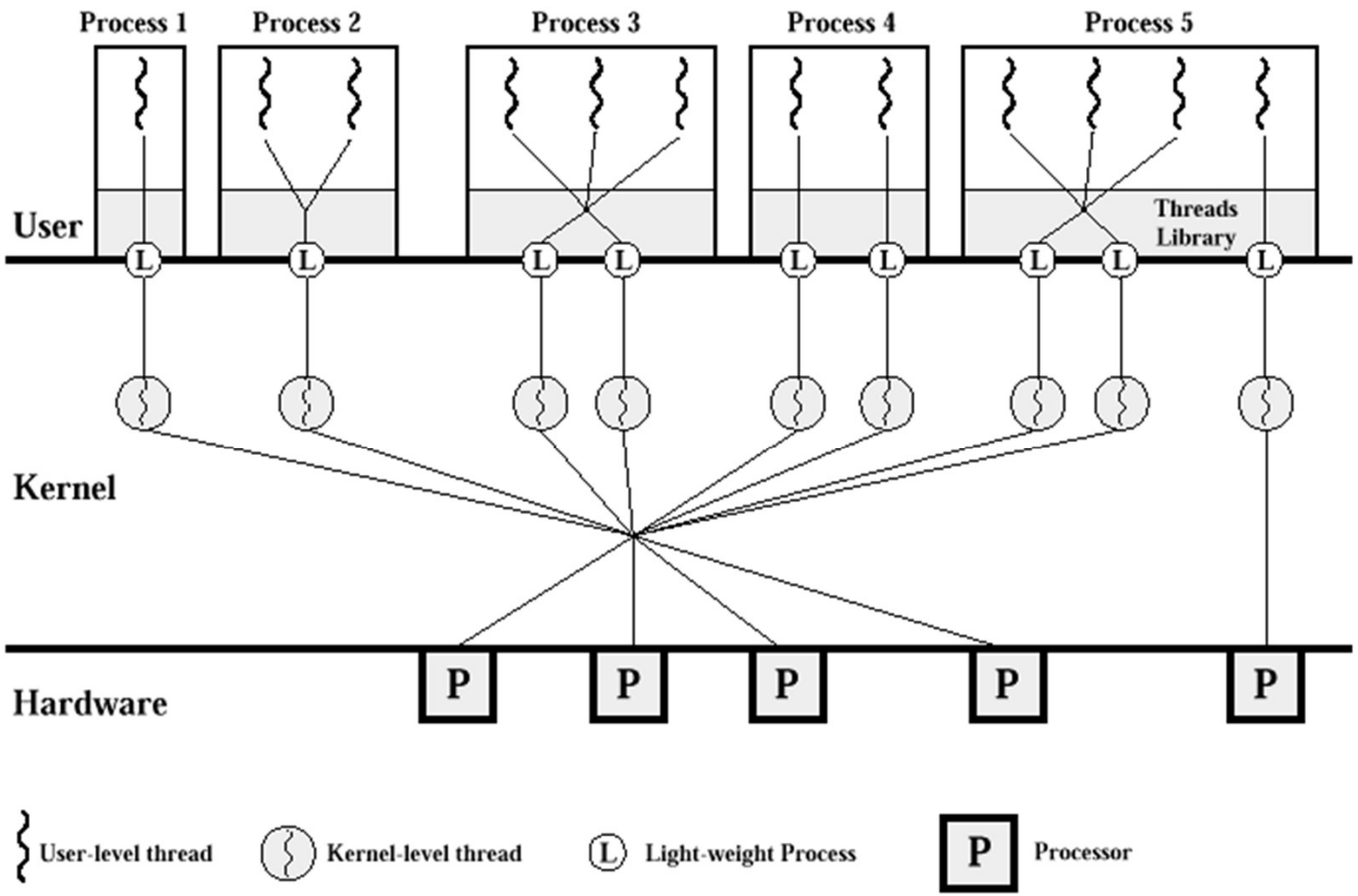
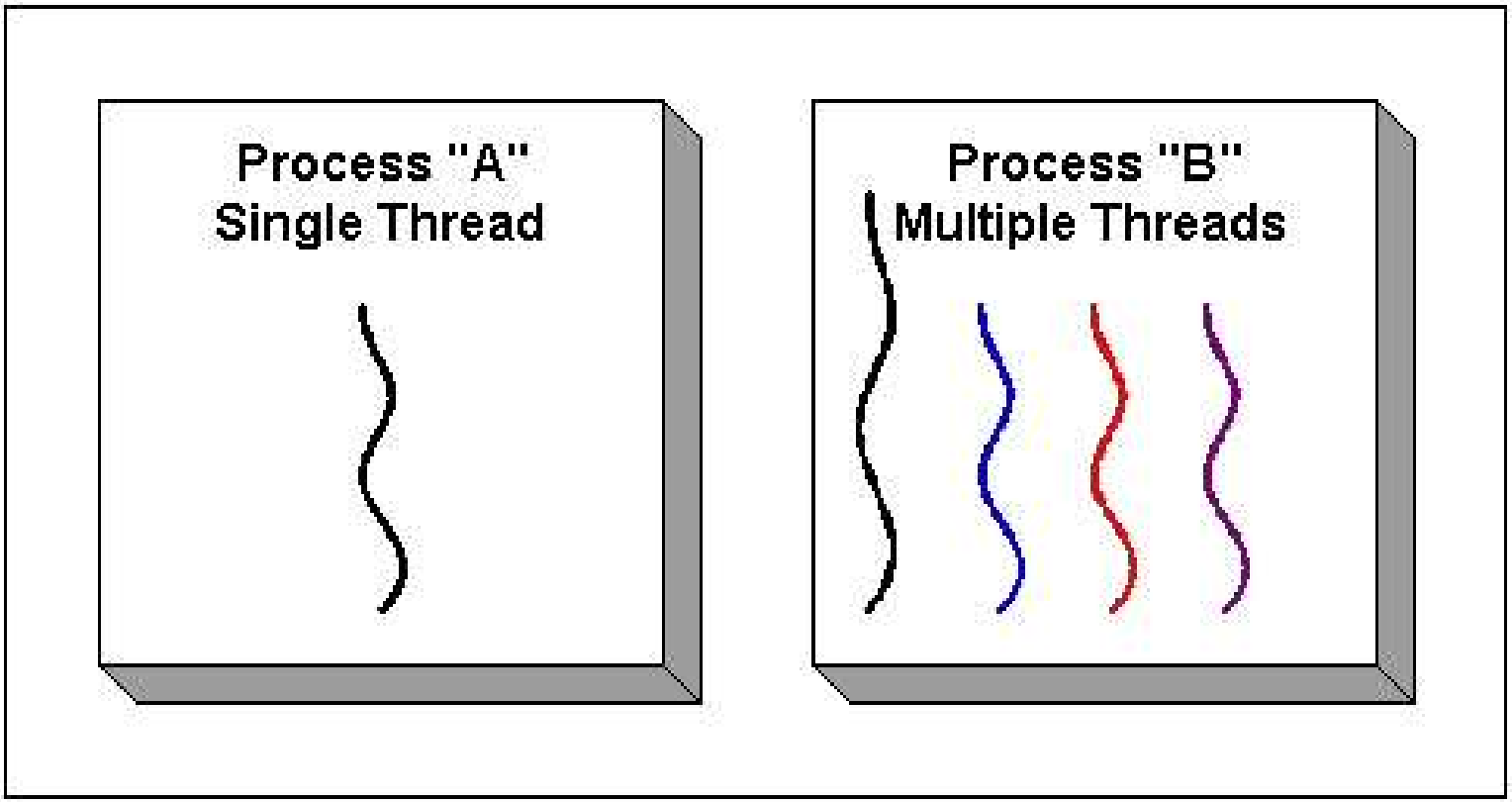
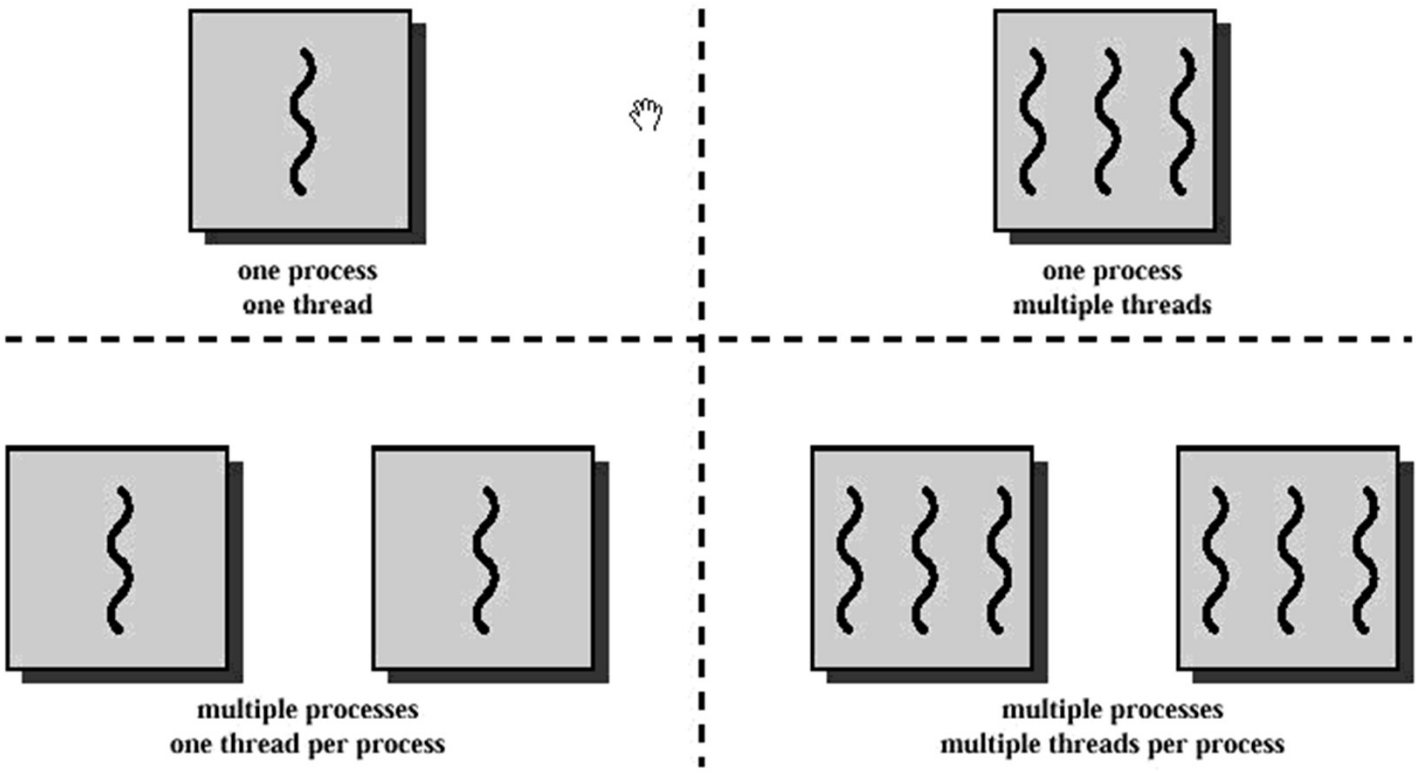
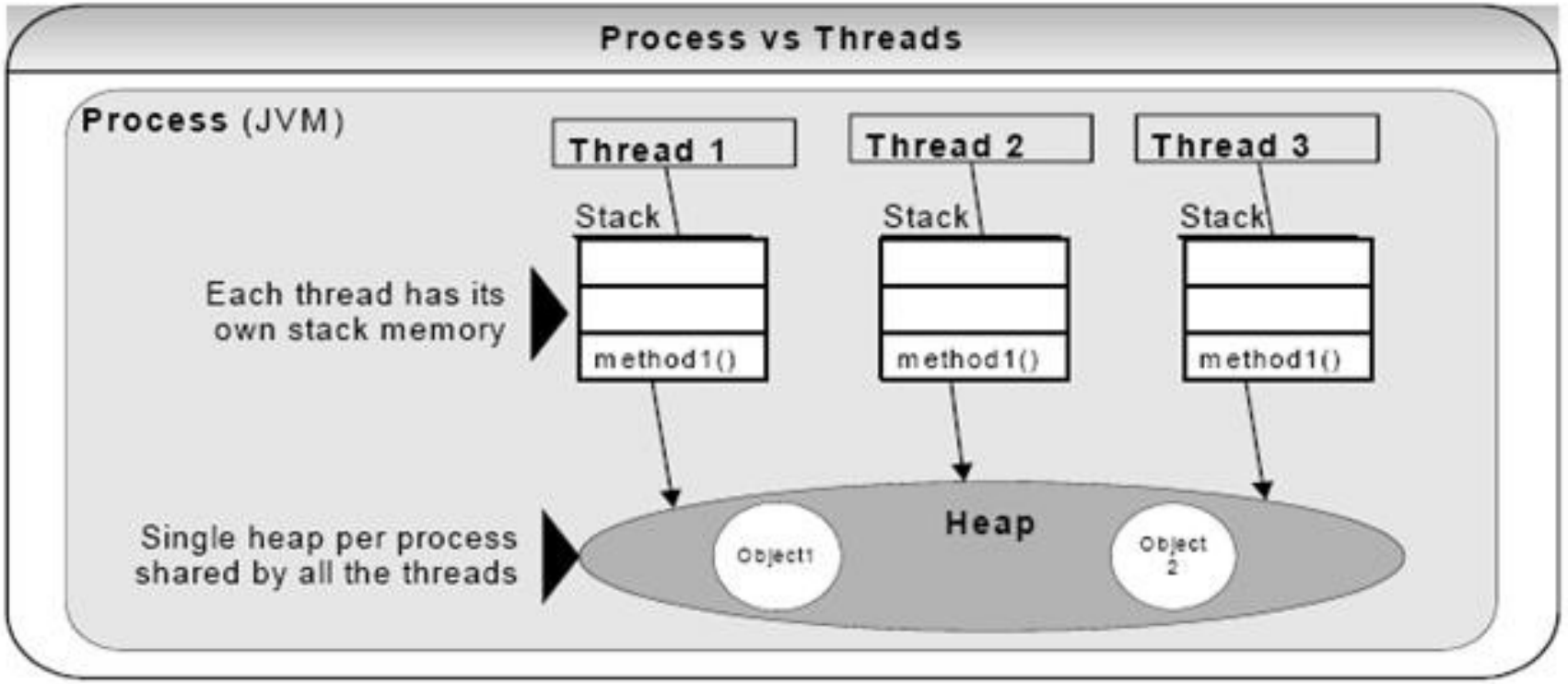
Multithreading and Concurrency Control

