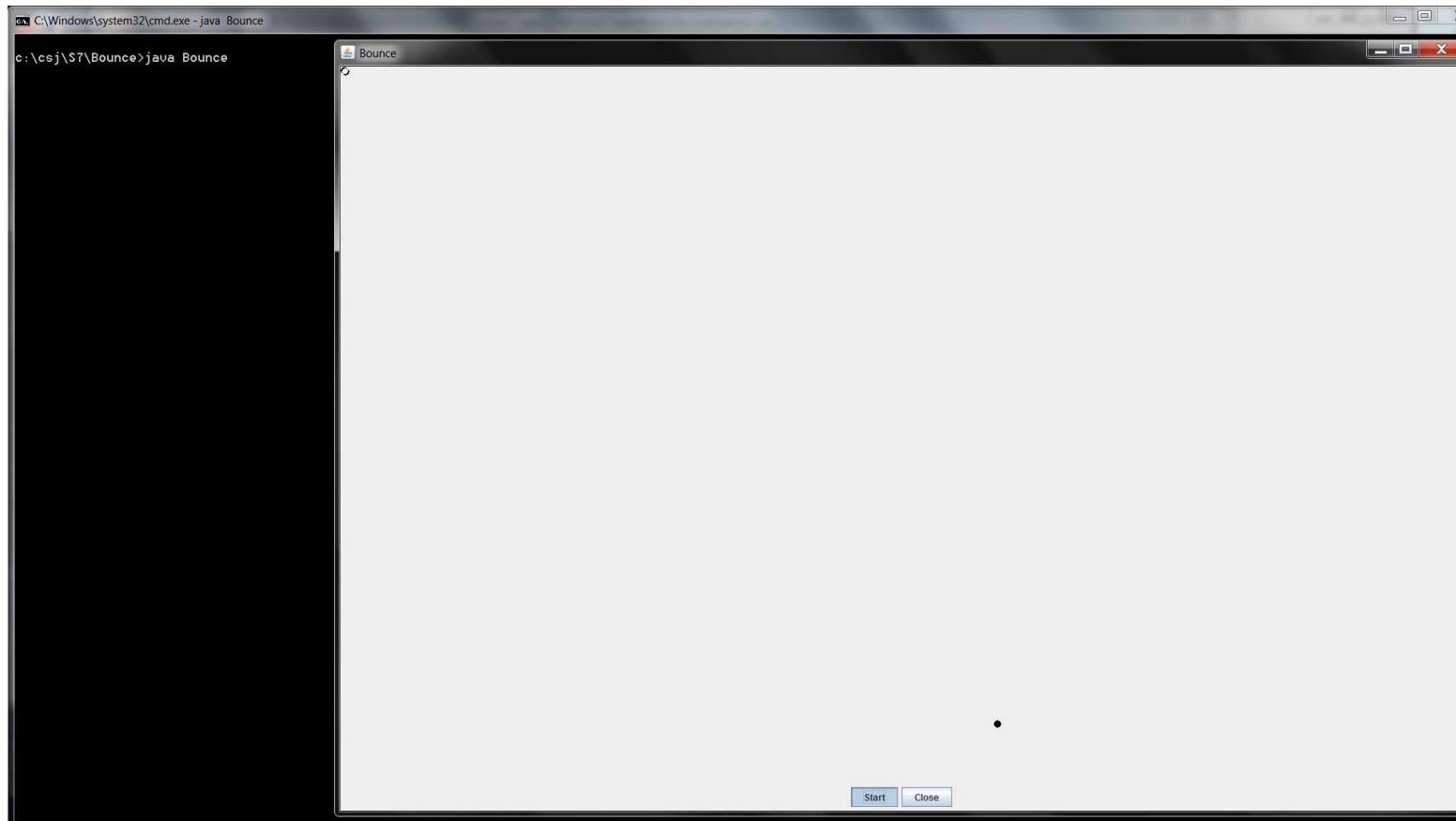
## ◆ Benefits of using threads in Java

- Can improve the performance of complex applications.
- Useful in GUI applications for improving the responsiveness of the user interface
- Improve resource utilization and throughput.
  - ❖ Simplified handling of asynchronous events - If a program is single-threaded, the processor remains idle while it waits for a synchronous I/O operation to complete.
  - ❖ A server application that accepts socket connections from multiple remote clients may be easier to develop when each connection is allocated its own thread and allowed to use synchronous I/O.
- Simplify the implementation of the JVM—the garbage usually runs in one or more dedicated threads.

# Threads and Synchronization in Java

---

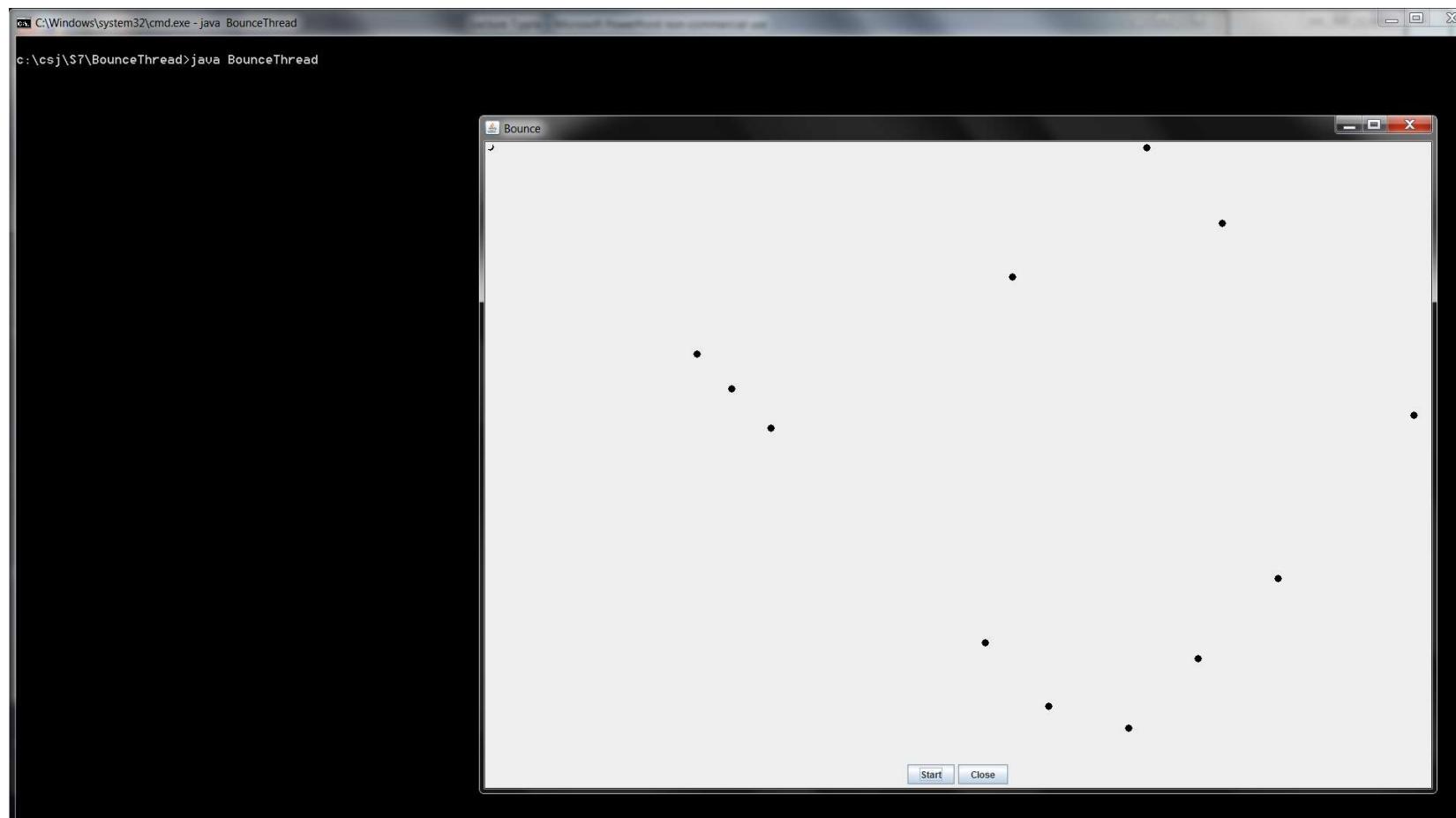
- ◆ What is wrong with the following application (Bouncing Ball)?
- ◆ What is wrong with the start button (new ball)?



# Threads and Synchronization in Java

---

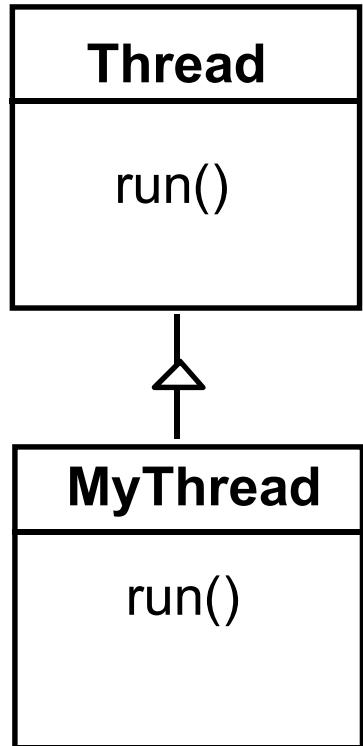
- ◆ Better application for the Bouncing Ball
- ◆ What has changed?



