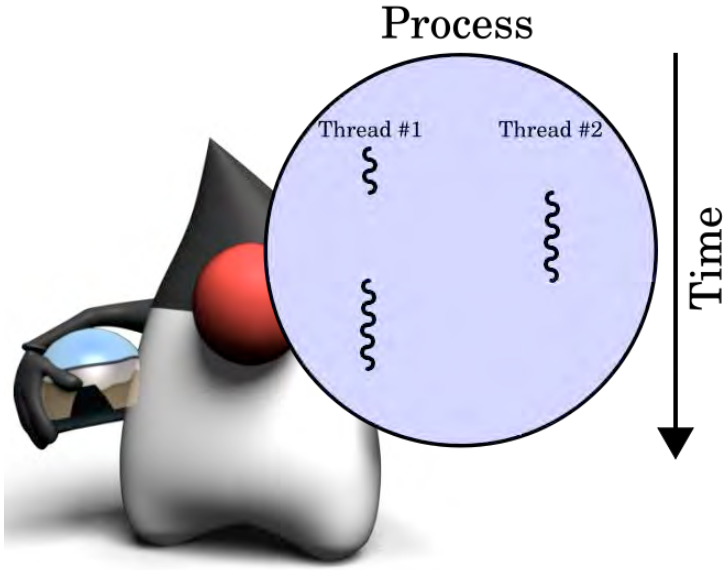
Java as a Second Language

Week 8: Threads



CS3773
Simson Garfinkel

This week:

- Threads
- Walls /Othello/Desktop Search

Week 8 (Feb 25) - Threads
Week 9 (March 3) - Open 1
Week 10 (March 10) - Open 2
Week 11 (March 17) - Open 3
March 27* - Final Project Due

Final Projects

By now, you should know what your final project is.

If you don't, email me a proposal ***today***.

Class Survey

You were sent a link to a SurveyMonkey survey.

Please fill it out! We will be using the survey results to decide the content of the next 4 weeks.

From the results so far, my plans are to:

- Cover threads & synchornization (Monday & Tuesday)
- Cover the "Solaris Device Drivers in Java" article.
- Some specific examples of reading & writing files
- Build a chat program using Othello server from start-to-finish

Modern computers have 5 different memory systems. The larger the memory, the slower it runs.

Architectural Registers

EAX
EBX
ECX
EDX
ESI
EDI

L1 Cache

L2 Cache

Main Memory

Disks

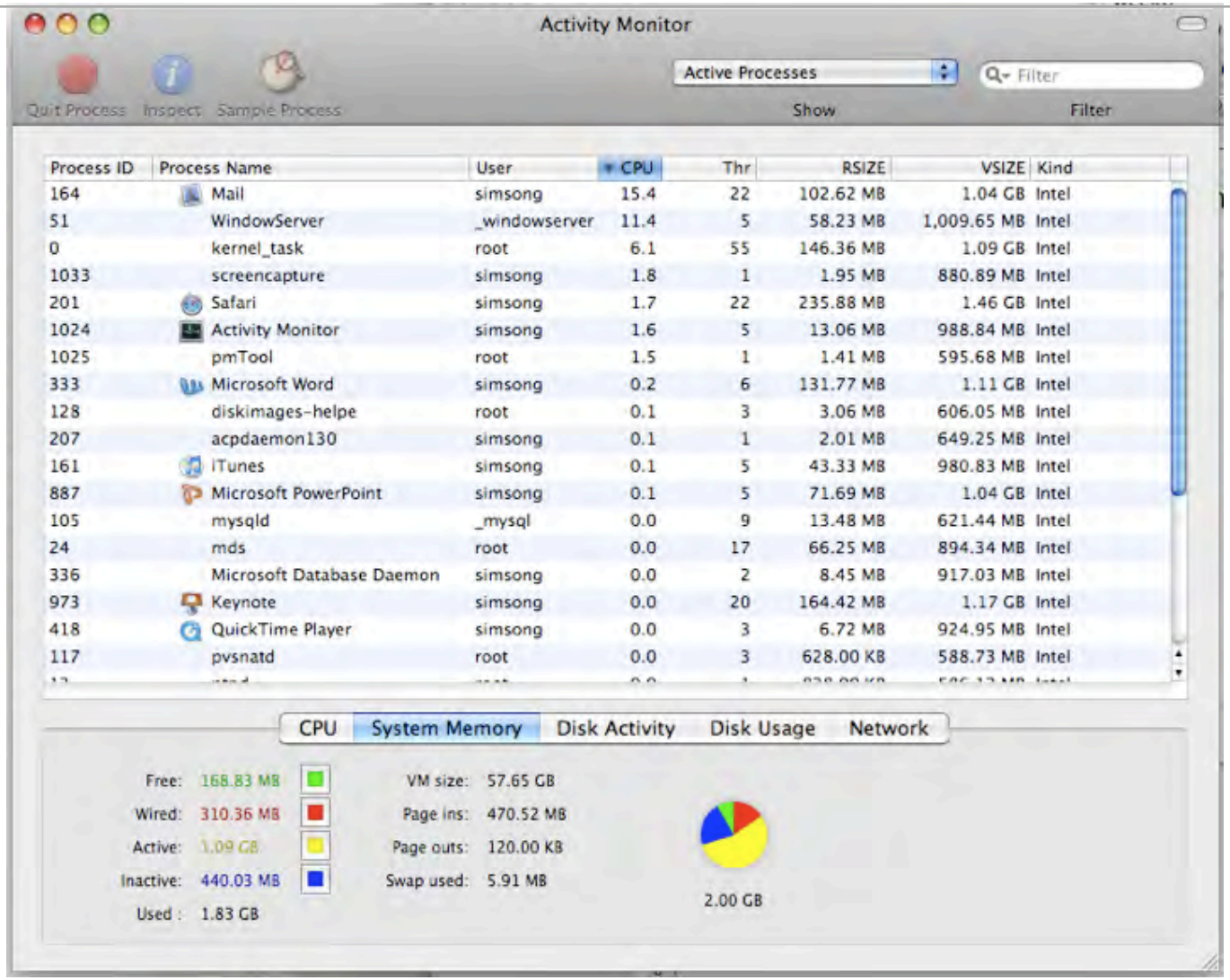
Active Register File

R001
R002
R003
R004
...
Rnnn

Increasing Storage Capacity

Increasing Speed

Modern computers run many different programs at the same time.