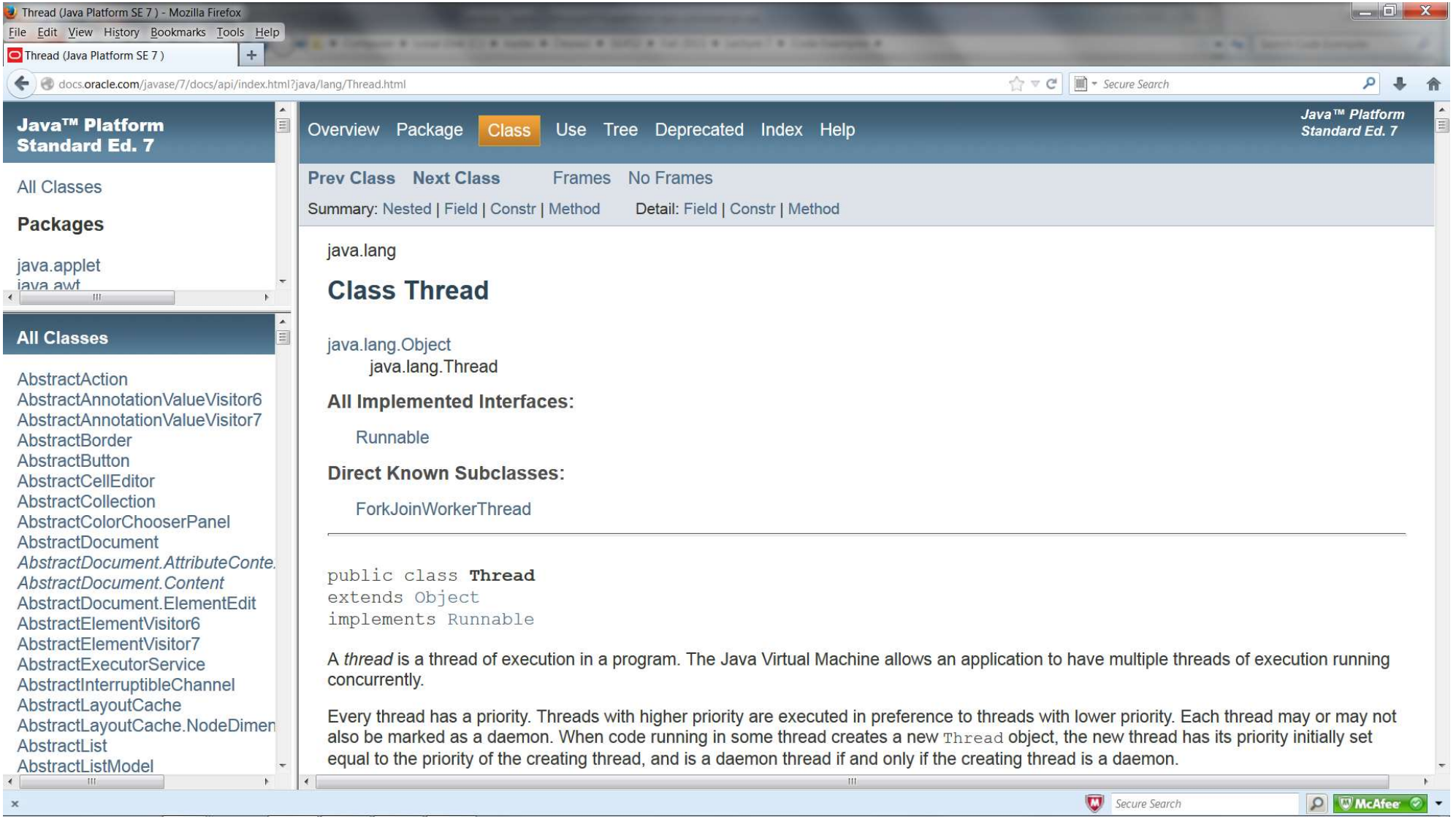

Process 1
Process 2
Process 3
Process 4
Process 5
User
Threads Library
L
Kernel
Hardware
P
User-level thread
Kernel-level thread
Light-weight Process
Processor


Process "A"
Single Thread
Process "B"
Multiple Threads

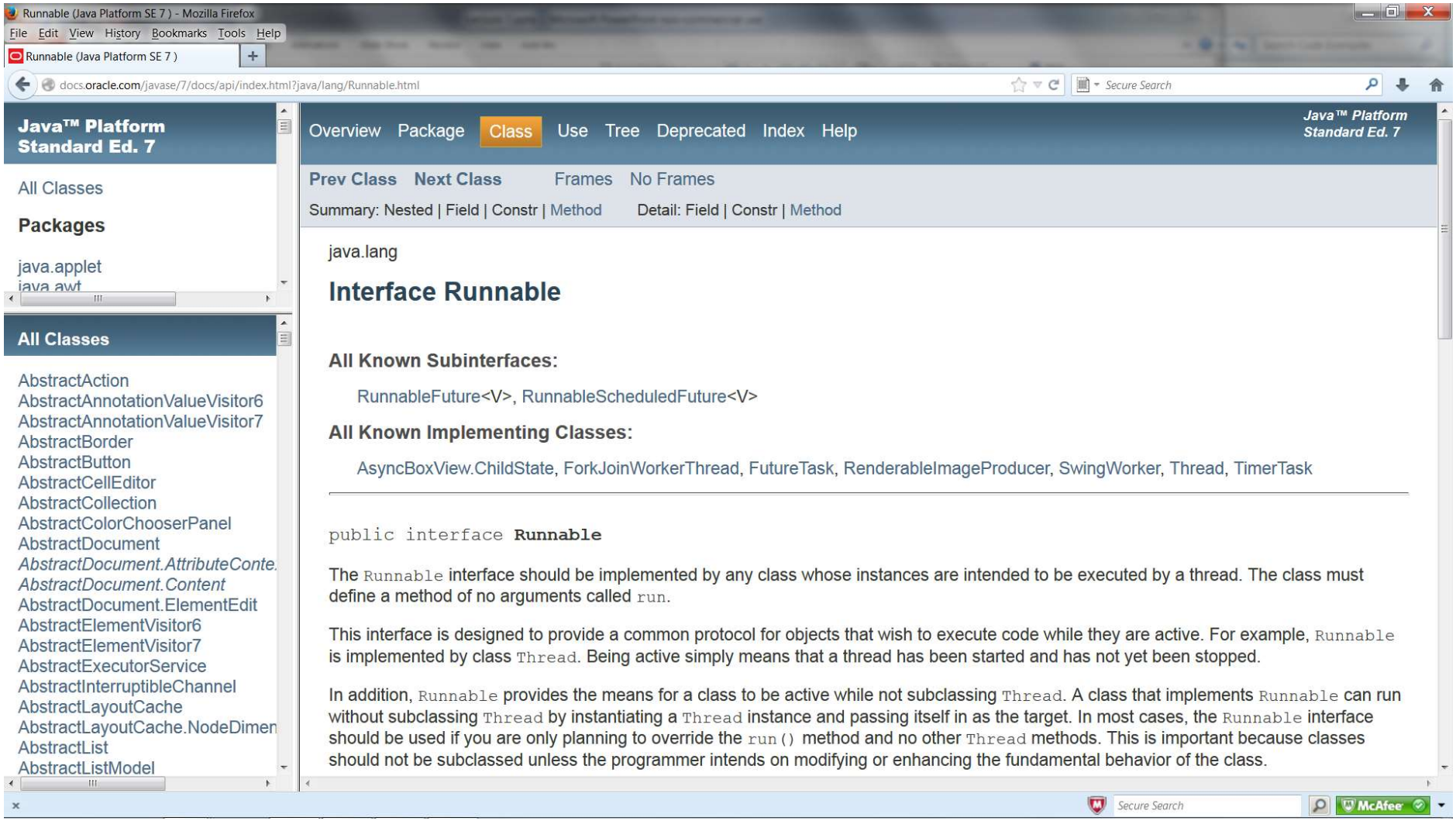

one process
one thread
one process
multiple threads
multiple processes
one thread per process
multiple processes
multiple threads per process