## Threads in Java

---

A Thread class manages a single sequential thread of control.  
Threads may be created and deleted dynamically.



The Thread class executes instructions from its method run(). The actual code executed depends on the implementation provided for run() in a derived class.

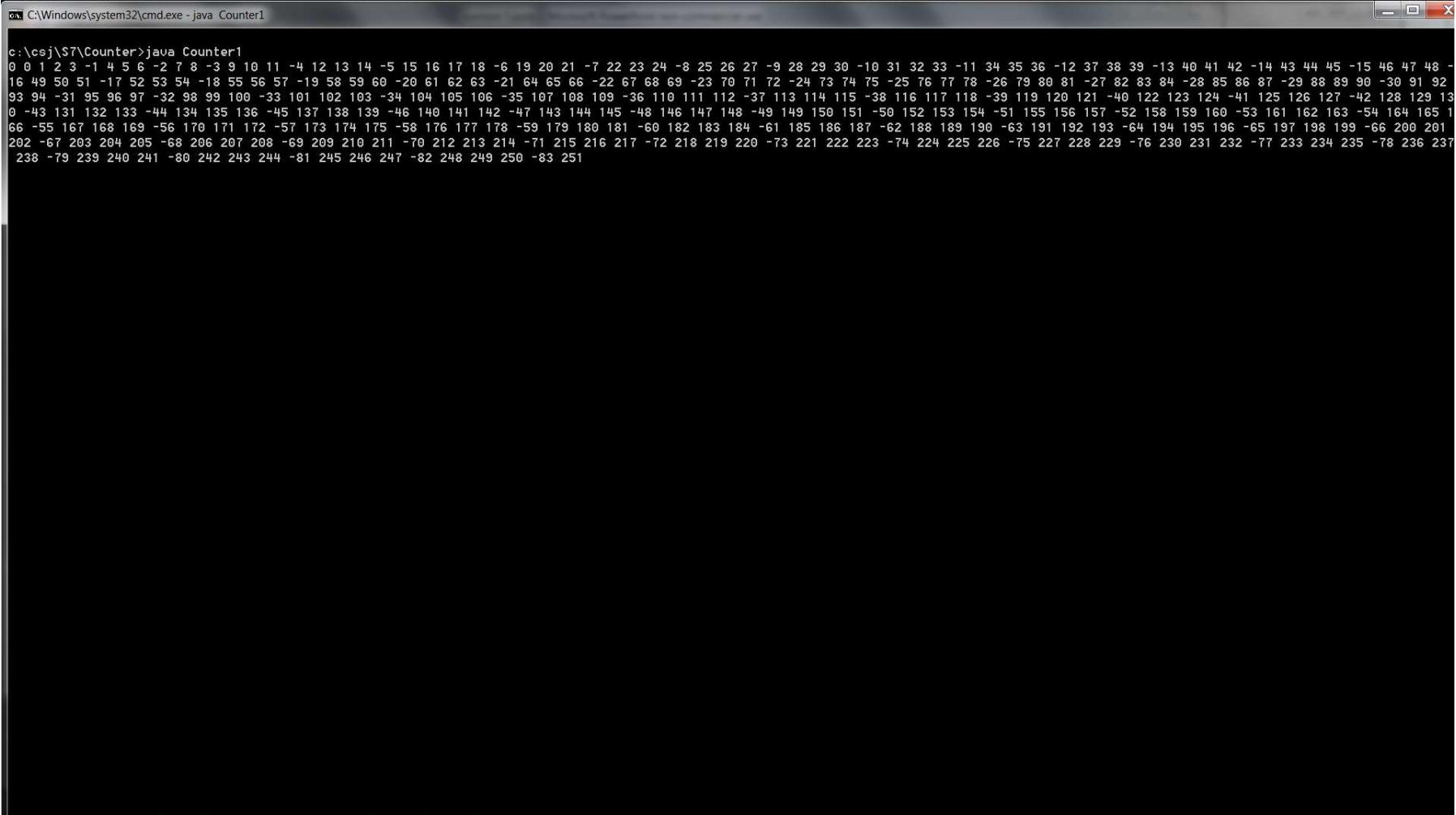
```
class MyThread extends Thread {  
    public void run() {  
        //.....  
    }  
}
```

**Thread x = new MyThread();**

# Threads in Java

---

An example extending the Thread class -- Counter1.java



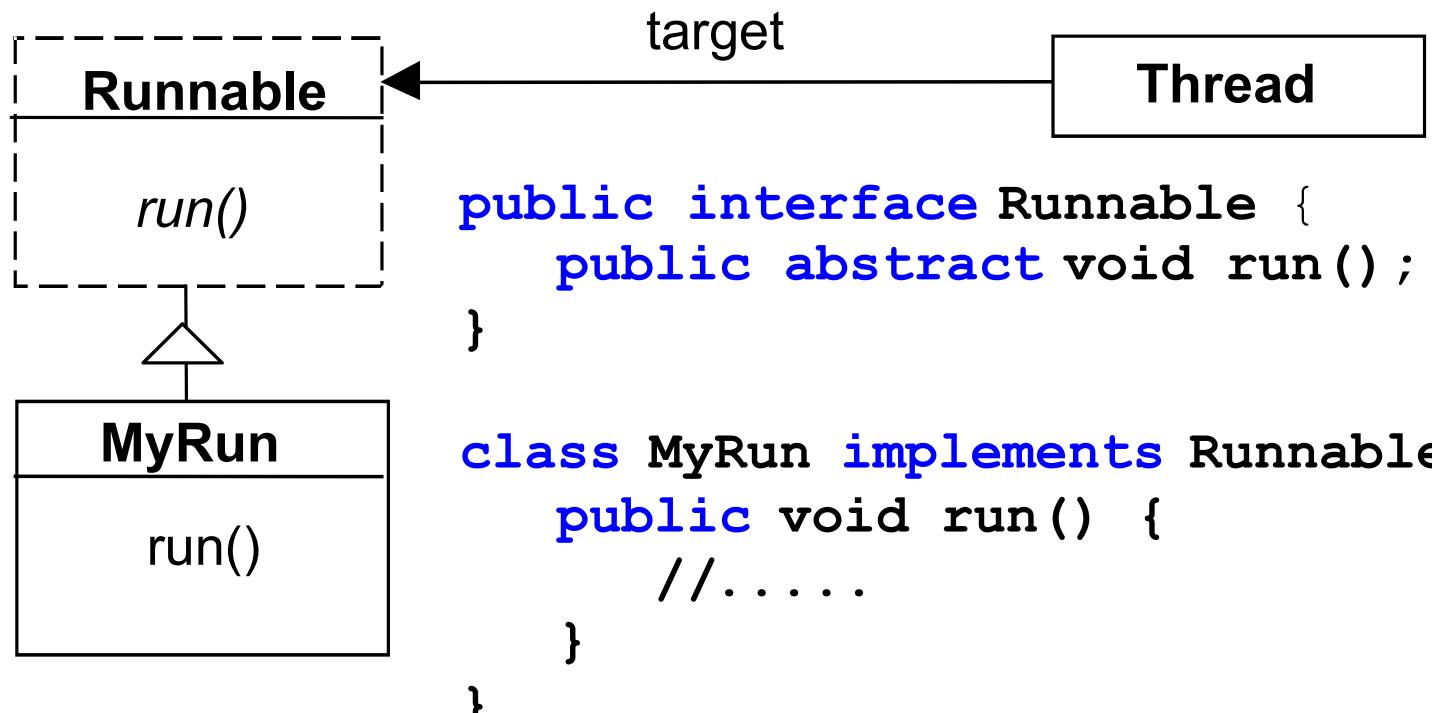
The screenshot shows a Windows command prompt window titled "C:\Windows\system32\cmd.exe - java Counter1". The window contains the output of a Java application named Counter1. The output consists of a long sequence of integers, each on a new line, ranging from -238 to 237. The numbers are generated by two threads running simultaneously, as evidenced by the interleaved nature of the output. Some numbers appear multiple times, such as 198 appearing twice.

```
c:\csj\$7\Counter>java Counter1
0 0 1 2 3 -1 4 5 6 -2 7 8 -3 9 10 11 -4 12 13 14 -5 15 16 17 18 -6 19 20 21 -7 22 23 24 -8 25 26 27 -9 28 29 30 -10 31 32 33 -11 34 35 36 -12 37 38 39 -13 40 41 42 -14 43 44 45 -15 46 47 48 -
16 49 50 51 -17 52 53 54 -18 55 56 57 -19 58 59 60 -20 61 62 63 -21 64 65 66 -22 67 68 69 -23 70 71 72 -24 73 74 75 -25 76 77 78 -26 79 80 81 -27 82 83 84 -28 85 86 87 -29 88 89 90 -30 91 92
93 94 -31 95 96 97 -32 98 99 100 -33 101 102 103 -34 104 105 106 -35 107 108 109 -36 110 111 112 -37 113 114 115 -38 116 117 118 -39 119 120 121 -40 122 123 124 -41 125 126 127 -42 128 129 13
0 -43 131 132 133 -44 134 135 136 -45 137 138 139 -46 140 141 142 -47 143 144 145 -48 146 147 148 -49 149 150 151 -50 152 153 154 -51 155 156 157 -52 158 159 160 -53 161 162 163 -54 164 165 1
66 -55 167 168 169 -56 170 171 172 -57 173 174 175 -58 176 177 178 -59 179 180 181 -60 182 183 184 -61 185 186 187 -62 188 189 190 -63 191 192 193 -64 194 195 196 -65 197 198 199 -66 200 201
202 -67 203 204 205 -68 206 207 208 -69 209 210 211 -70 212 213 214 -71 215 216 217 -72 218 219 220 -73 221 222 223 -74 224 225 226 -75 227 228 229 -76 230 231 232 -77 233 234 235 -78 236 237
238 -79 239 240 241 -80 242 243 244 -81 245 246 247 -82 248 249 250 -83 251
```