Each program's history can be can be thought of as a "thread" of connected events

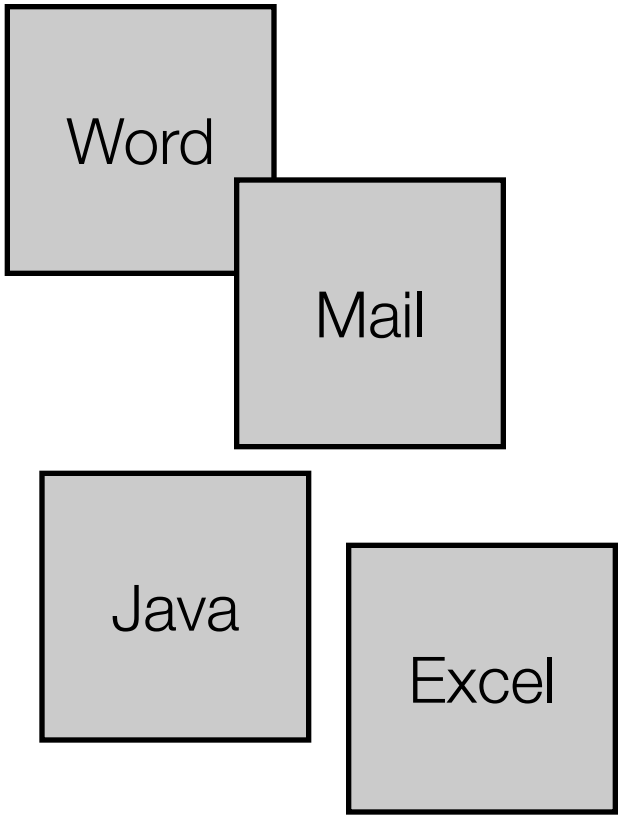
- **Receive File->Open**
- **Open File**
- **Read File into memory**
- **Create main window**
- **Repeat:**
  - **Wait for event**
  - **Update screen**

- **Read Mailboxes**
- **Create main window**
- **Repeat:**
  - **Check for mail**
  - **If mail, load into mailbox**
  - **check for keyboard/mouse event**
  - **Update screen**

Computers follow a *basic strategy* for making the most efficient use of their resources.

1. Find a program that's *ready to run.*
   - Microsoft Word
2. Run it until it *blocks*:
   - Request for a keyboard event.
   - Request to *read* or *write* to the disk
   - Request for information from the network
   - Used too much CPU time (1 sec)
3. When the program blocks:
   - Save what the program was doing.
   - Find another program to run (goto step 1)

This is called a ***scheduling policy***.



This same scheduling policy can be applied to tasks within a program

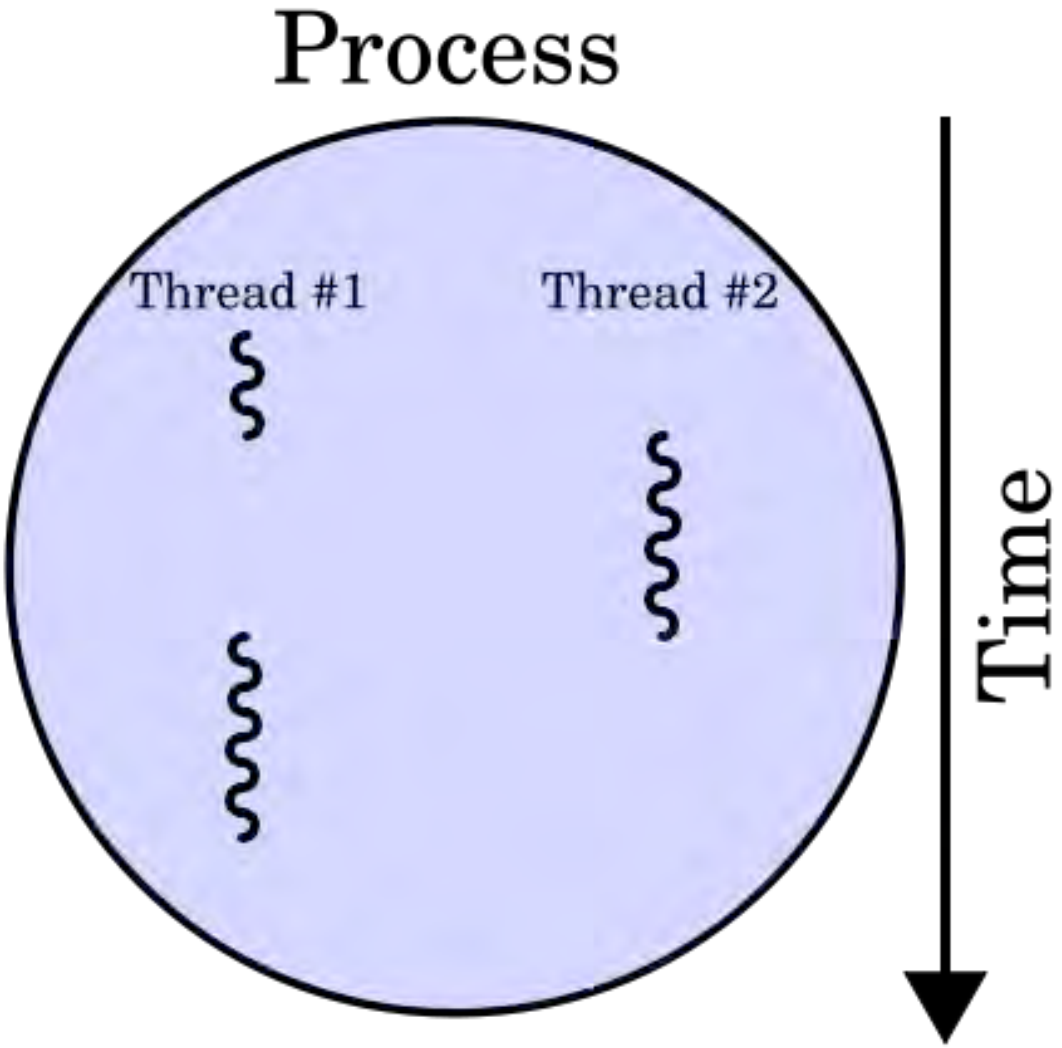
- **Read Mailboxes**
- **Create main window**
- **Repeat:**
  - **Check for mail**
  - **If mail, load into mailbox**
  - **Check for keyboard/mouse event**
  - **Update screen**