Process vs Threads
Process (JVM)
Thread 1
Thread 2
Thread 3
Stack
Stack
Stack
Each thread has its own stack memory
method1()
method1()
method1()
Heap
Single heap per process shared by all the threads
Object1
Object 2

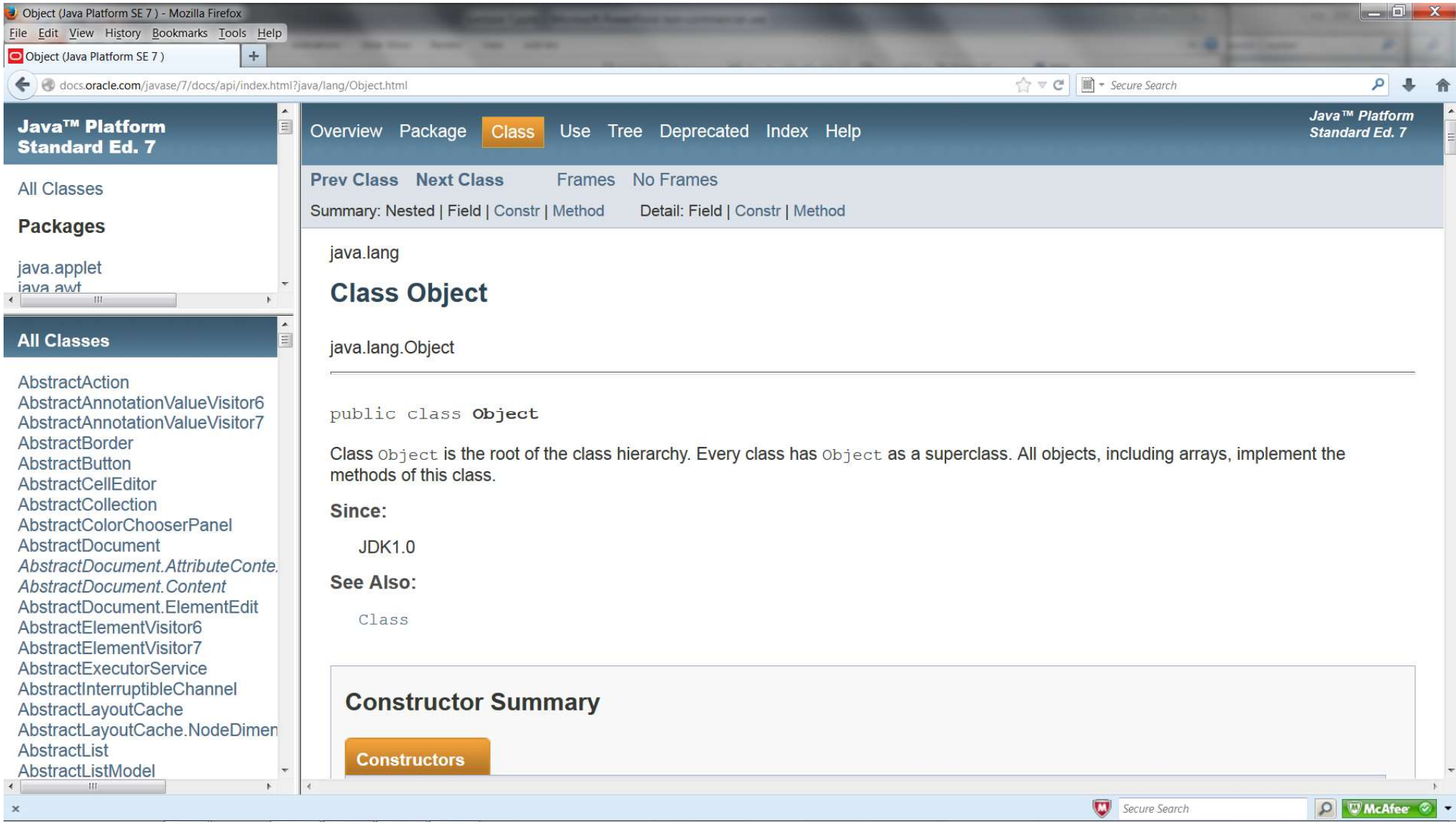
Three Java APIs you must know for Multithreading



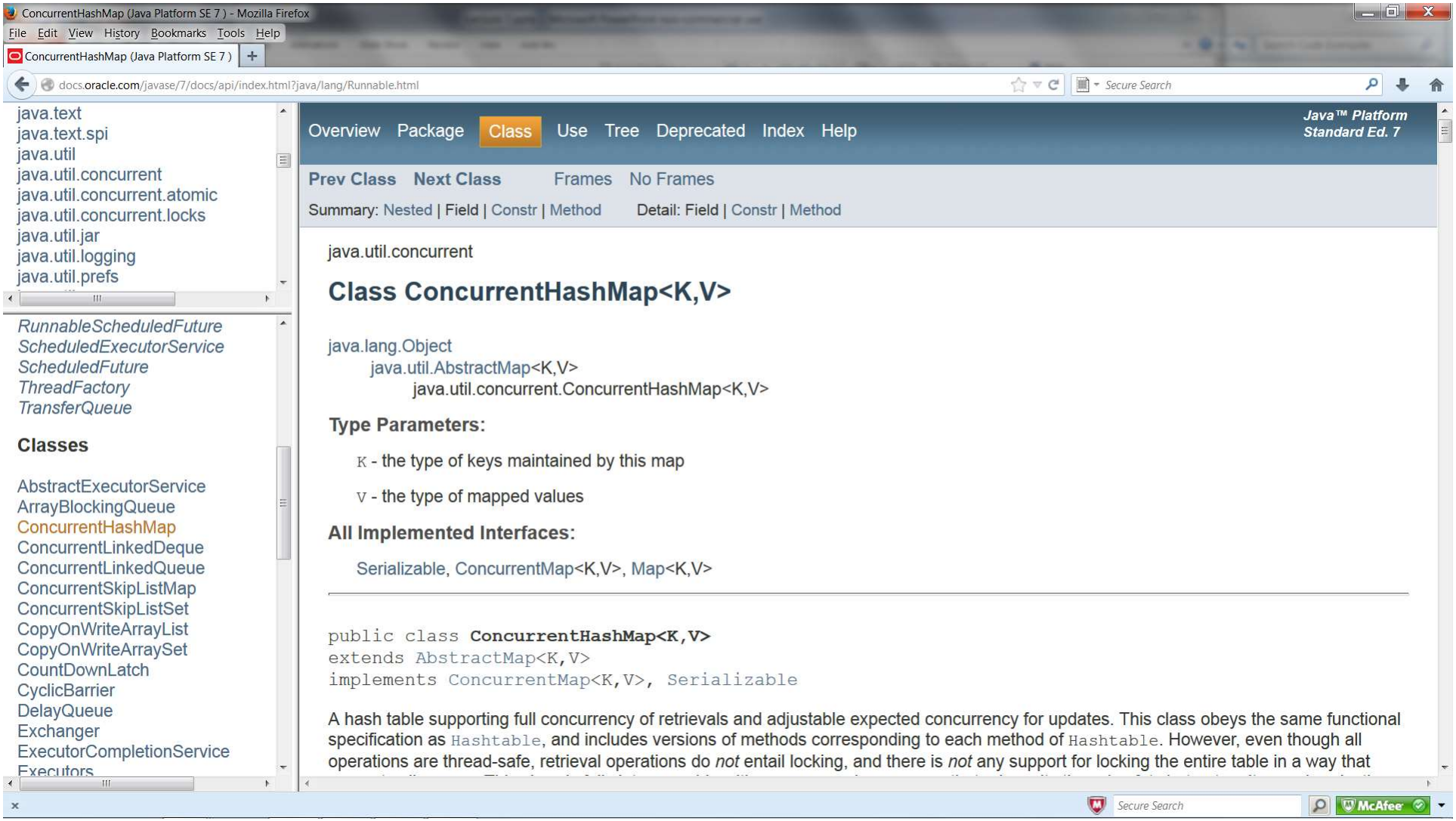
Three Java APIs you must know for Multithreading



Three Java APIs you must know for Multithreading



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



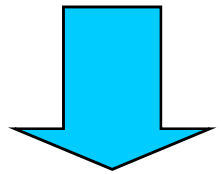
Concurrent Processes

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.

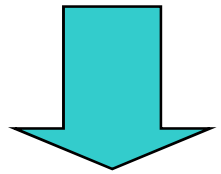
Designing concurrent software can be complex and error prone.

A rigorous engineering approach is essential.

Concept of a process as a sequence of actions.



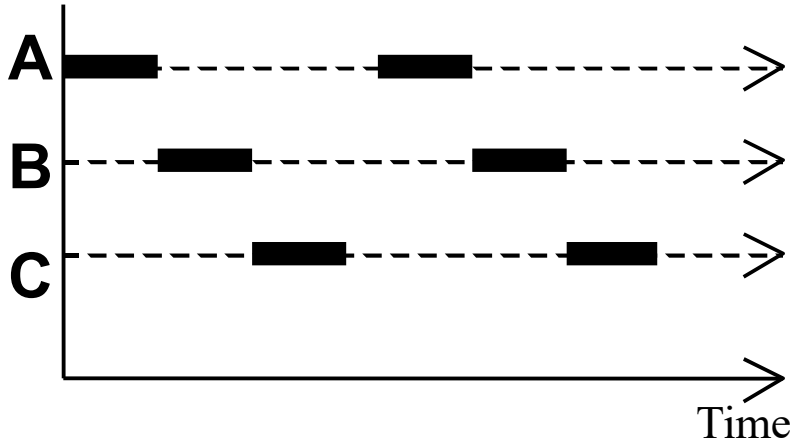
Model processes as finite state machines.



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.