## Threads in Java

---

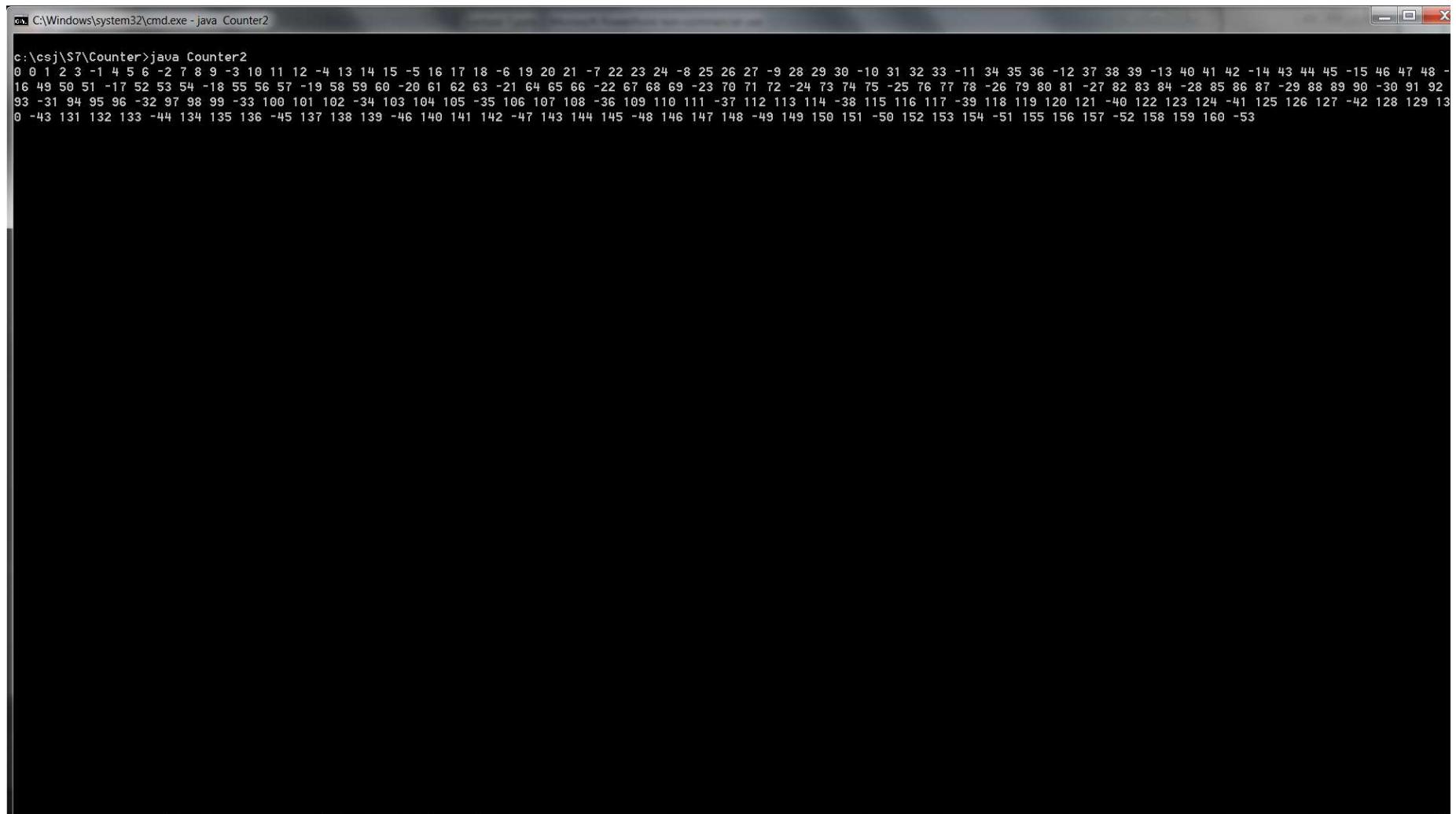
Since Java does not permit multiple inheritance, we often implement the `run()` method in a class not derived from `Thread` but from the interface `Runnable`.



# Threads in Java

---

An example implementing the Runnable Interface – Counter2.java



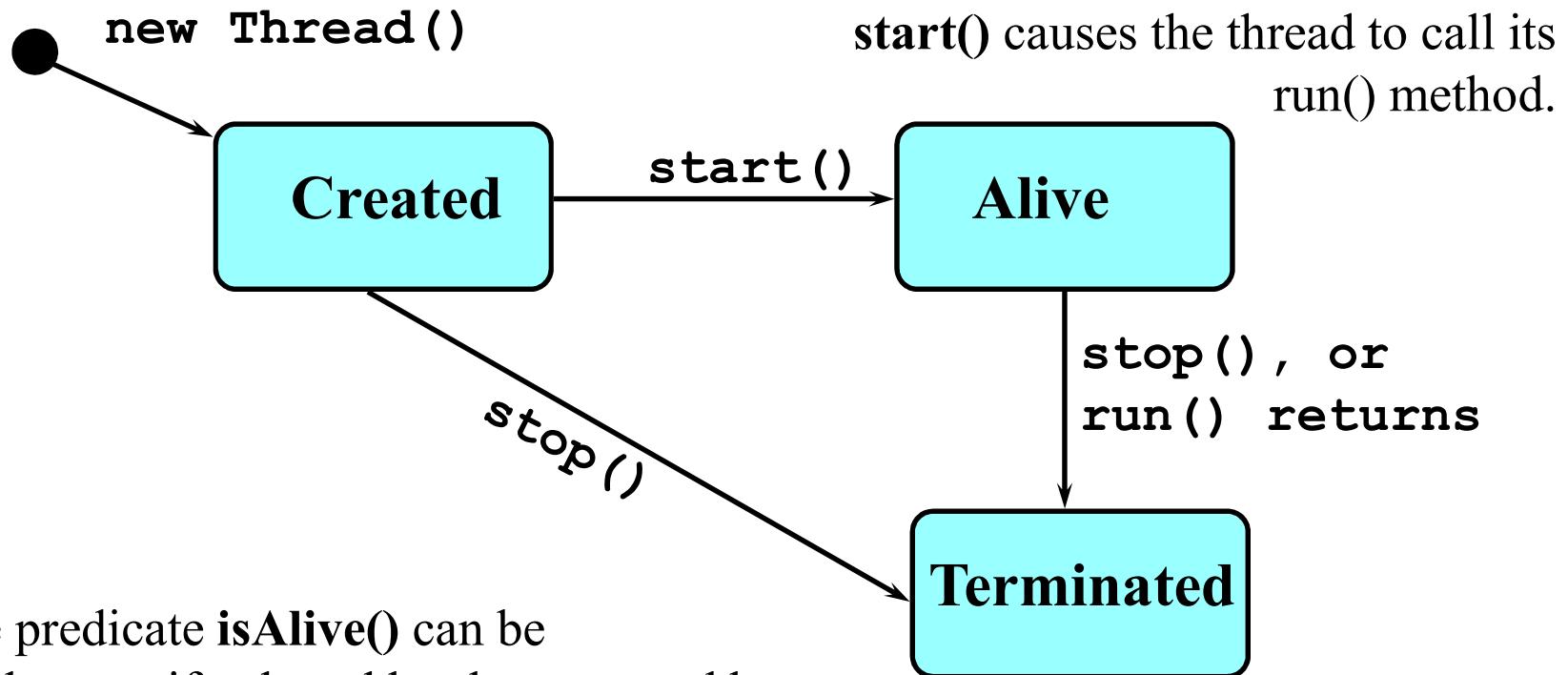
The screenshot shows a Windows command prompt window titled "C:\Windows\system32\cmd.exe - java Counter2". The window displays the output of a Java application named Counter2. The output consists of a long list of integers, both positive and negative, separated by spaces. The numbers range from -53 to 138, with many values between 0 and 100. The list is approximately 100 lines long.

```
c:\csj\$7\Counter>java Counter2
0 0 1 2 3 -1 4 5 6 -2 7 8 9 -3 10 11 12 -4 13 14 15 -5 16 17 18 -6 19 20 21 -7 22 23 24 -8 25 26 27 -9 28 29 30 -10 31 32 33 -11 34 35 36 -12 37 38 39 -13 40 41 42 -14 43 44 45 -15 46 47 48 -16 49 50 51 -17 52 53 54 -18 55 56 57 -19 58 59 60 -20 61 62 63 -21 64 65 66 -22 67 68 69 -23 70 71 72 -24 73 74 75 -25 76 77 78 -26 79 80 81 -27 82 83 84 -28 85 86 87 -29 88 89 90 -30 91 92 93 -31 94 95 96 -32 97 98 99 -33 100 101 102 -34 103 104 105 -35 106 107 108 -36 109 110 111 -37 112 113 114 -38 115 116 117 -39 118 119 120 121 -40 122 123 124 -41 125 126 127 -42 128 129 130 -43 131 132 133 -44 134 135 136 -45 137 138 139 -46 140 141 142 -47 143 144 145 -48 146 147 148 -49 149 150 151 -50 152 153 154 -51 155 156 157 -52 158 159 160 -53
```