- **Check for mail**
- **If mail, load into mailbox**

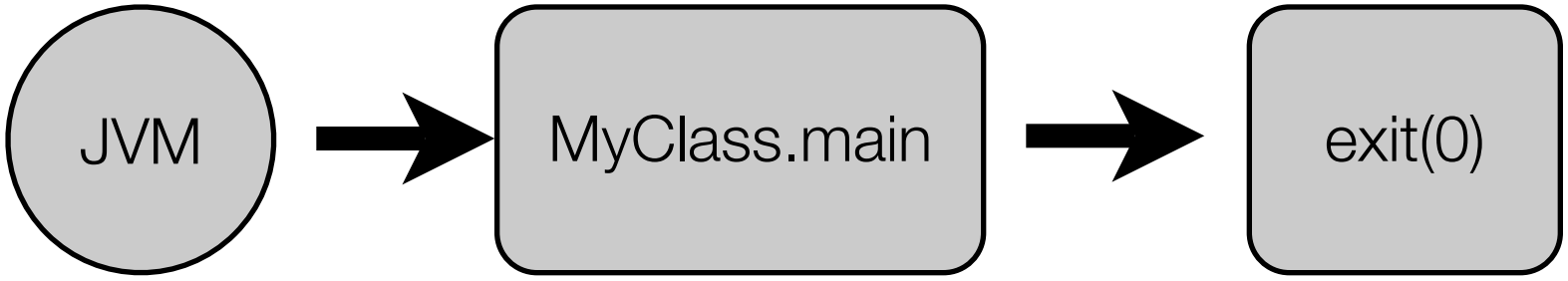
- **Check for keyboard/mouse event**
- **Process the event**

- **Update screen**

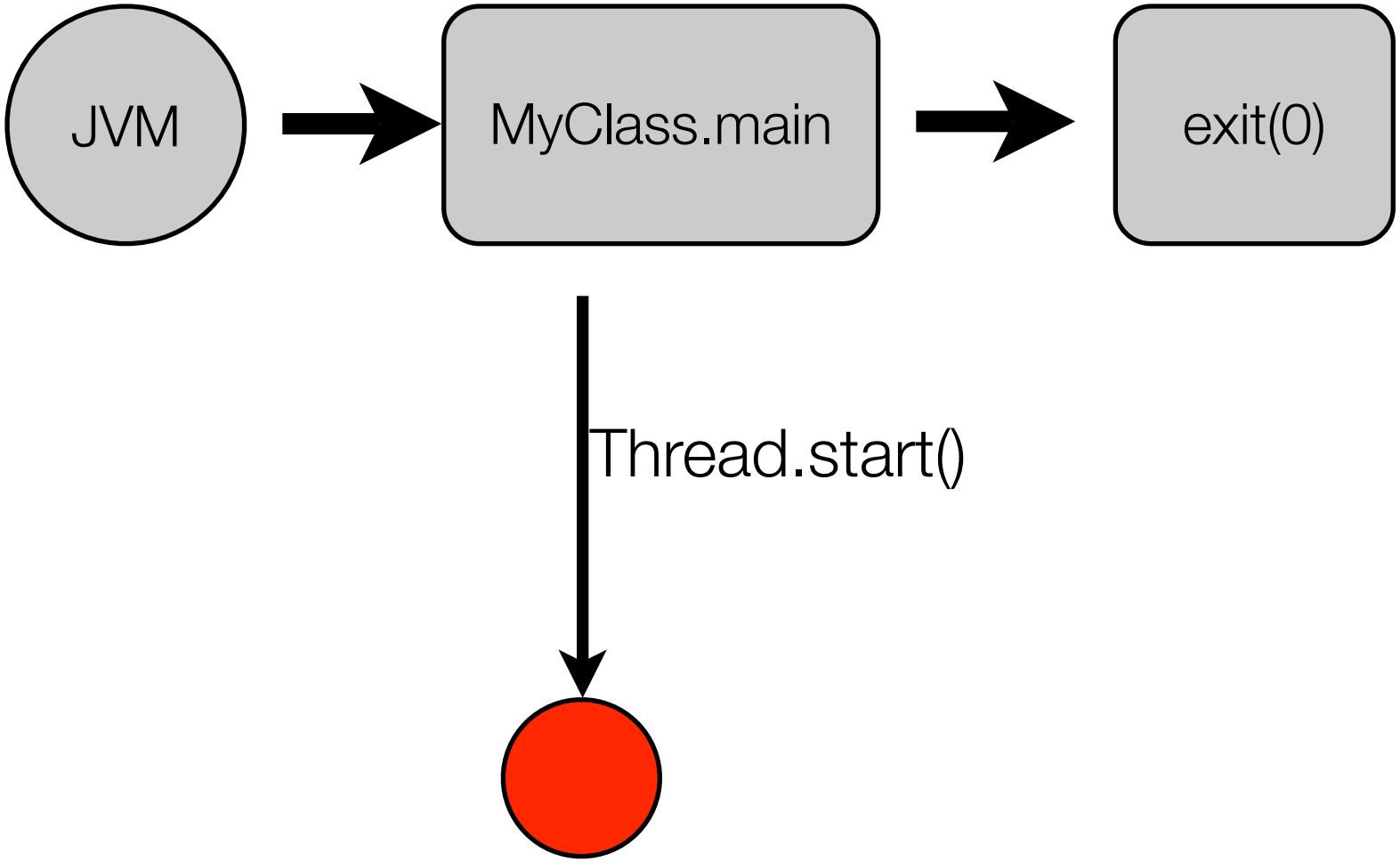
Producing this "classic" view of threads inside a computer:



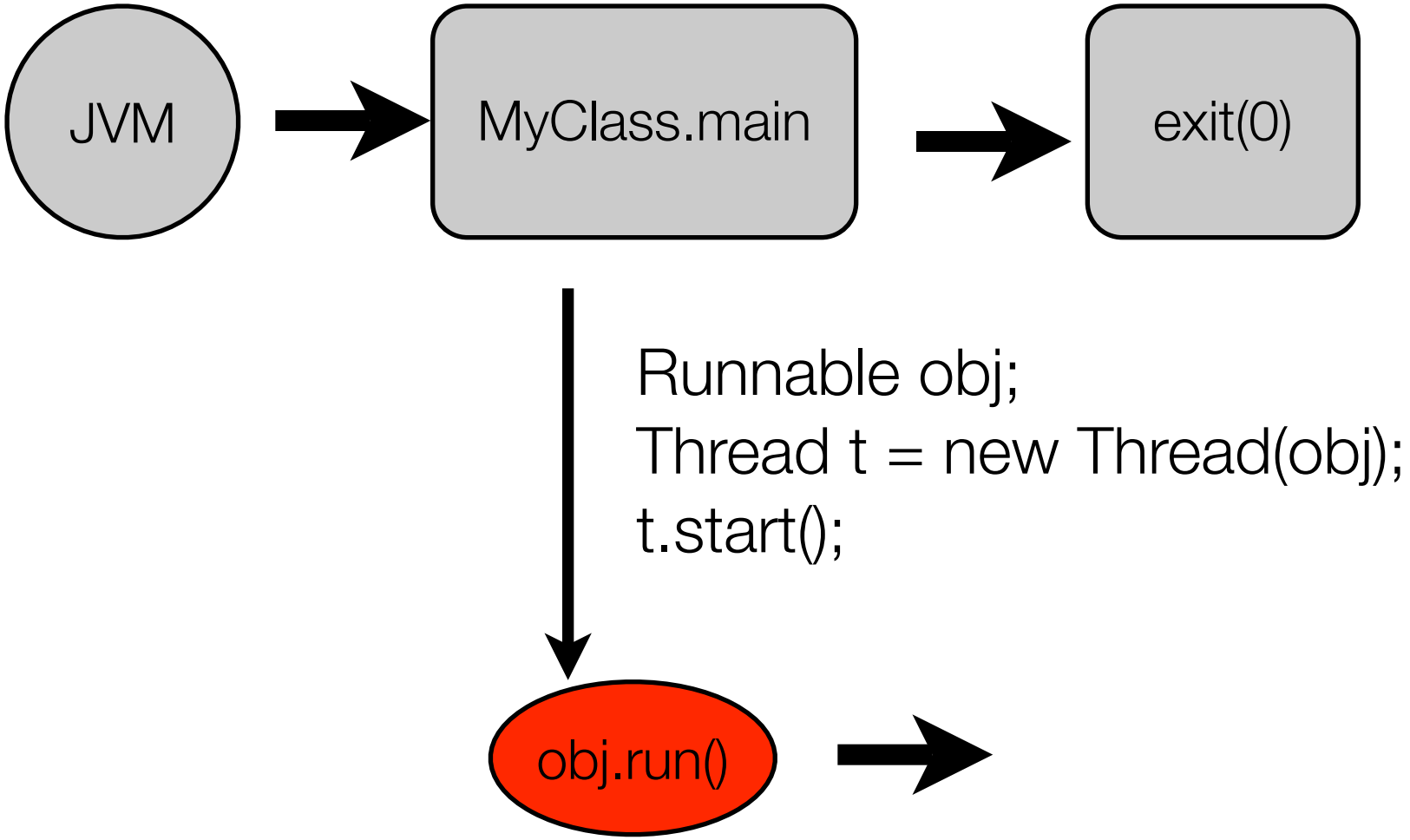
Until now, we have only had a *single* thread of execution in our programs (except for the Flatland bug.)