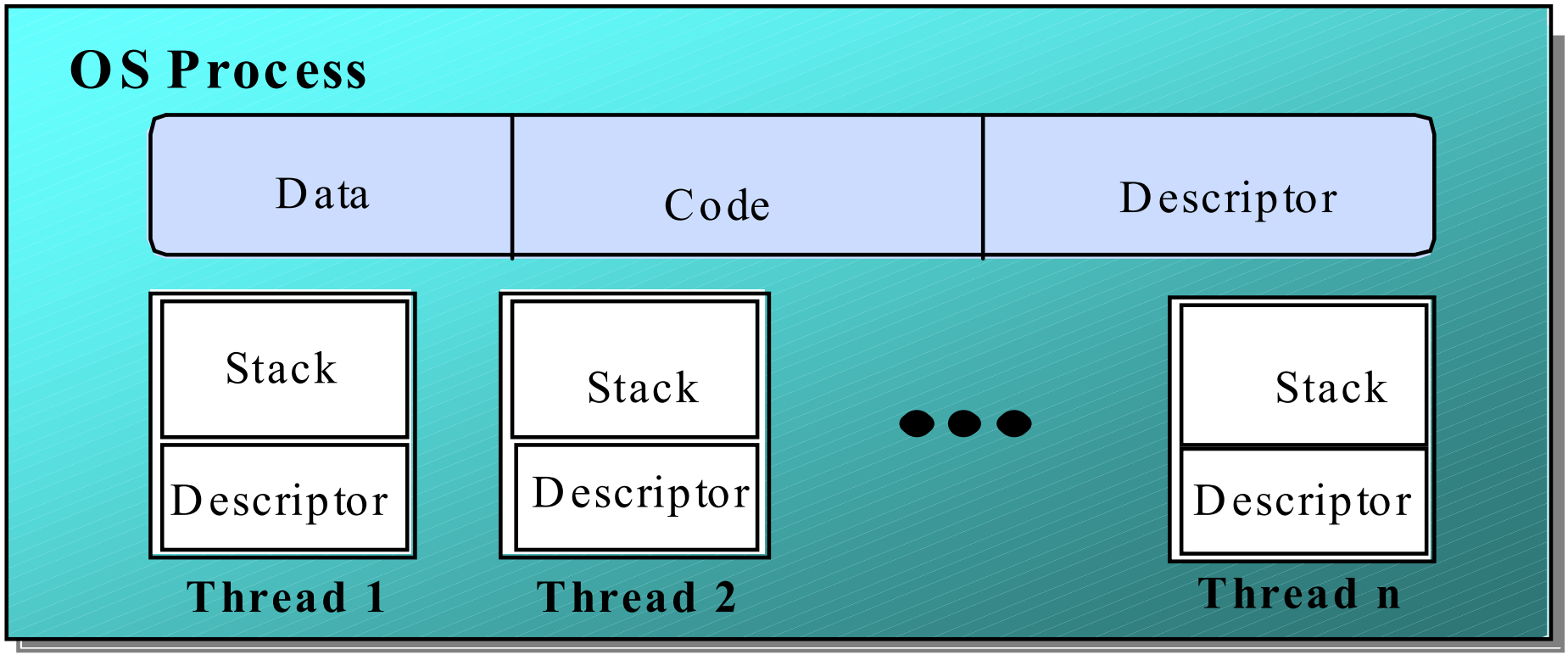
- ***Parallelism***
  - *Physically* simultaneous processing.
  - Involves multiple PEs and/or
  - independent device operations.

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

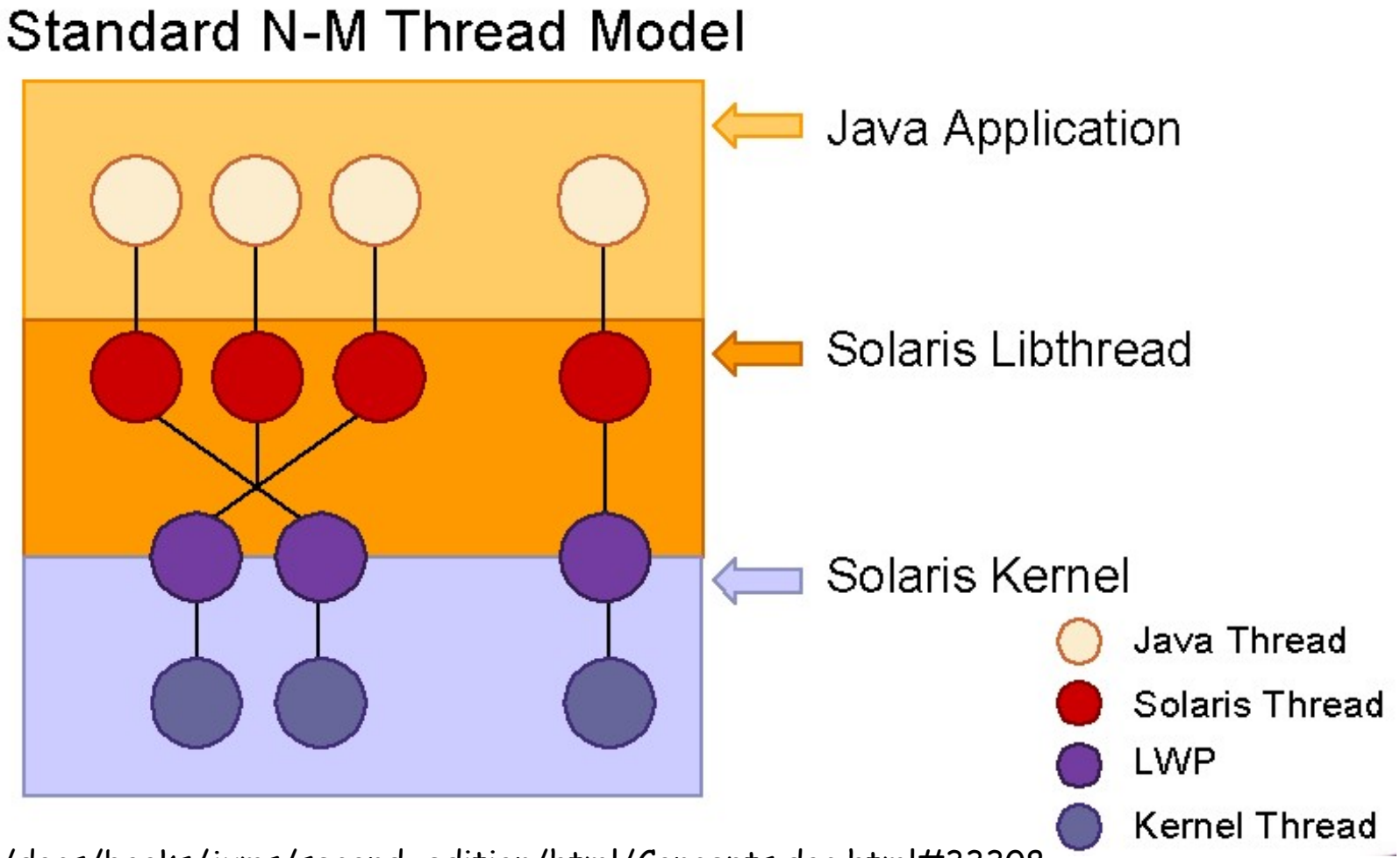
- How should we model process execution speed?
  - arbitrary speed
- How do we model concurrency?
  - arbitrary relative order of actions from different processes (**interleaving** but preservation of each process order)

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



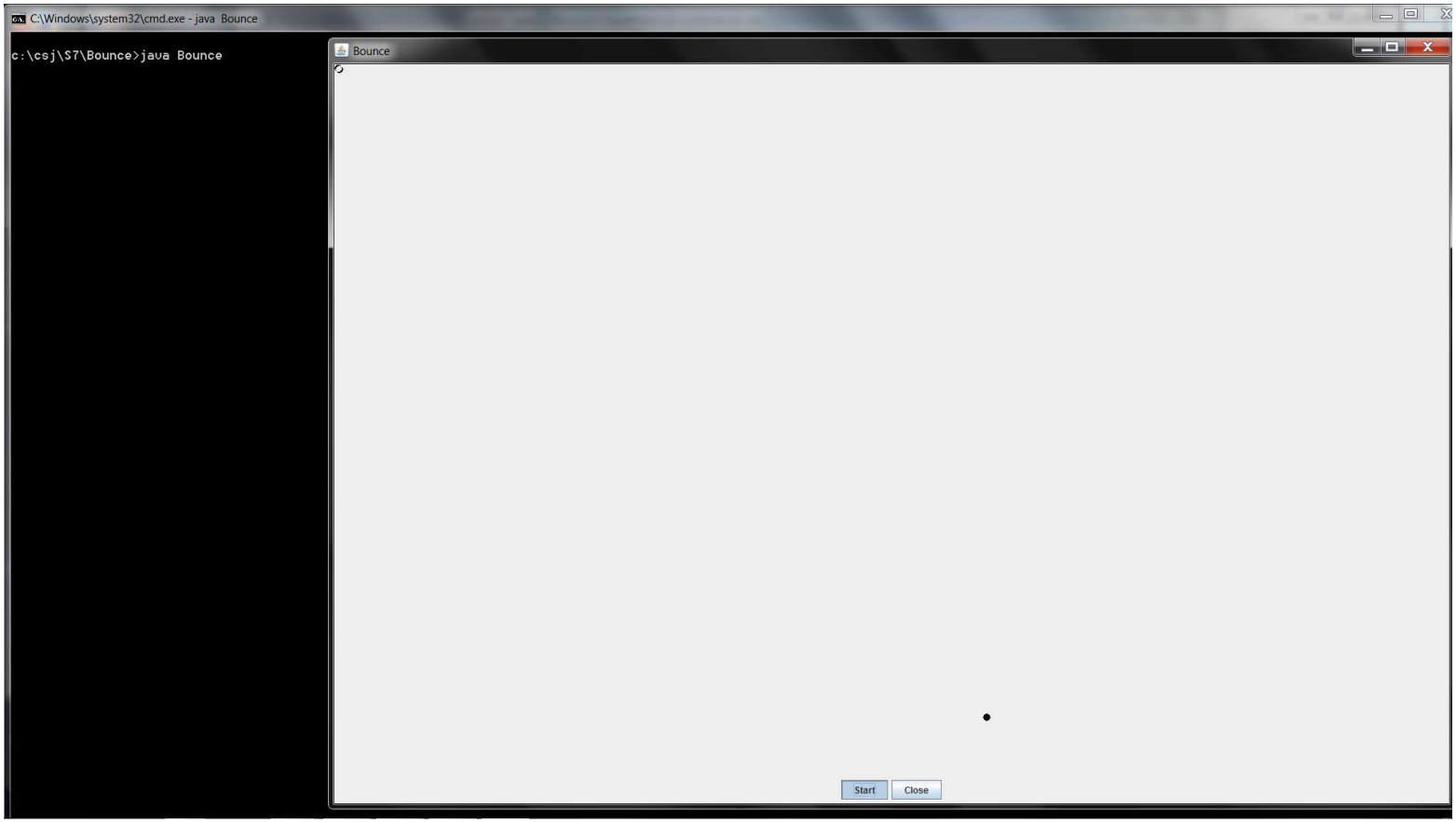
- See http://java.sun.com/docs/books/jvms/second_edition/html/Concepts.doc.html#33308
- See the URL http://www.oracle.com/technetwork/java/threads-140302.html for some performance measurements for Solaris 7, 8, and 9
- See http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html#PerformanceTuning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

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.