## thread life-cycle in Java

---

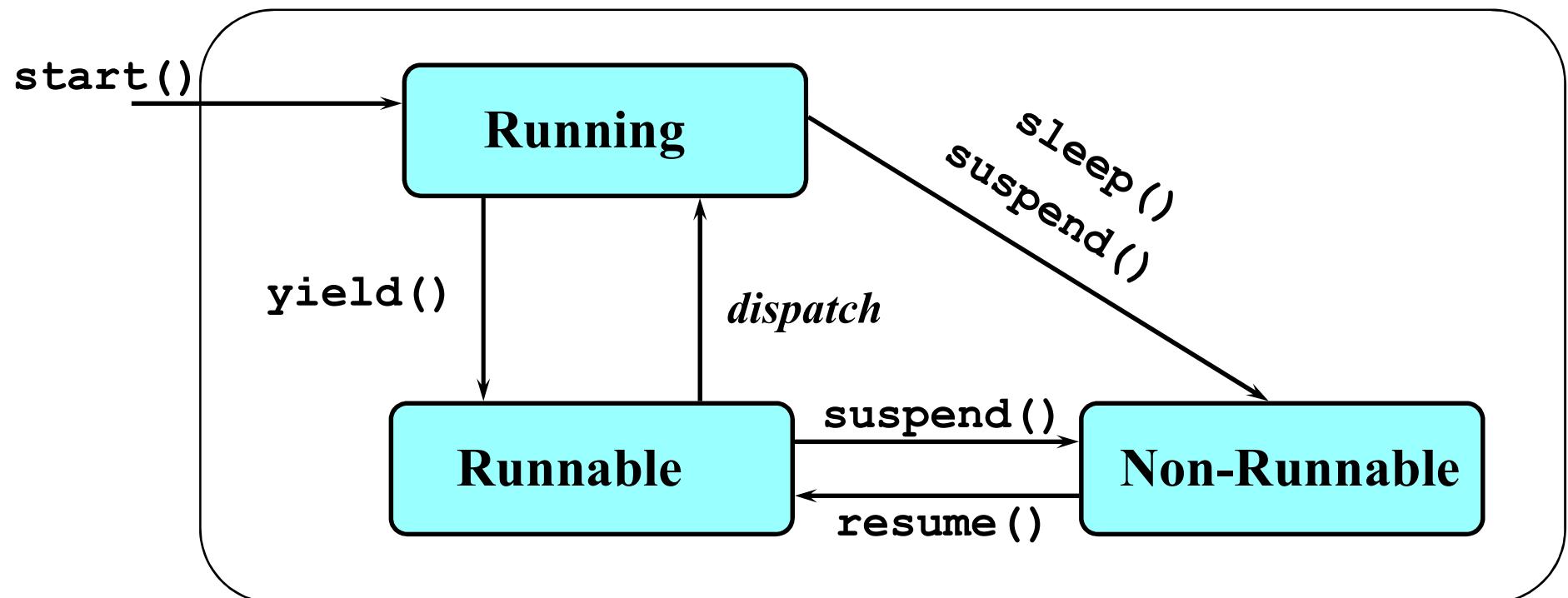
An overview of the life-cycle of a thread as state transitions:



The predicate `isAlive()` can be used to test if a thread has been started but not terminated. Once terminated, it cannot be restarted.

## Thread alive states in Java

Once started, an **alive** thread has a number of substates :



`wait()` also makes a Thread Non-Runnable, and  
`notify()` Runnable

`stop()`, or  
`run()` returns

## Java thread lifecycle - an FSP specification

---

|              |   |   |  |
|--------------|---|---|--|
| THREAD       | = | CREATED ,   |  |
| CREATED      | = | (start<br>  stop<br>= ({suspend,sleep})<br>  yield<br>  {stop,end}<br>  run | ->RUNNING<br>->TERMINATED) ,<br>->NON_RUNNABLE<br>->RUNNABLE<br>->TERMINATED<br>->RUNNING) , |
| RUNNING      | = | (suspend<br>  dispatch<br>  stop  | ->NON_RUNNABLE<br>->RUNNING<br>->TERMINATED) ,   |
| RUNNABLE     | = | (resume<br>  stop   | ->RUNNABLE<br>->TERMINATED) ,  |
| NON_RUNNABLE | = | (resume<br>  stop   | ->RUNNABLE<br>->TERMINATED) ,  |
| TERMINATED   | = | STOP .  |  |

## Thread Safety

---



## Thread Safety

---

- ◆ Thread-safe code is about managing access to a *state variable*, and in particular to a *shared mutable state variable*
  - By *shared*, we mean that a variable could be accessed by multiple threads
  - By *mutable*, we mean that its value could change during its lifetime
- ◆ The object's *state* is its data, stored in *state variables* such as instance or static fields.
- ◆ Thread safety is needed to protect *data* from uncontrolled concurrent access
- ◆ Making an object thread-safe requires using synchronization to coordinate access to its mutable state variables

## Thread Safety

---

- ◆ Whenever more than one thread accesses a given state variable, and one of them might write to it, they all must coordinate their access to it using synchronization
- ◆ The primary mechanism for synchronization in Java is the synchronized keyword, which provides exclusive locking, but the term "synchronization" also includes the use of volatile variables, explicit locks, and atomic variables.
- ◆ *It is much easier to design a class to be thread-safe than to retrofit it for thread safety later*

## Thread Safety

---

- ◆ For large application, identifying whether multiple threads might access a given variable is not easy.
- ◆ However, the same object-oriented techniques that help you write well-organized, maintainable classes—such as encapsulation and data hiding—can also help you create thread-safe classes.
- ◆ The less code that has access to a particular variable, the easier it is to reason about the conditions under which a given variable might be accessed.
- ◆ The Java language doesn't force you to encapsulate your state variables, but the better encapsulated your state variables, the easier it is to make your program thread-safe

## Thread Safety

---

If multiple threads access the same mutable state variable without appropriate synchronization, *your program is broken*. There are three ways to fix it:

- *Don't share* the state variable across threads;
- Make the state variable *immutable*; or
- Use *synchronization* whenever accessing the state variable.

Stateless objects are always thread-safe.

When designing thread-safe classes, good object-oriented techniques—encapsulation, immutability, and clear specification of invariants—are your best friends.

## Thread Safety

---

- ◆ The definition of thread safety is correlated to the concept of correctness.
- ◆ Correctness means that a class *conforms to its specification*.
- ◆ A good specification defines *invariants* constraining an object's state and *postconditions* describing the effects of its operations.

A class is *thread-safe* if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code.

Thread-safe classes encapsulate any needed synchronization so that clients need not provide their own.

## Thread Safety

---

- ◆ Why the following servlet is stateless?
  - Do we have state variables for the servlet object?