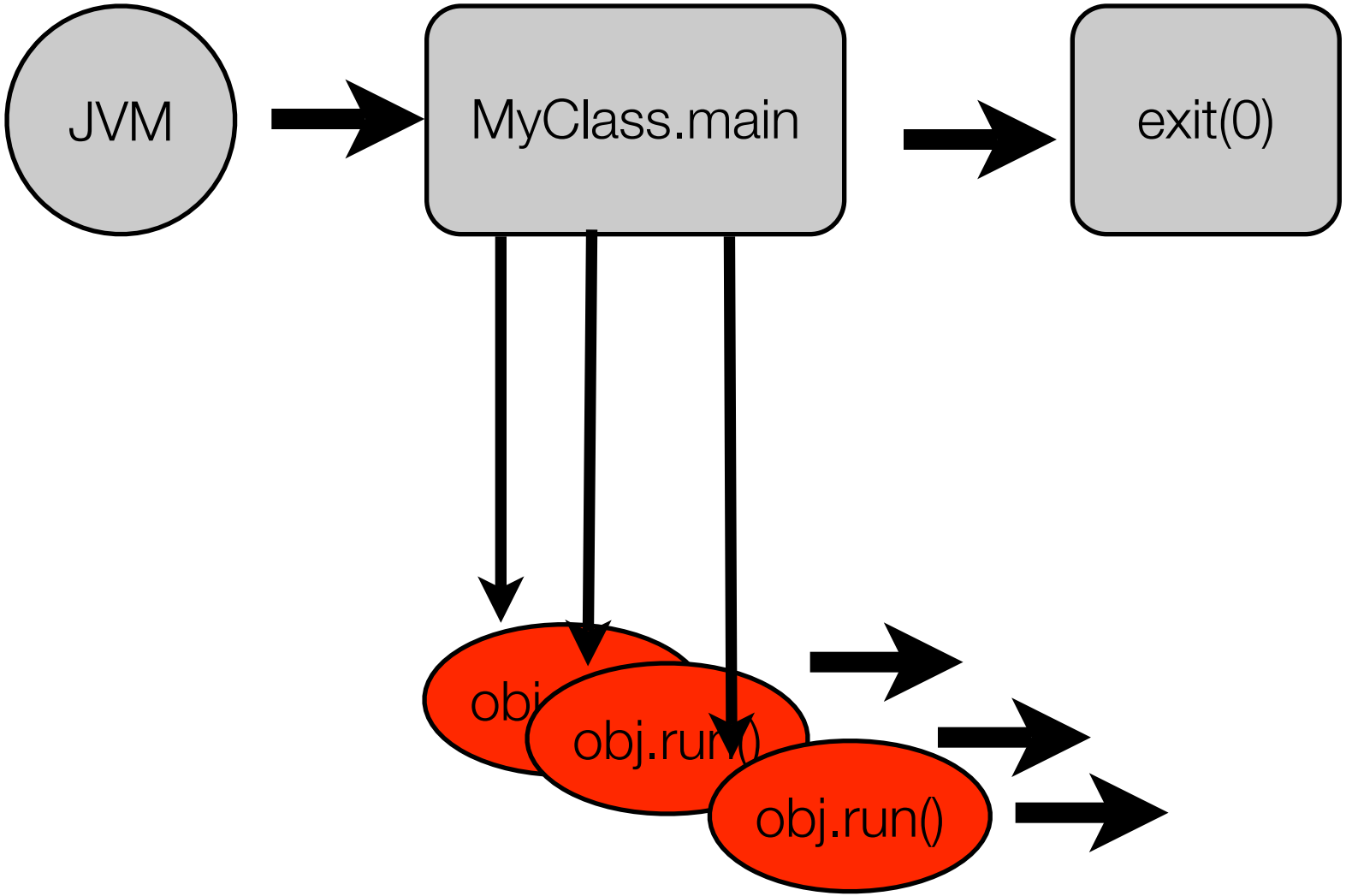
The Thread class lets you create multiple threads of execution