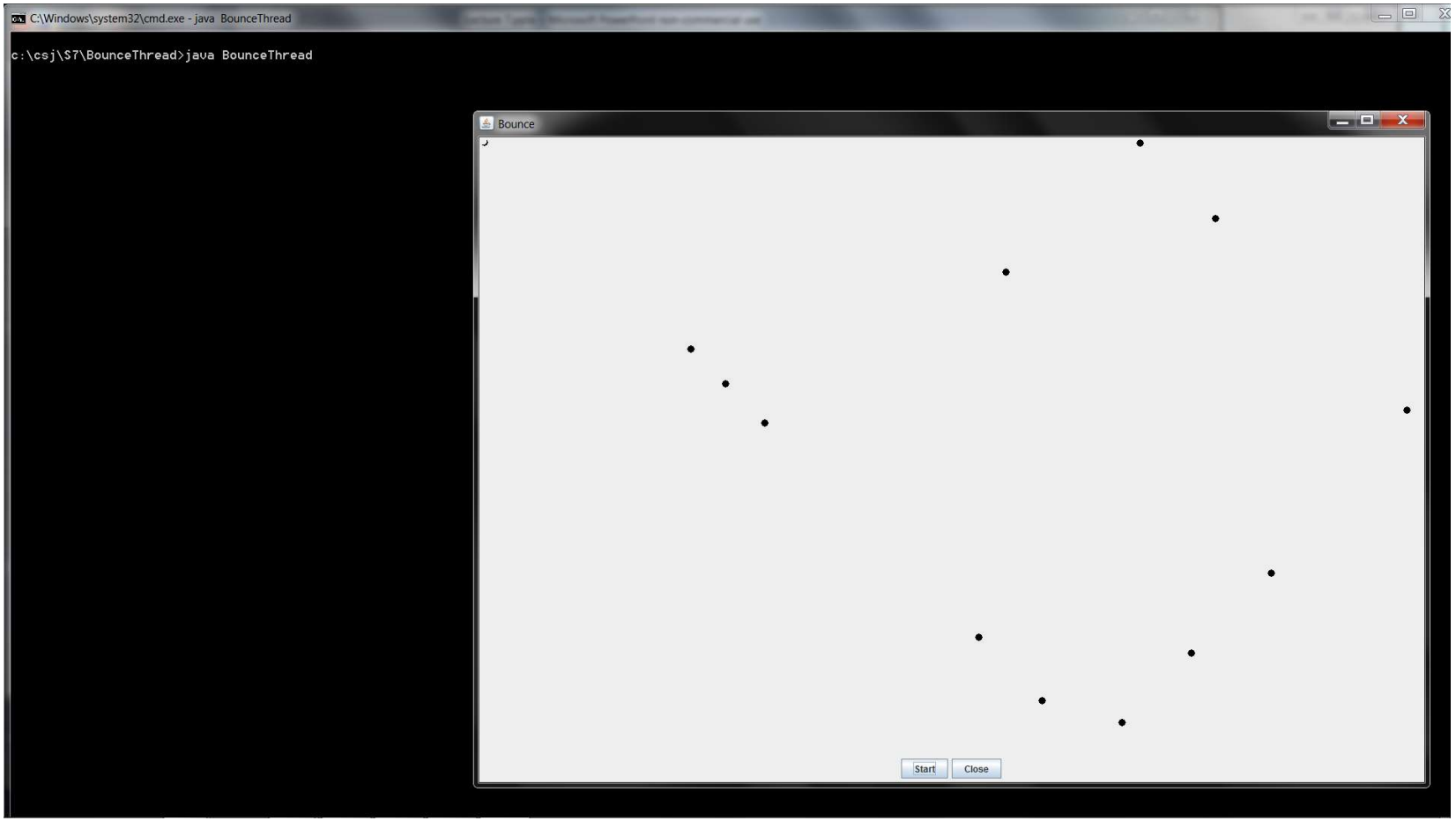
Threads and Synchronization in Java

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



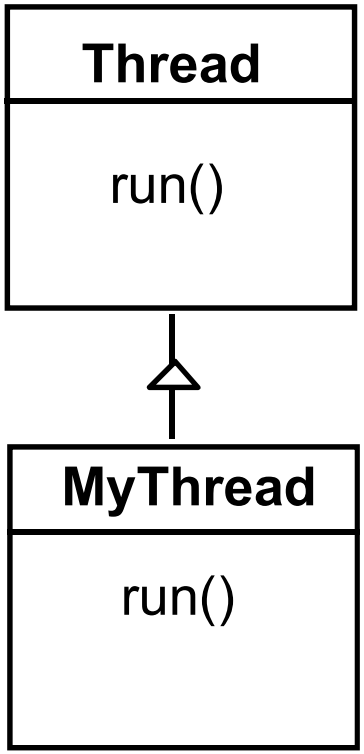
Threads and Synchronization in Java

- 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.



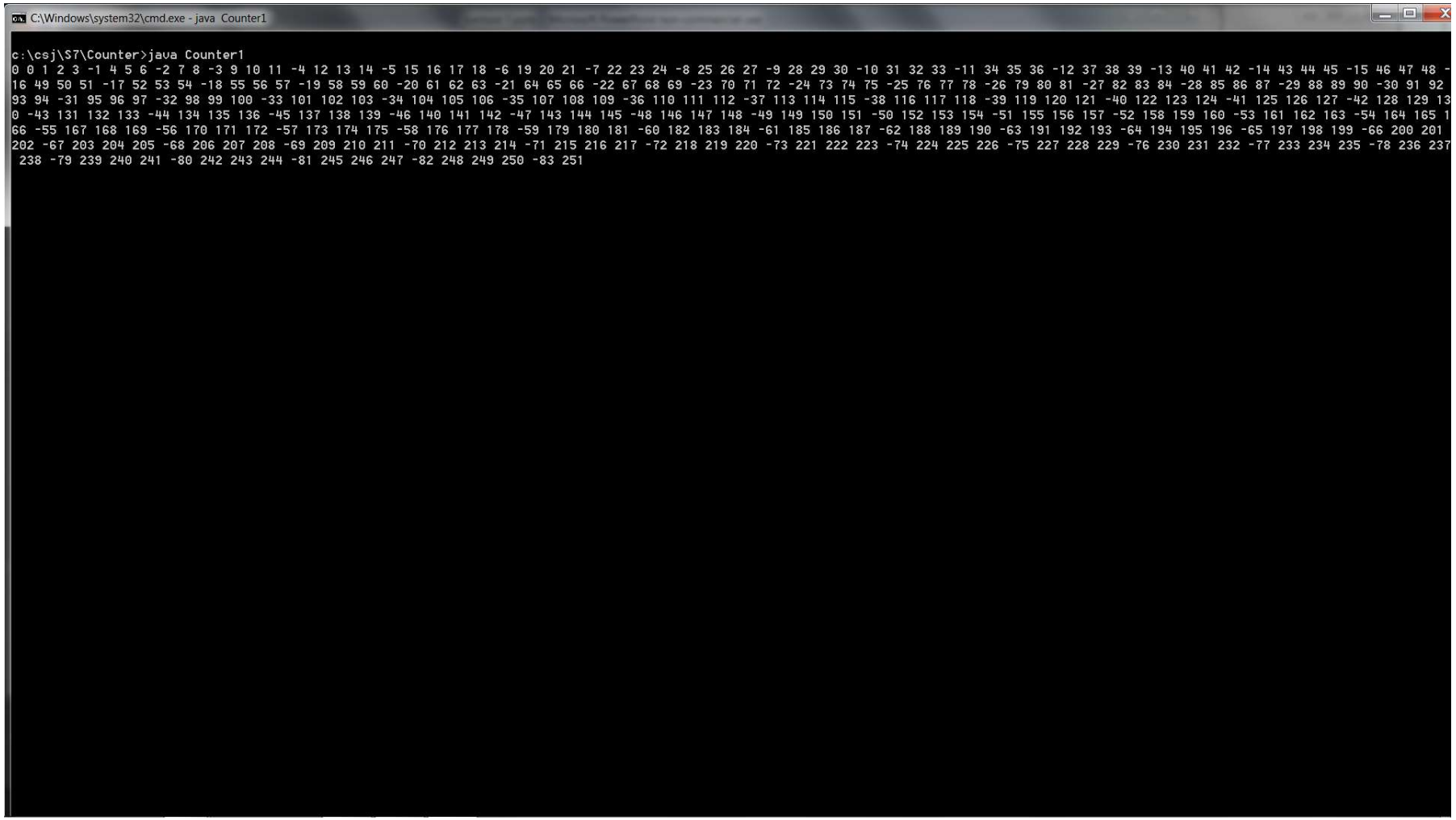
The Thread class executes instructions from its method run(). The actual code executed depends on the implementation provided for run() in a derived class.

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

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

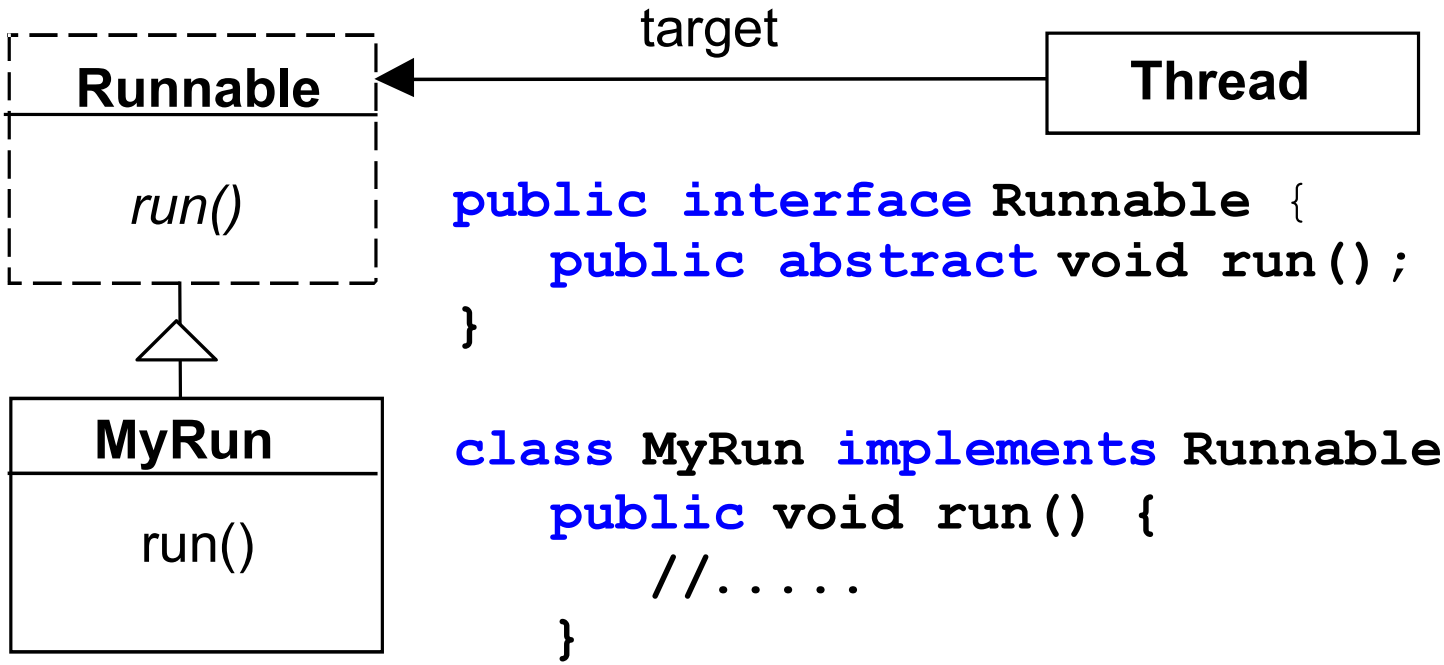
Threads in Java

An example extending the Thread class -- Counter1.java



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.