---

```
@ThreadSafe
public class StatelessFactorizer implements Servlet {
    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        encodeIntoResponse(resp, factors);
    }
}
```

---

## Thread Safety

---

- ◆ Why the following servlet is stateful?
  - Do we have state variables for the servlet object?
  - Is it thread-safe?

```
@NotThreadSafe
public class UnsafeCountingFactorizer implements Servlet {
    private long count = 0;

    public long getCount() { return count; }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        ++count;
        encodeIntoResponse(resp, factors);
    }
}
```

## Thread Safety

---

- ◆ The previous `UnsafeCountingFactorizer` servlet is not thread-safe, even though it might work just fine in a single-threaded environment
- ◆ The increment operation, `++count`, may look like a single action because of its compact syntax, it is not *atomic*, it is a shorthand for a sequence of three discrete operations: fetch the current value, add one to it, and write the new value back.
- ◆ This is an example of a *read-modify-write* operation, in which the resulting state is derived from the previous state.
- ◆ `UnsafeCountingFactorizer` has several ***race conditions*** that make its results unreliable.
- ◆ A ***race condition*** occurs when the correctness of a computation depends on the relative timing or interleaving of multiple threads by the runtime; in other words, when getting the right answer relies on lucky timing

# Thread Safety

---

## ◆ Race conditions in lazy initialization

- A common idiom that uses check-then-act is *lazy initialization*.
- The goal of lazy initialization is to defer initializing an object until it is actually needed while at the same time ensuring that it is initialized only once
- The getInstance method first checks the existing of ExpensiveObject instance; otherwise it creates a new instance and returns it after retaining a reference to it so that future invocations can avoid the expensive creation code.

---

```
@NotThreadSafe
public class LazyInitRace {
    private ExpensiveObject instance = null;

    public ExpensiveObject getInstance() {
        if (instance == null)
            instance = new ExpensiveObject();
        return instance;
    }
}
```

---

## Thread Safety

---

### ◆ Race conditions in lazy initialization

- LazyInitRace has race conditions that can impact its correctness in a negative way.
- Assume threads *A* and *B* execute getInstance at the same time, *A* sees that instance is null, and instantiates a new ExpensiveObject.
- *B* also checks if instance is null. Whether instance is null at this point depends unpredictably on timing, including the vagaries of scheduling and how long *A* takes to instantiate the ExpensiveObject and set the instance field.
- If instance is null when *B* examines it, the two callers to getInstance may receive two different results, even though getInstance is always supposed to return the same instance.

### ◆ Like most concurrency errors, race conditions don't always result in failure: some unlucky timing is also required.

## Thread Safety

---

- ◆ To avoid race conditions, we must prevent other threads from using a variable while we're in the middle of modifying it, so we can ensure that other threads can observe or modify the state only before we start or after we finish, but not in the middle.

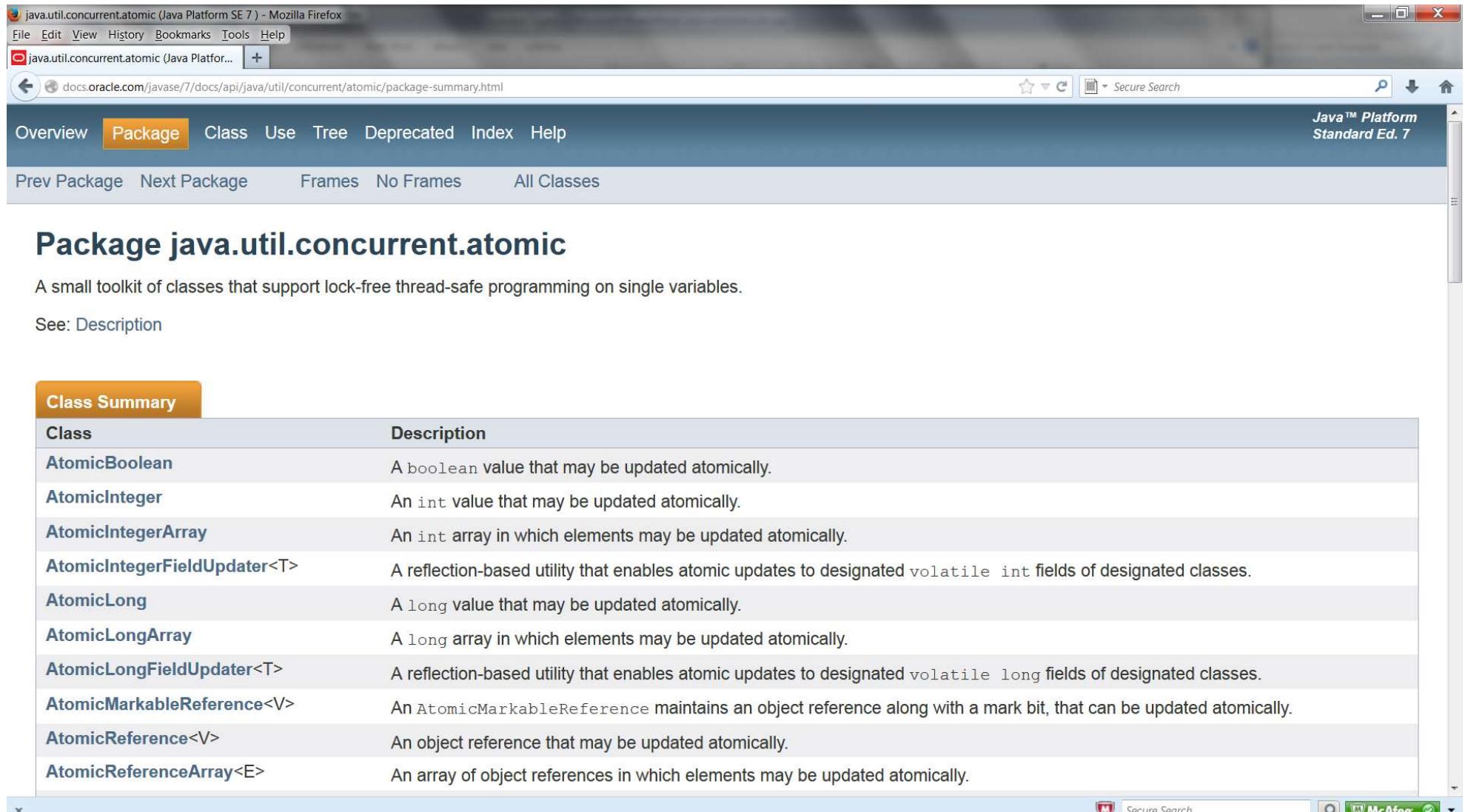
Operations *A* and *B* are *atomic* with respect to each other if, from the perspective of a thread executing *A*, when another thread executes *B*, either all of *B* has executed or none of it has. An *atomic operation* is one that is atomic with respect to all operations, including itself, that operate on the same state.

## Thread Safety

---

- ◆ To avoid race conditions, we must prevent other threads from using a variable while we're in the middle of modifying it, so we can ensure that other threads can observe or modify the state only before we start or after we finish, but not in the middle.
- ◆ To ensure thread safety, **check-then-act** operations (like lazy initialization) and **read-modify-write** operations (like increment) must always be atomic.
- ◆ We refer to **check-then-act** and **read-modify-write** sequences as **compound actions**: sequences of operations that **must be executed atomically in order to remain thread-safe**.
- ◆ Later we'll consider *locking*, Java's built-in mechanism for ensuring atomicity, but for now we will use `java.util.concurrent.atomic` to fix the problem in the previous example

# Thread Safety



The screenshot shows a Mozilla Firefox browser window with the title "java.util.concurrent.atomic (Java Platform SE 7) - Mozilla Firefox". The address bar contains "docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/package-summary.html". The page content is the Java™ Platform Standard Ed. 7 documentation for the java.util.concurrent.atomic package. The "Package" tab is selected in the navigation bar. The main content area displays the package summary, class descriptions, and a table of atomic classes.

**Package java.util.concurrent.atomic**

A small toolkit of classes that support lock-free thread-safe programming on single variables.

See: Description

| Class Summary                                      |   |
|--|---|
| Class  | Description   |
| <a href="#">AtomicBoolean</a>                      | A <code>boolean</code> value that may be updated atomically.  |
| <a href="#">AtomicInteger</a>                      | An <code>int</code> value that may be updated atomically.   |
| <a href="#">AtomicIntegerArray</a>                 | An <code>int</code> array in which elements may be updated atomically.  |
| <a href="#">AtomicIntegerFieldUpdater&lt;T&gt;</a> | A reflection-based utility that enables atomic updates to designated <code>volatile int</code> fields of designated classes.  |
| <a href="#">AtomicLong</a>                         | A <code>long</code> value that may be updated atomically.   |
| <a href="#">AtomicLongArray</a>                    | A <code>long</code> array in which elements may be updated atomically.  |
| <a href="#">AtomicLongFieldUpdater&lt;T&gt;</a>    | A reflection-based utility that enables atomic updates to designated <code>volatile long</code> fields of designated classes. |
| <a href="#">AtomicMarkableReference&lt;V&gt;</a>   | An <code>AtomicMarkableReference</code> maintains an object reference along with a mark bit, that can be updated atomically.  |
| <a href="#">AtomicReference&lt;V&gt;</a>           | An object reference that may be updated atomically.   |
| <a href="#">AtomicReferenceArray&lt;E&gt;</a>      | An array of object references in which elements may be updated atomically.  |

# Thread Safety

The screenshot shows a Mozilla Firefox browser window displaying the Java Platform Standard Edition 7 documentation for the `AtomicLong` class. The URL in the address bar is `docs.oracle.com/javase/7/docs/api/index.html?java/util/concurrent/atomic/package-summary.html`. The page title is "AtomicLong (Java Platform SE 7) - Mozilla Firefox". The main content area shows the `Class` tab selected, with the package name `java.util.concurrent.atomic` and the class name `AtomicLong`. It lists the `Object`, `Number`, and `AtomicLong` interfaces implemented by `AtomicLong`. The `All Implemented Interfaces:` section includes `Serializable`. Below this, the class definition is shown as:

```
public class AtomicLong
extends Number
implements Serializable
```

The `Since:` section indicates that the class was introduced in version 1.5.