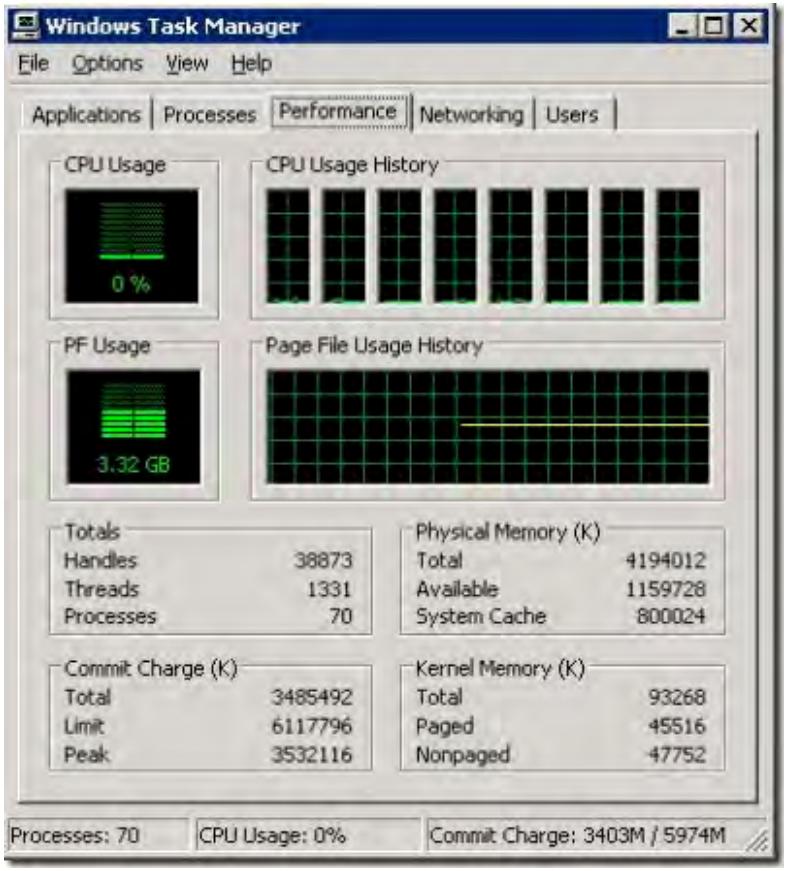
The Thread class lets you create multiple threads of execution for any object that implements Runnable.

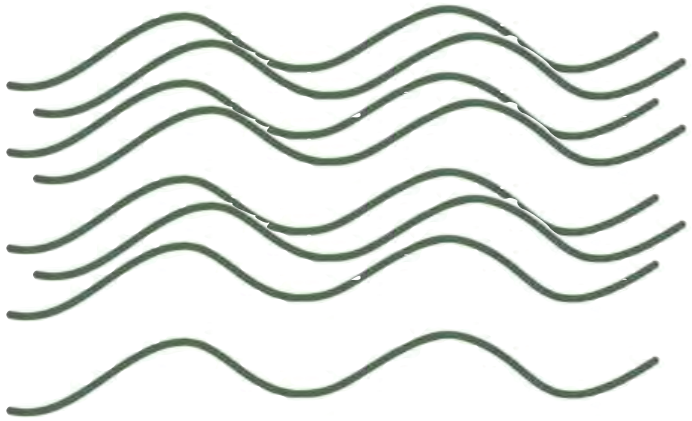


You can make *as many threads as you want.*
You *cannot control when they run.*



On a multiprocessor (or multi-core machine), multiple threads can literally run at the same time





Let's look at some code: ThreadDemo implements Runnable

```
class ThreadDemo implements Runnable {
    String name;
    ThreadDemo(String name){
        this.name = name;
    }
    public void run(){
        for(int i=0;i<10;i++){
            System.out.printf("Thread '%s' i=%d %n",name,i);
        }
    }
    public static void main(String[] args){

        Thread t;

        t = new Thread(new ThreadDemo("first"));
        t.start();

        t = new Thread(new ThreadDemo("second"));
        t.start();
    }
}
```

Now, let's try it with threads.
Run #1

```
09:01 PM Obsidian:~/current/cs3773/week8$ java ThreadDemo
Thread 'first' i=0
Thread 'second' i=0
Thread 'second' i=1
Thread 'second' i=2
Thread 'second' i=3
Thread 'second' i=4
Thread 'second' i=5
Thread 'second' i=6
Thread 'second' i=7
Thread 'second' i=8
Thread 'second' i=9
Thread 'first' i=1
Thread 'first' i=2
Thread 'first' i=3
Thread 'first' i=4
Thread 'first' i=5
Thread 'first' i=6
Thread 'first' i=7
Thread 'first' i=8
Thread 'first' i=9
09:01 PM Obsidian:~/current/cs3773/week8$
```

Now, let's try it with threads.
Run #2

```
09:01 PM Obsidian:~/current/cs3773/week8$ java ThreadDemo
Thread 'first' i=0
Thread 'first' i=1
Thread 'first' i=2
Thread 'first' i=3
Thread 'first' i=4
Thread 'first' i=5
Thread 'first' i=6
Thread 'first' i=7
Thread 'first' i=8
Thread 'first' i=9
Thread 'second' i=0
Thread 'second' i=1
Thread 'second' i=2
Thread 'second' i=3
Thread 'second' i=4
Thread 'second' i=5
Thread 'second' i=6
Thread 'second' i=7
Thread 'second' i=8
Thread 'second' i=9
09:01 PM Obsidian:~/current/cs3773/week8$
```

Now, let's try it with threads.
Run #3

```
09:01 PM Obsidian:~/current/cs3773/week8$ java ThreadDemo
Thread 'first' i=0
Thread 'first' i=1
Thread 'first' i=2
Thread 'second' i=0
Thread 'second' i=1
Thread 'second' i=2
Thread 'second' i=3
Thread 'second' i=4
Thread 'second' i=5
Thread 'second' i=6
Thread 'second' i=7
Thread 'second' i=8
Thread 'second' i=9
Thread 'first' i=3
Thread 'first' i=4
Thread 'first' i=5
Thread 'first' i=6
Thread 'first' i=7
Thread 'first' i=8
Thread 'first' i=9
09:01 PM Obsidian:~/current/cs3773/week8$
```

Now, let's try it with threads.
Run #4

```
09:01 PM Obsidian:~/current/cs3773/week8$ java ThreadDemo
Thread 'first' i=0
Thread 'second' i=0
Thread 'second' i=1
Thread 'second' i=2
Thread 'second' i=3
Thread 'second' i=4
Thread 'second' i=5
Thread 'second' i=6
Thread 'second' i=7
Thread 'second' i=8
Thread 'second' i=9
Thread 'first' i=1
Thread 'first' i=2
Thread 'first' i=3
Thread 'first' i=4
Thread 'first' i=5
Thread 'first' i=6
Thread 'first' i=7
Thread 'first' i=8
Thread 'first' i=9
09:01 PM Obsidian:~/current/cs3773/week8$
```