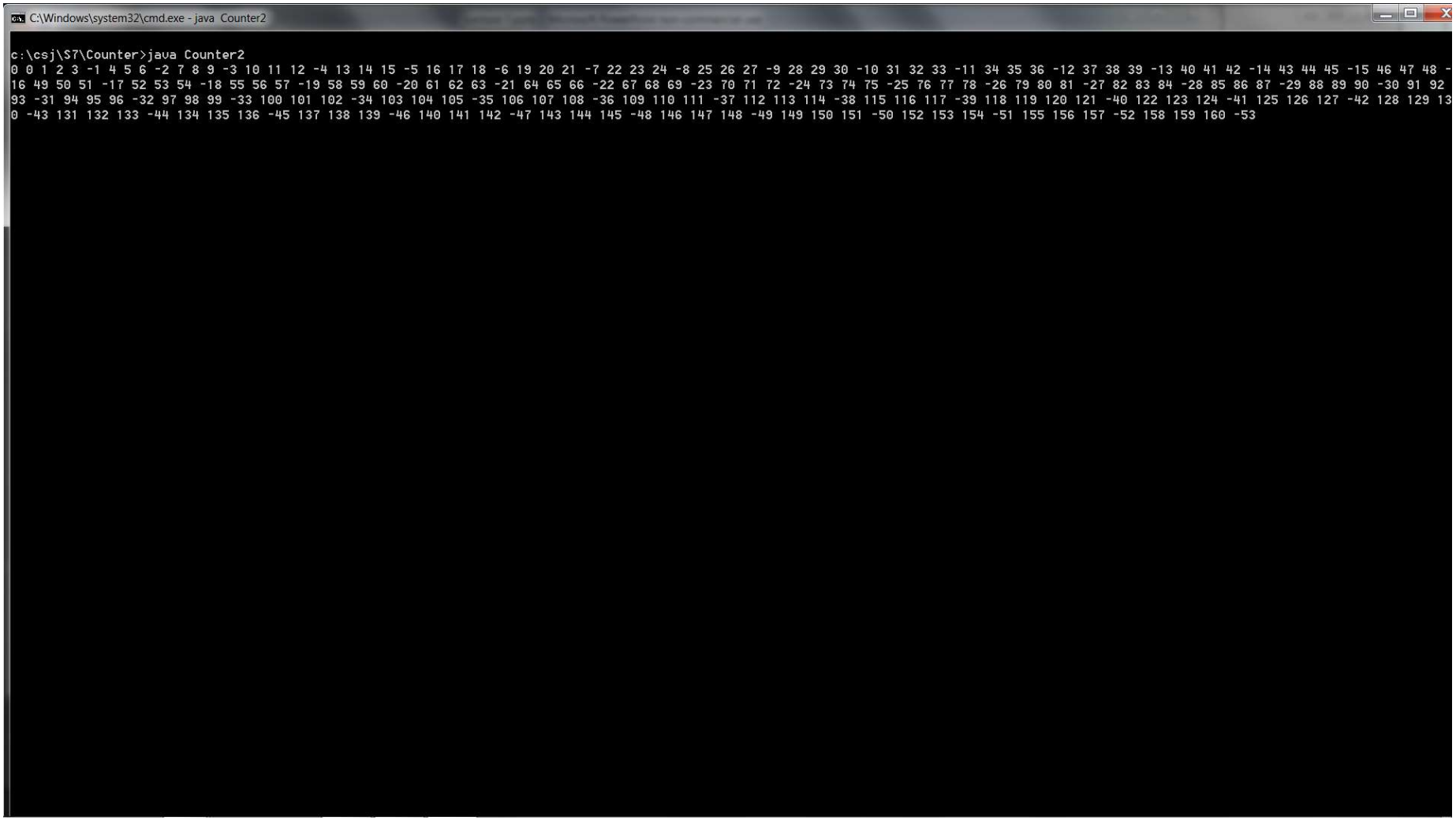

```
public interface Runnable {
    public abstract void run();
}

class MyRun implements Runnable{
    public void run() {
        //.....
    }
}

Thread x = new Thread(new MyRun());
```

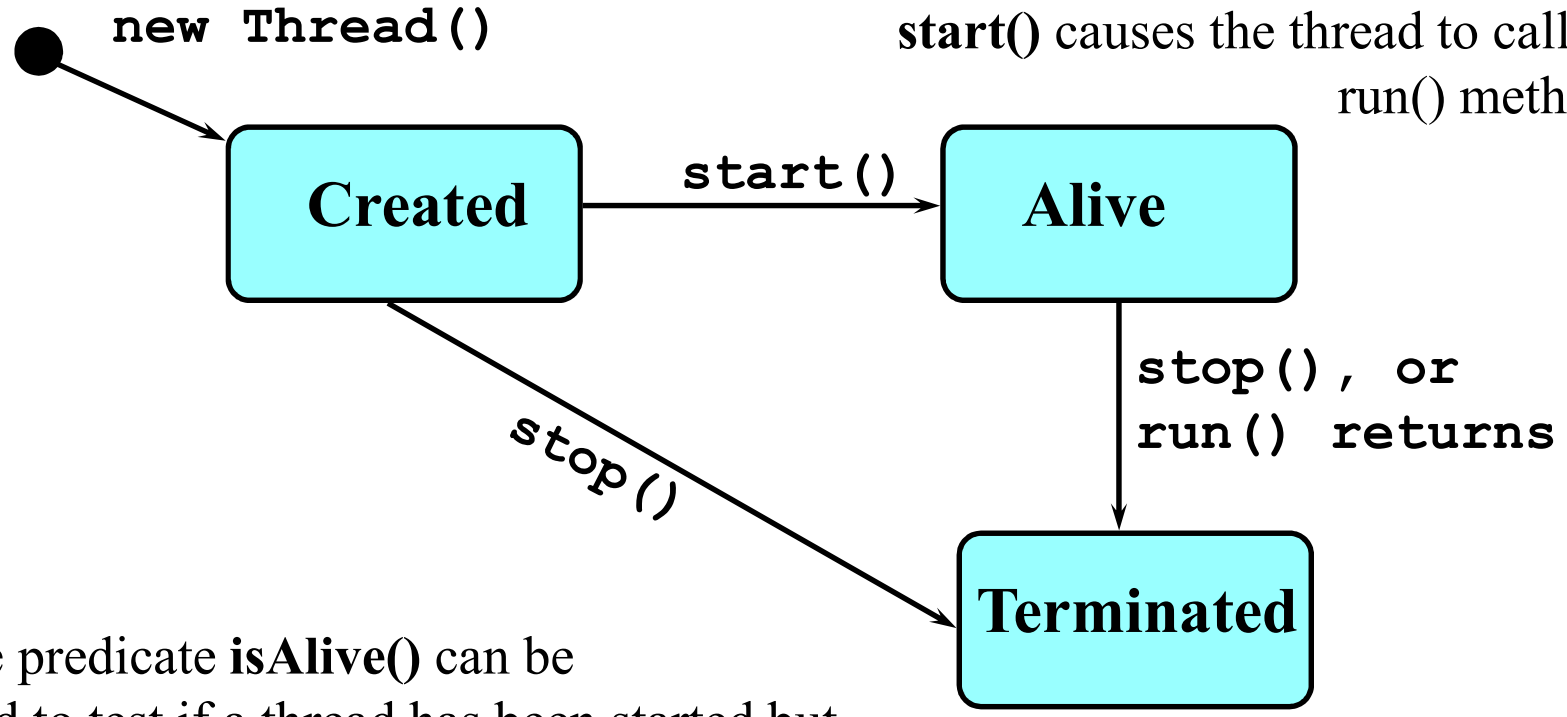
Threads in Java

An example implementing the Runnable Interface – Counter2.java



thread life-cycle in Java

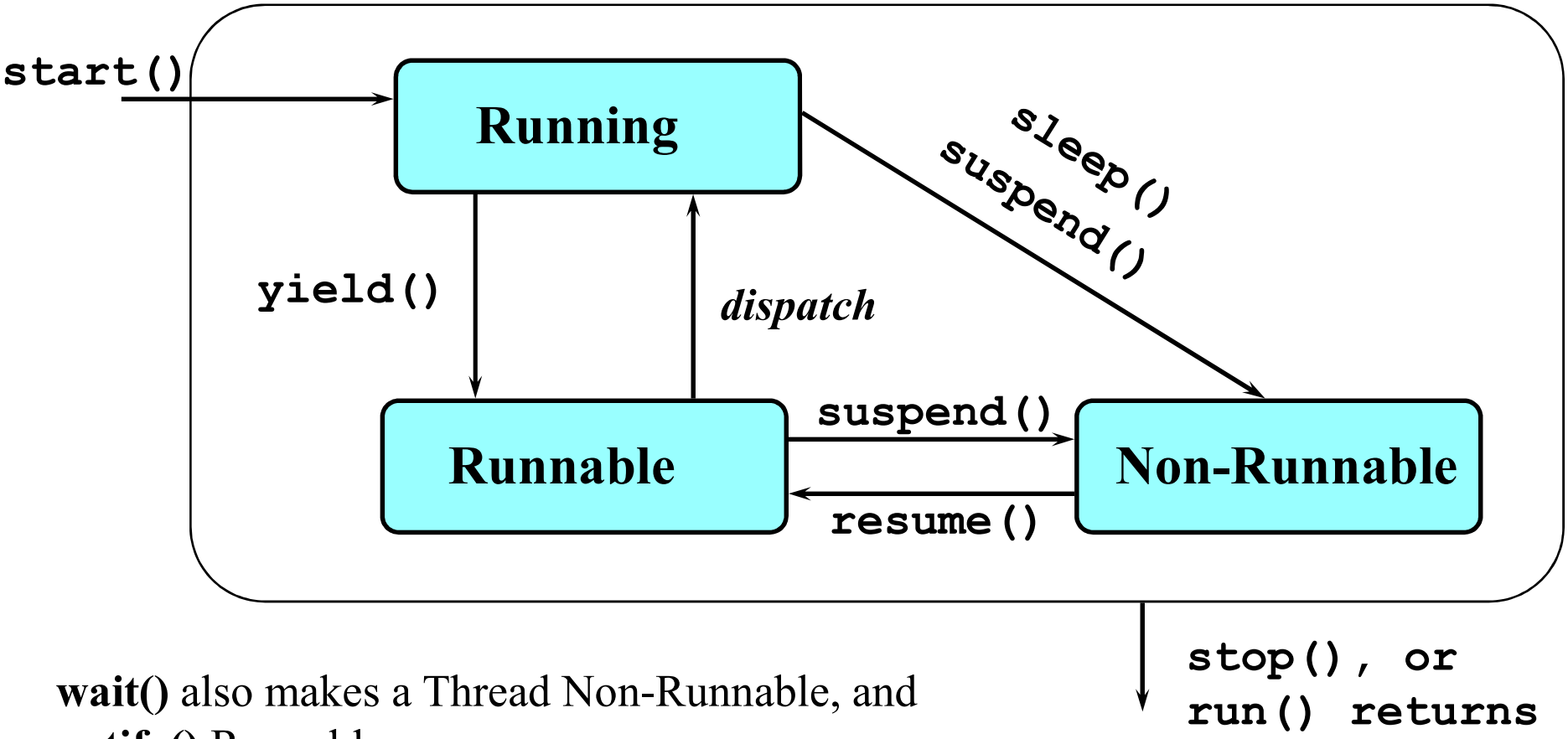
An overview of the life-cycle of a thread as state transitions:



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

Java thread lifecycle - an FSP specification

```
THREAD       = CREATED,
CREATED      = (start              ->RUNNING
                |stop               ->TERMINATED),
RUNNING      = ({suspend,sleep}    ->NON_RUNNABLE
                |yield              ->RUNNABLE
                |{stop,end}         ->TERMINATED
                |run                ->RUNNING),
RUNNABLE     = (suspend            ->NON_RUNNABLE
                |dispatch           ->RUNNING
                |stop               ->TERMINATED),
NON_RUNNABLE = (resume             ->RUNNABLE
                |stop               ->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 vatiables, 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 negitve 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

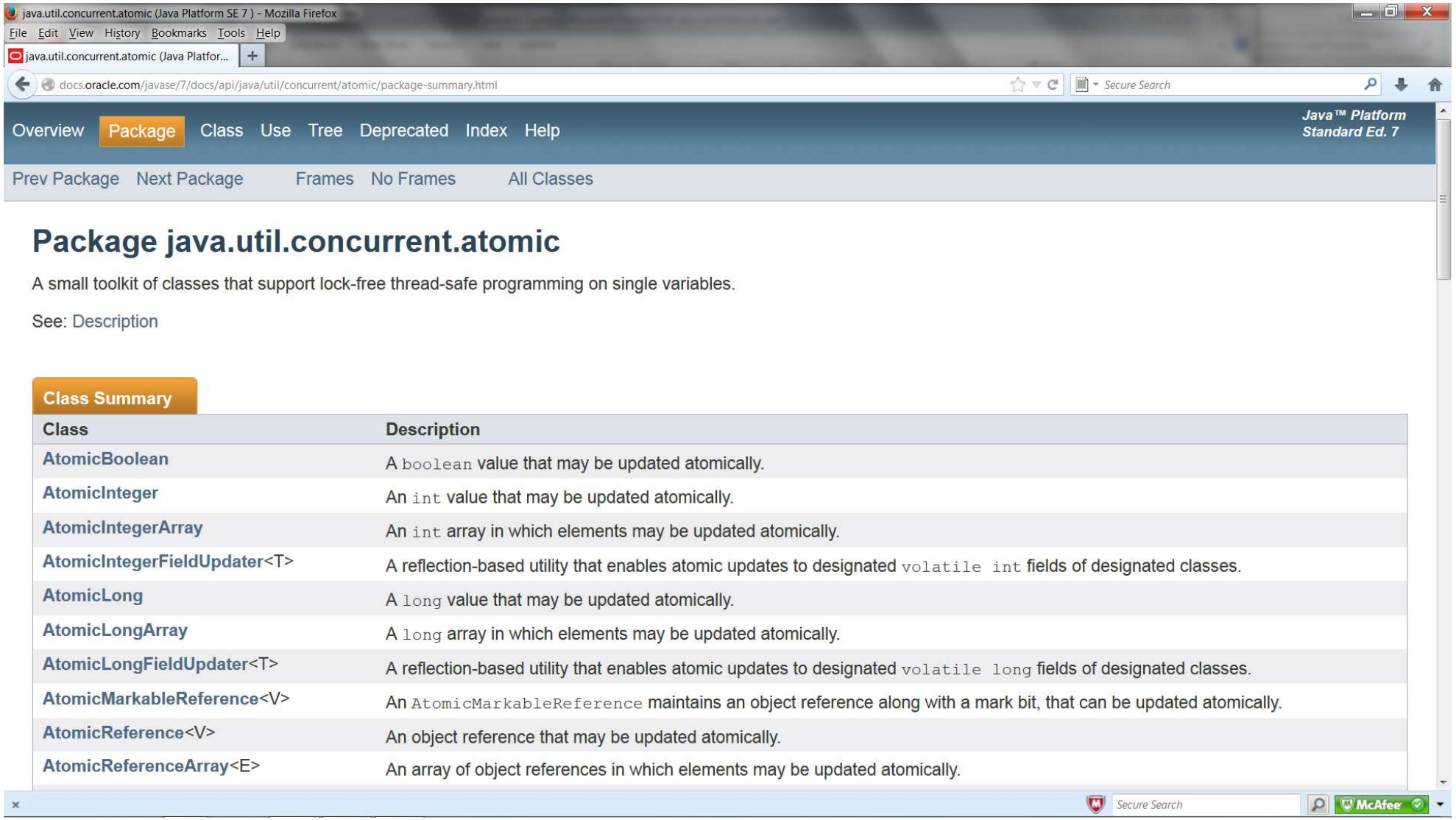
- To avoid race conditions, we must prevent other threads from using a variable while we're in the middle of modifying it, so we can ensure that other threads can observe or modify the state only before we start or after we finish, but not in the middle.

- To ensure thread safety, **check-then-act** operations (like lazy initialization) and **read-modify-write** operations (like increment) must always be atomic.