**Java™ Platform Standard Ed. 7**

Overview Package Class Use Tree Deprecated Index Help

Prev Class Next Class Frames No Frames

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.util.concurrent.atomic

## Class AtomicLong

java.lang.Object  
    java.lang.Number  
        java.util.concurrent.atomic.AtomicLong

All Implemented Interfaces:

    Serializable

---

```
public class AtomicLong
extends Number
implements Serializable
```

A long value that may be updated atomically. See the `java.util.concurrent.atomic` package specification for description of the properties of atomic variables. An `AtomicLong` is used in applications such as atomically incremented sequence numbers, and cannot be used as a replacement for a `Long`. However, this class does extend `Number` to allow uniform access by tools and utilities that deal with numerically-based classes.

Since:

1.5

# Thread Safety

The screenshot shows a Mozilla Firefox window displaying the Java API documentation for the `java.util.concurrent.atomic.AtomicLong` class. The URL in the address bar is `docs.oracle.com/javase/7/docs/api/index.html?java/util/concurrent/atomic/package-summary.html`. The left sidebar lists packages like `java.applet` and `java.awt`, and a detailed list of all classes. The main content area is titled "Methods" and lists various atomic operations. One method, `incrementAndGet()`, is highlighted with a yellow box.

| Modifier and Type | Method and Description   |
|-------------------|--|
| long              | <code>addAndGet(long delta)</code><br>Atomically adds the given value to the current value.  |
| boolean           | <code>compareAndSet(long expect, long update)</code><br>Atomically sets the value to the given updated value if the current value == the expected value. |
| long              | <code>decrementAndGet()</code><br>Atomically decrements by one the current value.  |
| double            | <code>doubleValue()</code><br>Returns the value of the specified number as a <code>double</code> .   |
| float             | <code>floatValue()</code><br>Returns the value of the specified number as a <code>float</code> .   |
| long              | <code>get()</code><br>Gets the current value.  |
| long              | <code>getAndAdd(long delta)</code><br>Atomically adds the given value to the current value.  |
| long              | <code>getAndDecrement()</code><br>Atomically decrements by one the current value.  |
| long              | <code>getAndIncrement()</code><br>Atomically increments by one the current value.  |
| long              | <code>getAndSet(long newValue)</code><br>Atomically sets to the given value and returns the old value.   |
| long              | <code>incrementAndGet()</code><br>Atomically increments by one the current value.  |
| int               | <code>intValue()</code><br>Returns the value of the specified number as an <code>int</code> .  |

## Thread Safety

---

- ◆ The `java.util.concurrent.atomic` package contains atomic variable classes for effecting atomic state transitions on numbers and object references.
- ◆ By replacing the long counter with an `AtomicLong`, we ensure that all actions that access the counter state are atomic. Because the state of the servlet is the state of the counter and the counter is thread-safe, our servlet is once again thread-safe.

---

```
@ThreadSafe
public class CountingFactorizer implements Servlet {
    private final AtomicLong count = new AtomicLong(0);

    public long getCount() { return count.get(); }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        count.incrementAndGet();
        encodeIntoResponse(resp, factors);
    }
}
```

---

## Thread Safety

---

Where practical, use existing thread-safe objects, like `AtomicLong`, to manage your class's state. It is simpler to reason about the possible states and state transitions for existing thread-safe objects than it is for arbitrary state variables, and this makes it easier to maintain and verify thread safety.

## Thread Safety

---

- ◆ The definition of thread safety requires that invariants be preserved regardless of timing or interleaving of operations in multiple threads.
- ◆ When multiple variables participate in an invariant, they are not independent: the value of one constrains the allowed value(s) of the others. Thus when updating one, you must update the others *in the same atomic operation*.
- ◆ Using atomic references, we cannot update two state variables simultaneously, even though each call to set is atomic; there is still a window of vulnerability when one has been modified and the other has not, and during that time other threads could see that the invariant does not hold.
- ◆ Similarly, the two values cannot be fetched simultaneously: between the time when thread A fetches the two values, thread B could have changed them, and again A may observe that the invariant does not hold.

## Thread Safety

---

- ◆ So, what is the solution to update/access multiple atomic variables?
  - Java provides a built-in locking mechanism for enforcing atomicity: the synchronized block.
  - A synchronized block has two parts: a reference to an object that will serve as the *lock*, and a block of code to be guarded by that lock.

To preserve state consistency, update related state variables in a single atomic operation.

## Thread Safety

---

- ◆ A synchronized method is a shorthand for a synchronized block that spans an entire method body, and whose lock is the object on which the method is being invoked.

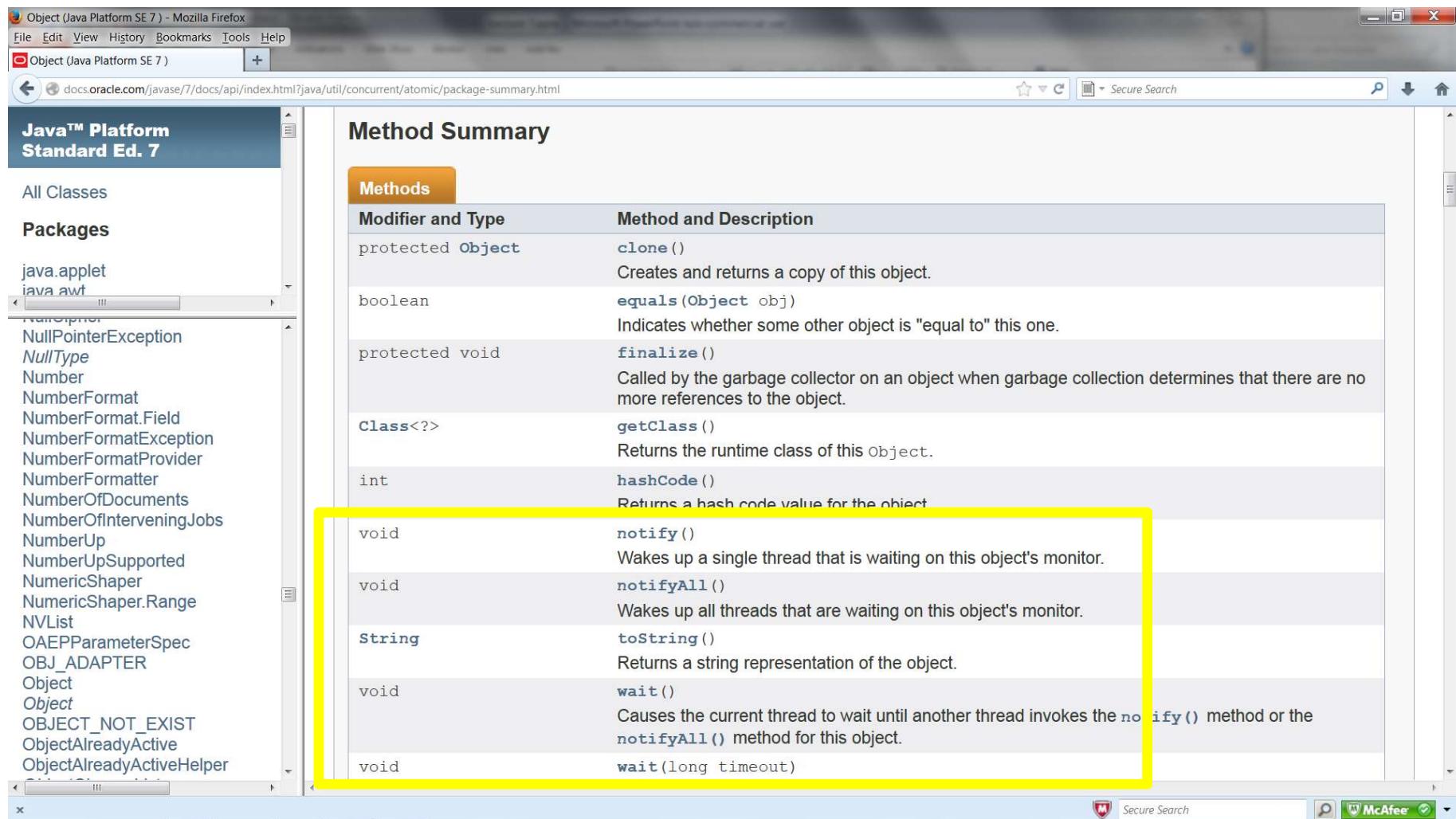
```
synchronized (lock) {  
    // Access or modify shared state guarded by lock  
}
```

- ◆ Every Java object can implicitly act as a lock for purposes of synchronization
- ◆ These built-in locks are called *intrinsic locks* or *monitor locks*.
- ◆ The lock is automatically acquired by the executing thread before entering a synchronized block and automatically released when control exits the synchronized block

# Thread Safety

The screenshot shows a Mozilla Firefox browser window displaying the Java API documentation for the `Object` class. The URL in the address bar is `docs.oracle.com/javase/7/docs/api/index.html?java/util/concurrent/atomic/package-summary.html`. The browser title bar reads "Object (Java Platform SE 7) - Mozilla Firefox". The page header includes the "Java™ Platform Standard Ed. 7" logo and navigation links for Overview, Package, Class (which is highlighted in orange), Use, Tree, Deprecated, Index, and Help. Below the header are links for Prev Class, Next Class, Frames, and No Frames, along with summary and detail links for Nested, Field, Constr, and Method. The main content area starts with the package name `java.lang` and the class name `Class Object`. It then lists the class `java.lang.Object` with its definition: `public class Object`. A note states: "Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class." Below this are sections for "Since:" (JDK1.0) and "See Also:" (Class). At the bottom is a "Constructor Summary" section with a "Constructors" button.