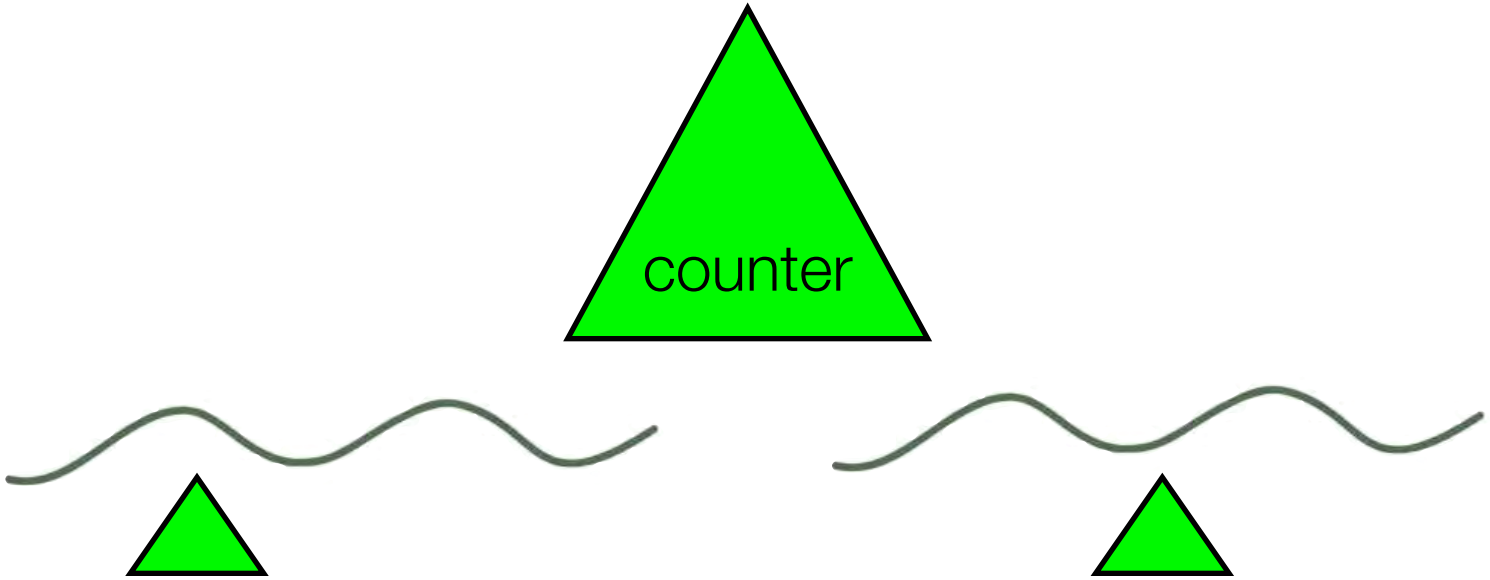
Now, let's try it with threads.
Run #5

```
09:01 PM Obsidian:~/current/cs3773/week8$ java ThreadDemo
Thread 'first' i=0
Thread 'first' i=1
Thread 'first' i=2
Thread 'first' i=3
Thread 'first' i=4
Thread 'first' i=5
Thread 'first' i=6
Thread 'first' i=7
Thread 'first' i=8
Thread 'first' i=9
Thread 'second' i=0
Thread 'second' i=1
Thread 'second' i=2
Thread 'second' i=3
Thread 'second' i=4
Thread 'second' i=5
Thread 'second' i=6
Thread 'second' i=7
Thread 'second' i=8
Thread 'second' i=9
09:01 PM Obsidian:~/current/cs3773/week8$
```

Now, let's try it with threads.
Run #6

```
09:01 PM Obsidian:~/current/cs3773/week8$ java ThreadDemo
Thread 'first' i=0
Thread 'second' i=0
Thread 'first' i=1
Thread 'second' i=1
Thread 'first' i=2
Thread 'second' i=2
Thread 'first' i=3
Thread 'second' i=3
Thread 'first' i=4
Thread 'second' i=4
Thread 'first' i=5
Thread 'second' i=5
Thread 'first' i=6
Thread 'second' i=6
Thread 'first' i=7
Thread 'second' i=7
Thread 'first' i=8
Thread 'second' i=8
Thread 'first' i=9
Thread 'second' i=9
09:01 PM Obsidian:~/current/cs3773/week8$
```

If two threads try to access the same variable at the same time, the results are unpredicatable.



The thread may do some work in a local variable.
The thread may cache and then write back results

Now both threads are incrementing "counter." What will happen when the threads run together?

```
class ThreadDemo implements Runnable {
    static int counter = 0;
    String name;
    ThreadDemo(String name){
        this.name = name;
    }

    public void run(){
        for(int i=0;i<10;i++){
            counter += 1;
            System.out.printf("Thread '%s' i=%d counter=%d %n",
                              name,i,counter);
        }
    }
    public static void main(String[] args){

        (new Thread(new ThreadDemo("first"))).start();
        (new Thread(new ThreadDemo("second"))).start();
    }
}
```

Run #1: Counter reaches 20…
… but where is counter=1? How about 14?

```
09:06 PM Obsidian:~/current/cs3773/week8$ java ThreadDemo
Thread 'first' i=0 counter=2
Thread 'second' i=0 counter=2
Thread 'first' i=1 counter=3
Thread 'second' i=1 counter=4
Thread 'first' i=2 counter=5
Thread 'second' i=2 counter=6
Thread 'first' i=3 counter=7
Thread 'second' i=3 counter=8
Thread 'first' i=4 counter=9
Thread 'second' i=4 counter=10
Thread 'first' i=5 counter=11
Thread 'second' i=5 counter=12
Thread 'first' i=6 counter=13
Thread 'first' i=7 counter=15
Thread 'first' i=8 counter=16
Thread 'first' i=9 counter=17
Thread 'second' i=6 counter=14
Thread 'second' i=7 counter=18
Thread 'second' i=8 counter=19
Thread 'second' i=9 counter=20
09:06 PM Obsidian:~/current/cs3773/week8$
```

Run #2: Now counter=14 is in the right place

```
9:06 PM Obsidian:~/current/cs3773/week8$ java ThreadDemo
Thread 'first' i=0 counter=2
Thread 'second' i=0 counter=2
Thread 'first' i=1 counter=3
Thread 'second' i=1 counter=4
Thread 'first' i=2 counter=5
Thread 'second' i=2 counter=6
Thread 'first' i=3 counter=7
Thread 'second' i=3 counter=8
Thread 'first' i=4 counter=9
Thread 'second' i=4 counter=10
Thread 'first' i=5 counter=11
Thread 'second' i=5 counter=12
Thread 'first' i=6 counter=13
Thread 'second' i=6 counter=14
Thread 'first' i=7 counter=15
Thread 'second' i=7 counter=16
Thread 'first' i=8 counter=17
Thread 'second' i=8 counter=18
Thread 'first' i=9 counter=19
Thread 'second' i=9 counter=20
09:06 PM Obsidian:~/current/cs3773/week8$
```