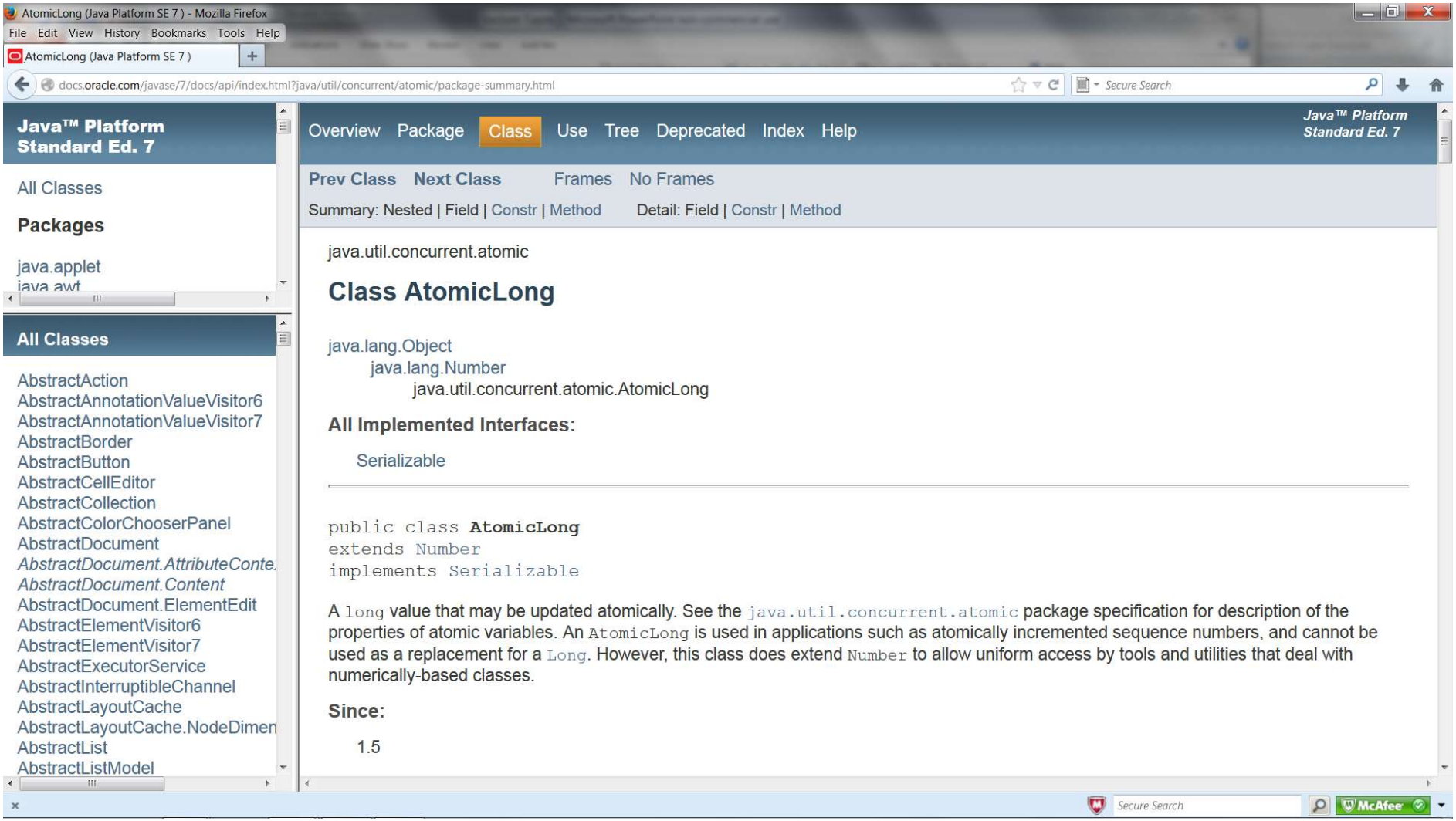
- We refer to **check-then-act** and **read-modify-write** sequences as *compound actions*: sequences of operations that ***must be executed atomically in order to remain thread-safe***.

- 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

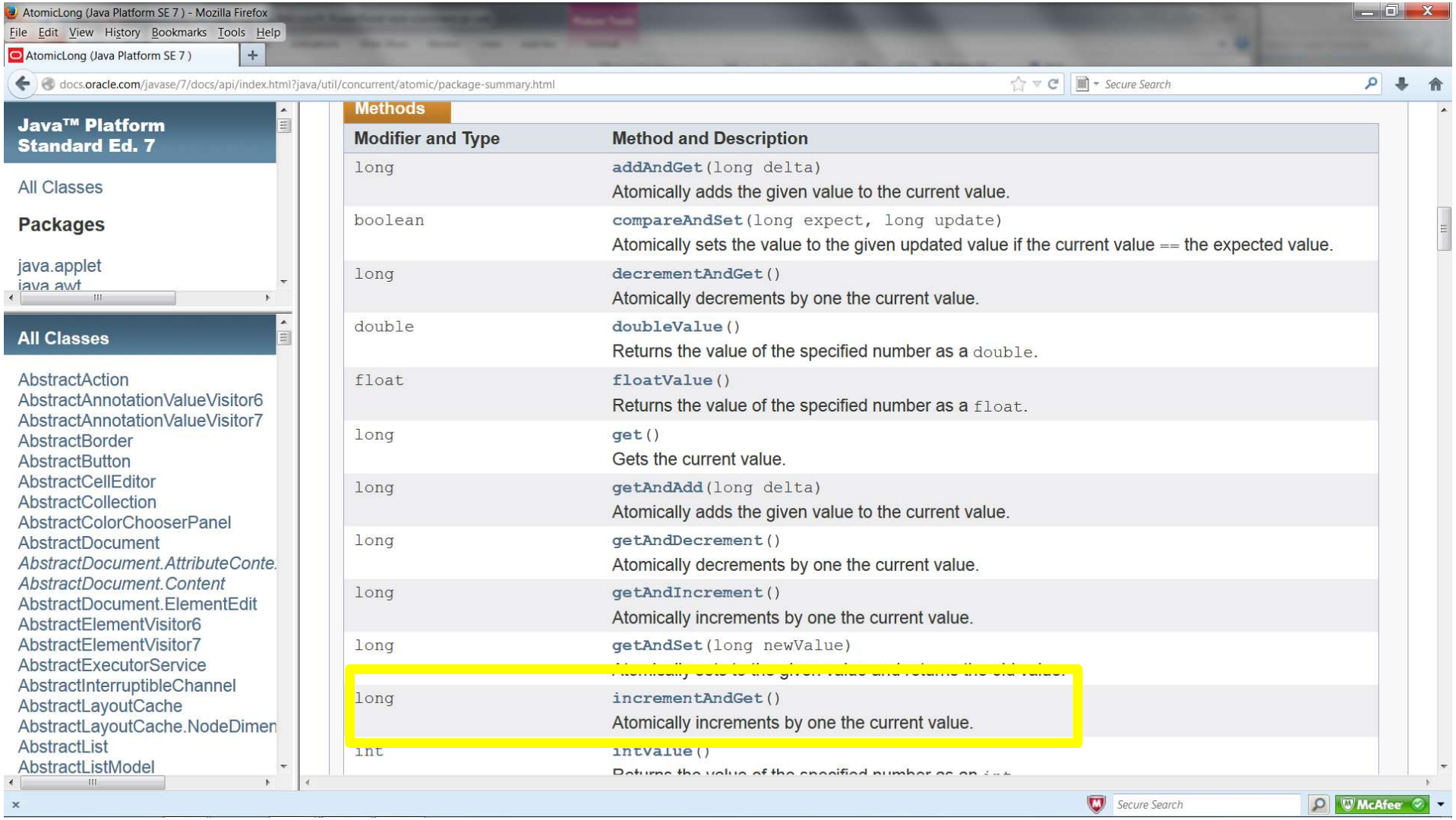
Thread Safety



Thread Safety



Thread Safety



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

- 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.

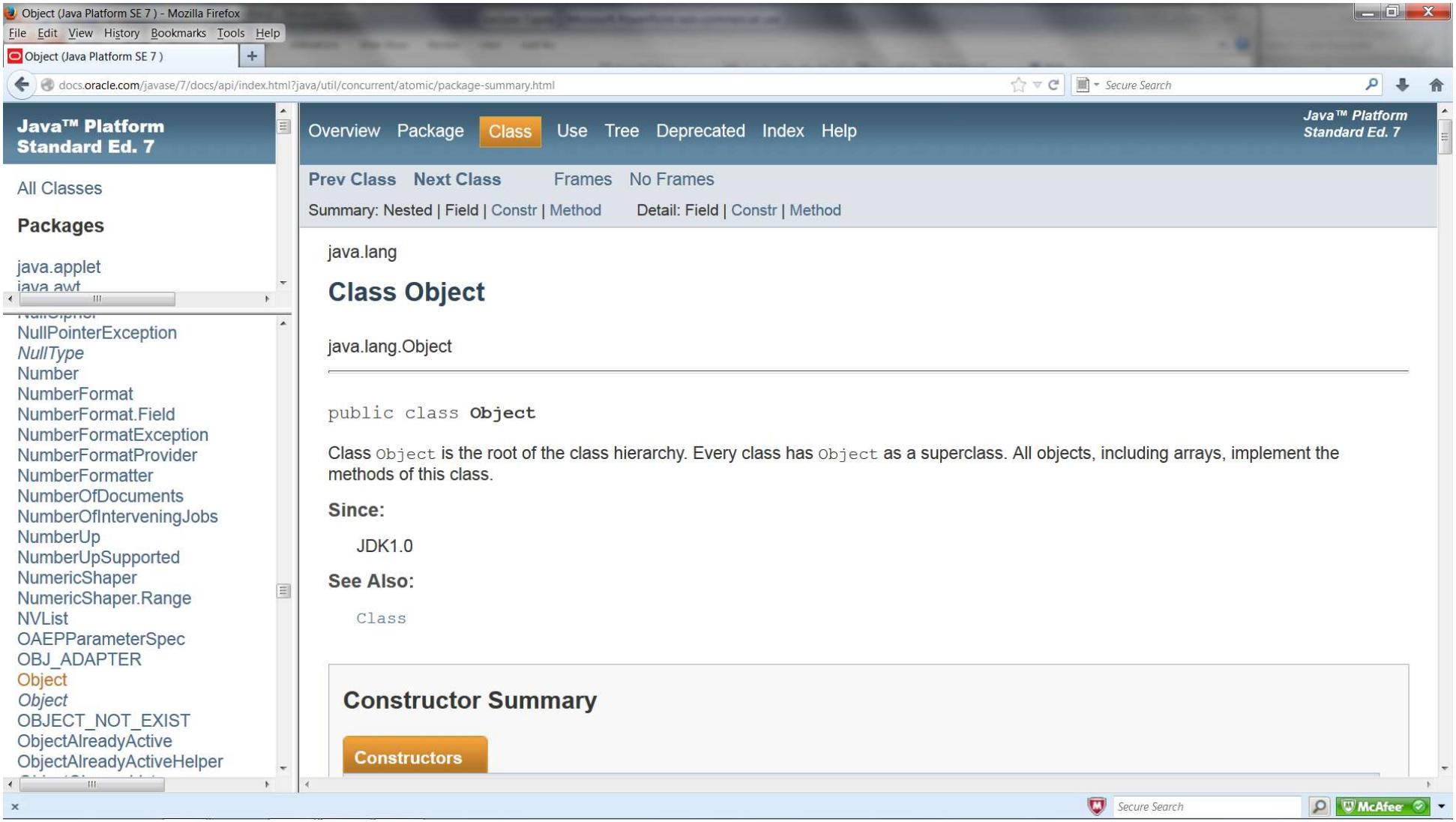
```
synchronized (lock) {
// Access or modify shared state guarded by lock
}
```

- Every Java object can implicitly act as a lock for purposes of synchronization

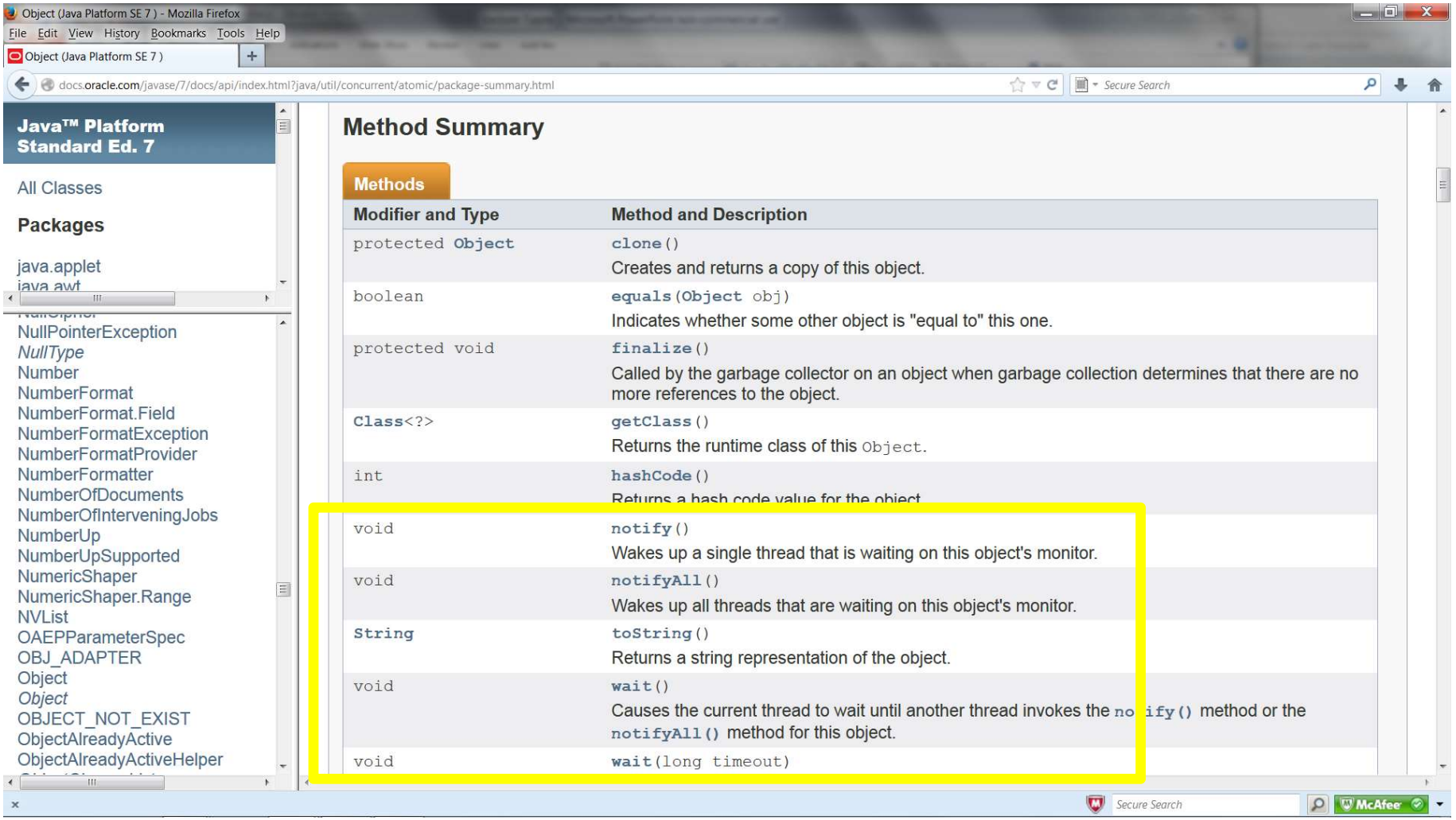
- These built-in locks are called *intrinsic locks* or *monitor locks*.

- 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



Thread Safety



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.

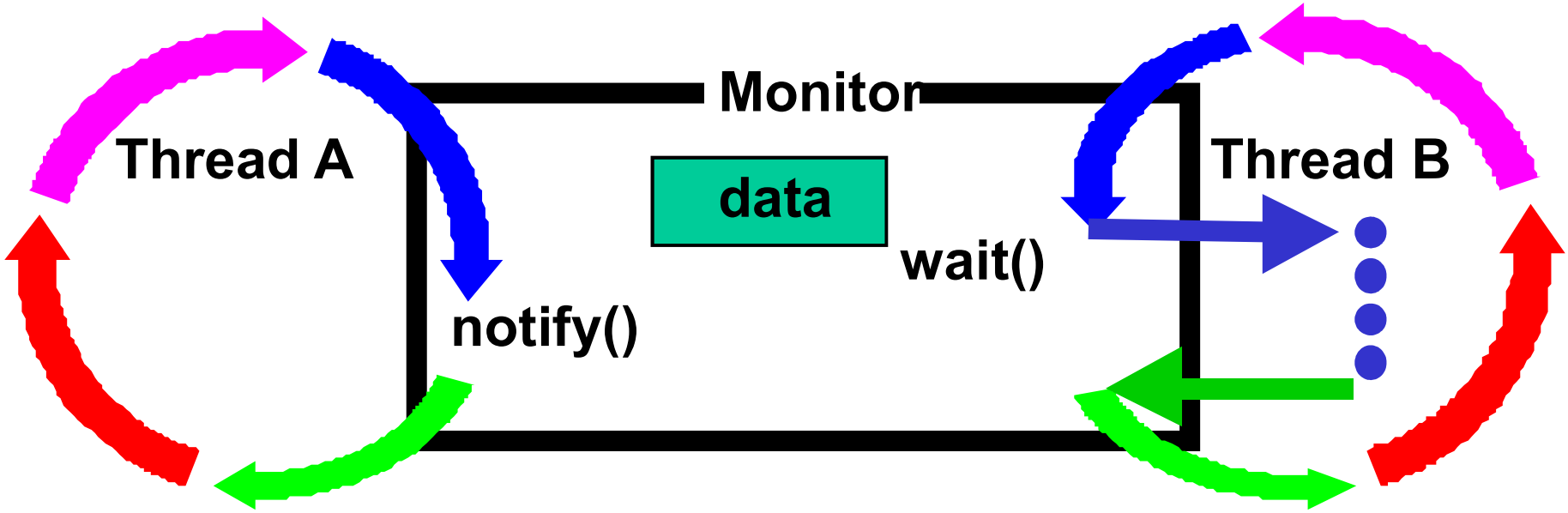
```
public final void wait()
                    throws InterruptedException
```

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 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();
  }
```