# Thread Safety



The screenshot shows a Mozilla Firefox window displaying the Java API documentation for the `Object` class. The URL in the address bar is `docs.oracle.com/javase/7/docs/api/index.html?java/util/concurrent/atomic/package-summary.html`. The left sidebar lists packages and classes, with `java.util` expanded to show various classes like `ArrayList`, `HashMap`, etc. The main content area is titled "Method Summary" and contains a table of methods. A yellow box highlights the `notify()` and `notifyAll()` methods.

| Modifier and Type             | Method and Description   |
|-------------------------------|--|
| <code>protected Object</code> | <code>clone()</code><br>Creates and returns a copy of this object.   |
| <code>boolean</code>          | <code>equals(Object obj)</code><br>Indicates whether some other object is "equal to" this one.   |
| <code>protected void</code>   | <code>finalize()</code><br>Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.                    |
| <code>Class&lt;?&gt;</code>   | <code>getClass()</code><br>Returns the runtime class of this Object.   |
| <code>int</code>              | <code>hashCode()</code><br>Returns a hash code value for the object.   |
| <code>void</code>             | <code>notify()</code><br>Wakes up a single thread that is waiting on this object's monitor.  |
| <code>void</code>             | <code>notifyAll()</code><br>Wakes up all threads that are waiting on this object's monitor.  |
| <code>String</code>           | <code>toString()</code><br>Returns a string representation of the object.  |
| <code>void</code>             | <code>wait()</code><br>Causes the current thread to wait until another thread invokes the <code>notify()</code> method or the <code>notifyAll()</code> method for this object. |
| <code>void</code>             | <code>wait(long timeout)</code>  |

## Thread Safety

---

- ◆ The only way to acquire an intrinsic lock is to enter a synchronized block or method guarded by that lock.
- ◆ Intrinsic locks in Java act as mutexes (or *mutual exclusion locks*), which means that at most one thread may own the lock. When thread *A* attempts to acquire a lock held by thread *B*, *A* must wait, or *block*, until *B* releases it. If *B* never releases the lock, *A* waits forever.
- ◆ Since only one thread at a time can execute a block of code guarded by a given lock, the synchronized blocks guarded by the same lock execute atomically with respect to one another

# Thread Safety

---

## ◆ Reentrancy

- If a thread requests a lock that is already held by another thread, the requesting thread blocks.
- Since intrinsic locks are *reentrant*, if a thread tries to acquire a lock that it already holds, the request succeeds.
- Reentrancy means that locks are acquired on a per-thread rather than per-invocation basis.
- Reentrancy is implemented by associating with each lock an acquisition count and an owning thread. When the count is zero, the lock is considered unheld.
- Reentrancy facilitates encapsulation of locking behavior, and thus simplifies the development of object-oriented concurrent code

## Thread Safety

---

### ◆ Guarding state with locks

- Compound actions on shared state, such as incrementing a hit counter (read-modify-write) or lazy initialization (check-then-act), must be made atomic to avoid race conditions.
- Holding a lock for the *entire duration* of a compound action can make that compound action atomic.
- Just wrapping the compound action with a synchronized block is not sufficient; if synchronization is used to coordinate access to a variable, it is needed *everywhere* that variable is accessed.
- When using locks to coordinate access to a variable, the *same* lock must be used wherever that variable is accessed

## Thread Safety

---

For each mutable state variable that may be accessed by more than one thread, *all* accesses to that variable must be performed with the *same* lock held. In this case, we say that the variable is *guarded by* that lock.

Every shared, mutable variable should be guarded by exactly one lock. Make it clear to maintainers which lock that is.

## Thread Safety

---

- ◆ There is no inherent relationship between an object's intrinsic lock and its state; an object's fields need not be guarded by its intrinsic lock (you could create and use other lock objects); though this is a valid locking convention that is used by many classes.
- ◆ A common locking convention is to encapsulate all mutable state within an object and to protect it from concurrent access by synchronizing any code path that accesses mutable state using the object's intrinsic lock. This pattern is used by many thread-safe classes, such as `Vector` and other synchronized collection classes.
- ◆ Acquiring the lock associated with an object does not prevent other threads from accessing that object—the only thing that acquiring a lock prevents any other thread from doing is acquiring that same lock.

## Thread Safety

---

- ◆ When a variable is guarded by a lock—meaning that every access to that variable is performed with that lock held—you've ensured that only one thread at a time can access that variable.
- ◆ When a class has invariants that involve more than one state variable, there is an additional requirement: each variable participating in the invariant must be guarded by the *same* lock.
- ◆ This allows you to access or update state variables in a single atomic operation, preserving the invariant

For every invariant that involves more than one variable, *all* the variables involved in that invariant must be guarded by the *same* lock.

# Thread Safety

---

## ◆ Poor concurrency:

- Performance for concurrent invocations could be limited by the availability of processing resources, and by the structure of the application itself.
- Performance for concurrent invocations could be improved by narrowing the scope of the synchronized block.
- Consider to exclude from synchronized blocks long-running operations that do not affect shared state, so that other threads are not prevented from accessing the shared state while the long-running operation is in progress
- Acquiring and releasing a lock has some overhead, so it is undesirable to break down synchronized blocks too far
- The code paths in each of the synchronized blocks should be "short enough"

## Thread Safety

---

### ◆ Concurrency Design Forces:

- Deciding how big or small to make synchronized blocks may require tradeoffs among competing design forces, including safety (which must not be compromised), simplicity, and performance

Avoid holding locks during lengthy computations or operations at risk of not completing quickly such as network or console I/O.

## Mutual Exclusion in Java

---

Concurrent activations of a method in Java can be made mutually exclusive by prefixing the method with the keyword **synchronized**.

```
class SynchronizedCounter {  
    int i;  
    ---  
    ---  
  
    synchronized void increment() {  
        i++;  
    }  
}
```

# Monitors and Condition synchronization

---

monitors:

encapsulated data + access procedures  
mutual exclusion + condition synchronization  
single access procedure active in the monitor

nested monitors problem: What is it?

Java Monitor Model:

1. guarded actions
2. private data and synchronized methods (exclusion).
3. `wait()`, `notify()` and `notifyAll()` for condition synch.
4. single thread active in the monitor at a time

## Condition synchronization

---

### Car Park Example:

A controller is required for a carpark, which only permits cars to enter when the carpark is not full and does not permit cars to leave when there are no cars in the carpark.

Car arrival and departure are simulated by separate threads.

## Carpark program

---

- Model - all entities are **processes** interacting by actions
- Program - need to identify **threads** and **monitors**
  - **thread** - **active** entity which initiates (output) actions
  - **monitor** - **passive** entity which responds to (input) actions.

## Carpark program

---

**Arrivals** and **Departures** implement **Runnable**,  
**CarParkControl** provides the control (condition synchronization).

Instances of these are created by the **start()** method of the  
**CarPark** applet :

```
public void start() {  
    CarParkControl c =  
        new DisplayCarPark(carDisplay,Places);  
    arrivals.start(new Arrivals(c));  
    departures.start(new Departures(c));  
}
```

## carpark program - Arrivals and Departures threads

---

```
class Arrivals implements Runnable {  
    CarParkControl carpark;  
  
    Arrivals(CarParkControl c) {carpark = c;}  
  
    public void run() {  
        try {  
            while(true) {  
                ...  
                carpark.arrive();  
                ...  
            }  
        } catch (InterruptedException e) {}  
    }  
}
```

Similarly **Departures**  
which calls  
**carpark.depart()**.

How do we implement the control of **CarParkControl**?