Run #2: Now counter=14 is in the right place

```
9:06 PM Obsidian:~/current/cs3773/week8$ java ThreadDemo
Thread 'first' i=0 counter=2
Thread 'second' i=0 counter=2
Thread 'first' i=1 counter=3
Thread 'second' i=1 counter=4
Thread 'first' i=2 counter=5
Thread 'second' i=2 counter=6
Thread 'first' i=3 counter=7
Thread 'second' i=3 counter=8
Thread 'first' i=4 counter=9
Thread 'second' i=4 counter=10
Thread 'first' i=5 counter=11
Thread 'second' i=5 counter=12
Thread 'first' i=6 counter=13
Thread 'second' i=6 counter=14
Thread 'first' i=7 counter=15
Thread 'second' i=7 counter=16
Thread 'first' i=8 counter=17
Thread 'second' i=8 counter=18
Thread 'first' i=9 counter=19
Thread 'second' i=9 counter=20
09:06 PM Obsidian:~/current/cs3773/week8$
```

Run #3:

```
09:06 PM Obsidian:~/current/cs3773/week8$ java ThreadDemo
Thread 'first' i=0 counter=2
Thread 'second' i=0 counter=2
Thread 'second' i=1 counter=4
Thread 'second' i=2 counter=5
Thread 'second' i=3 counter=6
Thread 'second' i=4 counter=7
Thread 'second' i=5 counter=8
Thread 'second' i=6 counter=9
Thread 'second' i=7 counter=10
Thread 'second' i=8 counter=11
Thread 'second' i=9 counter=12
Thread 'first' i=1 counter=3
Thread 'first' i=2 counter=13
Thread 'first' i=3 counter=14
Thread 'first' i=4 counter=15
Thread 'first' i=5 counter=16
Thread 'first' i=6 counter=17
Thread 'first' i=7 counter=18
Thread 'first' i=8 counter=19
Thread 'first' i=9 counter=20
09:06 PM Obsidian:~/current/cs3773/week8$
```

The synchronized() statement creates a **lock**. Only one thread can lock an object at a time.

```
Object obj = new Object();

synchronized (obj) {
    // This code is locked.
    // only one thread can be in this block at a time.
    // provided that all threads share 'obj'
}
```

```
class ThreadDemo implements Runnable {
    static Object lock = new Object();
    static int counter = 0;
    String name;
    ThreadDemo(String name){
        this.name = name;
    }

    public void run(){
        for(int i=0;i<10;i++){
            synchronized (lock) {
                counter += 1;
                System.out.printf("Thread '%s' i=%d counter=%d %n",
                                  name,i,counter);
            }
        }
    }
    public static void main(String[] args){

        (new Thread(new ThreadDemo("first"))).start();
        (new Thread(new ThreadDemo("second"))).start();
    }
}
```

Synchronized Run #1:

```
09:21 PM Obsidian:~/current/cs3773/week8$ java ThreadDemo
Thread 'first' i=0 counter=1
Thread 'second' i=0 counter=2
Thread 'first' i=1 counter=3
Thread 'second' i=1 counter=4
Thread 'first' i=2 counter=5
Thread 'second' i=2 counter=6
Thread 'first' i=3 counter=7
Thread 'second' i=3 counter=8
Thread 'first' i=4 counter=9
Thread 'second' i=4 counter=10
Thread 'first' i=5 counter=11
Thread 'second' i=5 counter=12
Thread 'first' i=6 counter=13
Thread 'second' i=6 counter=14
Thread 'first' i=7 counter=15
Thread 'second' i=7 counter=16
Thread 'first' i=8 counter=17
Thread 'second' i=8 counter=18
Thread 'first' i=9 counter=19
Thread 'second' i=9 counter=20
09:21 PM Obsidian:~/current/cs3773/week8$
```

Synchronized Run #2:

```
09:21 PM Obsidian:~/current/cs3773/week8$ java ThreadDemo
Thread 'first' i=0 counter=1
Thread 'second' i=0 counter=2
Thread 'second' i=1 counter=3
Thread 'second' i=2 counter=4
Thread 'second' i=3 counter=5
Thread 'second' i=4 counter=6
Thread 'second' i=5 counter=7
Thread 'second' i=6 counter=8
Thread 'second' i=7 counter=9
Thread 'second' i=8 counter=10
Thread 'second' i=9 counter=11
Thread 'first' i=1 counter=12
Thread 'first' i=2 counter=13
Thread 'first' i=3 counter=14
Thread 'first' i=4 counter=15
Thread 'first' i=5 counter=16
Thread 'first' i=6 counter=17
Thread 'first' i=7 counter=18
Thread 'first' i=8 counter=19
Thread 'first' i=9 counter=20
09:21 PM Obsidian:~/current/cs3773/week8$
```

Synchronized Run #3:

```
09:22 PM Obsidian:~/current/cs3773/week8$ java ThreadDemo
Thread 'first' i=0 counter=1
Thread 'second' i=0 counter=2
Thread 'first' i=1 counter=3
Thread 'second' i=1 counter=4
Thread 'second' i=2 counter=5
Thread 'second' i=3 counter=6
Thread 'first' i=2 counter=7
Thread 'second' i=4 counter=8
Thread 'first' i=3 counter=9
Thread 'second' i=5 counter=10
Thread 'first' i=4 counter=11
Thread 'second' i=6 counter=12
Thread 'first' i=5 counter=13
Thread 'second' i=7 counter=14
Thread 'first' i=6 counter=15
Thread 'second' i=8 counter=16
Thread 'first' i=7 counter=17
Thread 'second' i=9 counter=18
Thread 'first' i=8 counter=19
Thread 'first' i=9 counter=20
09:22 PM Obsidian:~/current/cs3773/week8$
```

Methods can be synchronized as well.
Only one thread can run the method at a time.

```
class ThreadDemo implements Runnable {
    static int counter = 0;
    String name;
    ThreadDemo(String name){
        this.name = name;
    }

    public synchronized int incrementCounter()  {
        counter += 1;
        return counter;
    }

    public void run(){
        for(int i=0;i<10;i++){
            System.out.printf("Thread '%s' i=%d counter=%d %n",
                              name,i,incrementCounter());
        }
    }
    public static void main(String[] args){

        (new Thread(new ThreadDemo("first"))).start();
        (new Thread(new ThreadDemo("second"))).start();
    }
}
```

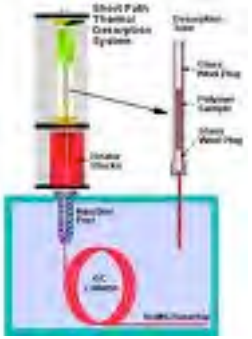
Notice that the synchronized method returned the counter value.

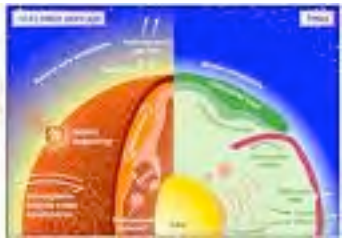
```
public synchronized int incrementCounter()  {
    counter += 1;
    return counter;
}

public void run(){
    for(int i=0;i<10;i++){
        System.out.printf("Thread '%s' i=%d counter=%d %n",
                          name,i,incrementCounter());
    }
}
```

run() shouldn't reference the same **counter** from multiple threads. Returning the *value* solves this potential problem.