## Carpark program - CarParkControl monitor

```
class CarParkControl {  
    protected int spaces;  
    protected int capacity;  
  
    CarParkControl(int n)  
        {capacity = spaces = n; }  
  
    synchronized void arrive() {  
        ... --spaces; ...  
    }  
  
    synchronized void depart() {  
        ... ++spaces; ...  
    }  
}
```

*mutual exclusion  
by synch methods*

*condition  
synchronization?*

*block if full?  
(spaces==0)*

*block if empty?  
(spaces==N)*

## condition synchronization in Java

---

Java provides a thread **wait set** per monitor (or per object) with the following methods:

**public final void notify()**

Wakes up a single thread that is waiting on this object's set.

**public final void notifyAll()**

Wakes up all threads that are waiting on this object's set.

**public final void wait()**

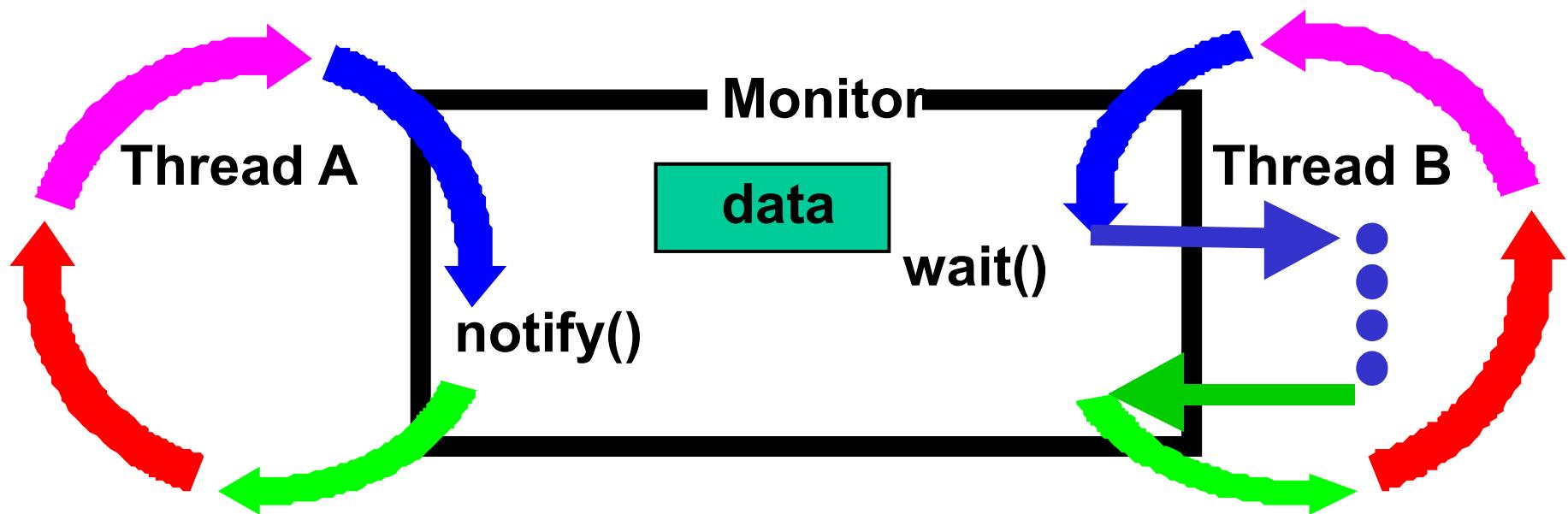
**throws InterruptedException**

Waits to be notified by another thread. The waiting thread releases the synchronization lock associated with the monitor. When notified, the thread must wait to reacquire the monitor before resuming execution.

## condition synchronization in Java

We refer to a thread *entering* a monitor when it acquires the mutual exclusion lock associated with the monitor and *exiting* the monitor when it releases the lock.

**Wait()** - causes the thread to exit the monitor,  
permitting other threads to enter the monitor.



## condition synchronization in Java

---

```
public synchronized void act()
    throws InterruptedException
{
    while (!cond) wait();
    // modify monitor data
    notifyAll()
}
```

The **while** loop is necessary to retest the condition *cond* to ensure that *cond* is indeed satisfied when it re-enters the monitor.

**notifyall()** is necessary to awaken other thread(s) that may be waiting to enter the monitor now that the monitor data has been changed.

## CarParkControl - condition synchronization

---

```
class CarParkControl {  
    protected int spaces;  
    protected int capacity;  
  
    CarParkControl(int n)  
        {capacity = spaces = n; }  
  
    synchronized void arrive() throws InterruptedException {  
        while (spaces==0) wait();  
        --spaces;  
        notify();  
    }  
  
    synchronized void depart() throws InterruptedException {  
        while (spaces==capacity) wait();  
        ++spaces;  
        notify();  
    }  
}
```

*Why is it safe to use `notify()` here rather than `notifyAll()`?*

## Modeling Concurrency in Java

---

**Active** entities (that initiate actions) are implemented as **threads**.

**Passive** entities (that respond to actions) are implemented as **monitors**.

- Each guarded action in the model of a monitor is implemented as a **synchronized** method which uses a while loop and **wait()** to implement the guard. The while loop condition is the negation of the model guard condition.
- Changes in the state of the monitor are signaled to waiting threads using **notify()** or **notifyAll()** .

# Semaphores in Java

---

*Wait = down(s):*

**if**  $s > 0$  **then**

    decrement  $s$

**else**

    block execution of the calling process

*Signal = up(s):*

**if** processes blocked on  $s$  **then**

    awaken one of them

**else**

    increment  $s$

## Semaphores in Java

---

Semaphores are passive objects, therefore implemented as **monitors**.

*(In practice, semaphores are a low-level mechanism often used in implementing the higher-level monitor construct.)*

```
public class Semaphore {  
    private int value;  
  
    public Semaphore (int initial)  
        {value = initial;}  
  
    synchronized public void up () {  
        ++value;  
        notify();  
    }  
  
    synchronized public void down ()  
        throws InterruptedException {  
        while (value== 0) wait();  
        --value;  
    }  
}
```

## Bounded Buffer

---

### Bounded Buffer Example

A bounded buffer consists of a fixed number of slots. Items are put into the buffer by a *producer* process and removed by a *consumer* process. It can be used to smooth out transfer rates between the *producer* and *consumer*.



## Bounded buffer program - buffer monitor

---

```
public interface Buffer {...}

class BufferImpl implements Buffer {
    ...
    public synchronized void put(Object o)
        throws InterruptedException {
        while (count==size) wait();
        buf[in] = o; ++count; in=(in+1)%size;
        notify();
    }
    public synchronized Object get()
        throws InterruptedException {
        while (count==0) wait();
        Object o =buf[out];
        buf[out]=null; --count; out=(out+1)%size;
        notify();
        return (o);
    }
}
```

## Bounded buffer program - producer process

---

```
class Producer implements Runnable {  
    Buffer buf;  
    String alphabet= "abcdefghijklmnopqrstuvwxyz";  
    Producer(Buffer b) {buf = b;}  
  
    public void run() {  
        try {  
            int ai = 0;  
            while(true) {  
  
                buf.put(new Character(alphabet.charAt(ai)));  
                ai=(ai+1) % alphabet.length();  
  
            }  
        } catch (InterruptedException e) {}  
    }  
}
```

Similarly Consumer  
which calls **buf.get()**.

## Nested Monitors

---

Suppose that, in place of using the *count* variable and condition synchronization directly, we instead use two semaphores *full* and *empty* to reflect the state of the buffer.

```
class SemaBuffer implements Buffer {  
    ...  
  
    Semaphore full; //counts number of items  
    Semaphore empty; //counts number of spaces  
  
    SemaBuffer(int size) {  
        this.size = size; buf = new Object[size];  
        full = new Semaphore(0);  
        empty= new Semaphore(size);  
    }  
    ...  
}
```

## Nested Monitors - Bounded Buffer Program

---

```
synchronized public void put(Object o)
    throws InterruptedException {
    empty.down();
    buf[in] = o;
    ++count; in=(in+1)%size;
    full.up();
}

synchronized public Object get()
    throws InterruptedException{
    full.down();
    Object o =buf[out]; buf[out]=null;
    --count; out=(out+1)%size;
    empty.up();
    return (o);
}
```

*Does this behave  
as desired?*

*empty* is decremented during a **put** operation, which is blocked if *empty* is zero; *full* is decremented by a **get** operation, which is blocked if *full* is zero.

## Nested monitors - bounded buffer model

---

The Consumer tries to get a character, but the buffer is empty. It blocks and releases the lock on the semaphore full. The Producer tries to put a character into the buffer, but also blocks. **Why?**

This situation is known as the **nested monitor problem**.

## Nested monitors - revised bounded buffer program

---

The only way to avoid it in Java is by careful design. In this example, the deadlock can be removed by ensuring that the monitor lock for the buffer is not acquired until *after* semaphores are decremented.

```
public void put(Object o)
                throws InterruptedException {
    empty.down();
    synchronized(this) {
        buf[in] = o; ++count; in=(in+1)%size;
    }
    full.up();
}
```