Tuesday:
Volatile Variables

So what does **volatile** do?





keyword: volatile

The "volatile" keyword tells the compiler not to cache a variable.

Goetz calls them "synchronized lite."

Advantages:

- They aren't cached, so you always get a valid value.

Disadvantage:

- It might be an old value.
- Writes may not be updated properly.

Goetz "cheap read-write lock"

```
public class CheesyCounter {
    // Employs the cheap read-write lock trick
    // All mutative operations MUST be done
    // with the 'this' lock held
    private volatile int value;

    public int getValue() { return value; }

    public synchronized int increment() {
        return value++;
    }
}
```

Using CheesyCounter.java:

```
class ThreadDemo implements Runnable {
    static CheesyCounter counter = new CheesyCounter();
    String name;
    ThreadDemo(String name){
        this.name = name;
    }

    public void run(){
        for(int i=0;i<10;i++){
            counter.increment();
            System.out.printf("Thread '%s' i=%d counter=%d %n",
                              name,i,counter.getValue());
        }
    }
    public static void main(String[] args){

        (new Thread(new ThreadDemo("first"))).start();
        (new Thread(new ThreadDemo("second"))).start();
    }
}
```

CheesyCounter Run #1

```
07:22 PM imac2:~/current/cs3773/week8$ java ThreadDemo
Thread 'first' i=0 counter=2
Thread 'second' i=0 counter=2
Thread 'first' i=1 counter=3
Thread 'second' i=1 counter=4
Thread 'first' i=2 counter=5
Thread 'second' i=2 counter=6
Thread 'first' i=3 counter=7
Thread 'second' i=3 counter=8
Thread 'first' i=4 counter=9
Thread 'second' i=4 counter=10
Thread 'first' i=5 counter=11
Thread 'second' i=5 counter=12
Thread 'first' i=6 counter=13
Thread 'second' i=6 counter=14
Thread 'first' i=7 counter=15
Thread 'second' i=7 counter=16
Thread 'first' i=8 counter=17
Thread 'second' i=8 counter=18
Thread 'first' i=9 counter=19
Thread 'second' i=9 counter=20
07:22 PM imac2:~/current/cs3773/week8$ %
```

CheesyCounter Run #2

```
07:22 PM imac2:~/current/cs3773/week8$ java ThreadDemo
Thread 'first' i=0 counter=1
Thread 'second' i=0 counter=2
Thread 'first' i=1 counter=3
Thread 'first' i=2 counter=5
Thread 'first' i=3 counter=6
Thread 'second' i=1 counter=4
Thread 'first' i=4 counter=7
Thread 'first' i=5 counter=8
Thread 'first' i=6 counter=9
Thread 'first' i=7 counter=10
Thread 'second' i=2 counter=11
Thread 'first' i=8 counter=12
Thread 'second' i=3 counter=13
Thread 'second' i=4 counter=15
Thread 'second' i=5 counter=16
Thread 'second' i=6 counter=17
Thread 'second' i=7 counter=18
Thread 'first' i=9 counter=14
Thread 'second' i=8 counter=19
Thread 'second' i=9 counter=20
07:22 PM imac2:~/current/cs3773/week8$
```

But this will not work:

```
class ThreadDemo implements Runnable {
    static volatile int counter = 0;   /*** NOT GOOD ENOUGH ***/
    String name;
    ThreadDemo(String name){
        this.name = name;
    }

    public void run(){
        for(int i=0;i<10;i++){
       counter += 1;
            System.out.printf("Thread '%s' i=%d counter=%d %n",
                              name,i,counter);
        }
    }
    public static void main(String[] args){

   (new Thread(new ThreadDemo("first"))).start();
   (new Thread(new ThreadDemo("second"))).start();
    }
}
```

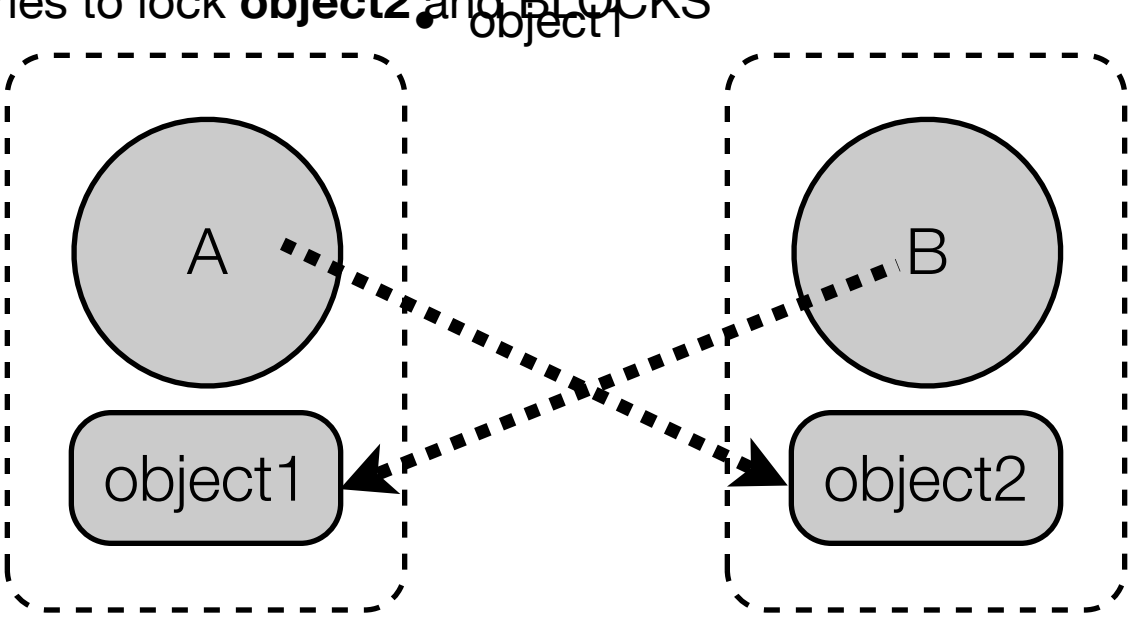
Run with "volatile counter"

```
07:21 PM imac2:~/current/cs3773/week8$ java ThreadDemo
Thread 'first' i=0 counter=2
Thread 'second' i=0 counter=2
Thread 'first' i=1 counter=3
Thread 'second' i=1 counter=4
Thread 'second' i=2 counter=6
Thread 'second' i=3 counter=7
Thread 'second' i=4 counter=8
Thread 'second' i=5 counter=9
Thread 'second' i=6 counter=10
Thread 'second' i=7 counter=11
Thread 'second' i=8 counter=12
Thread 'second' i=9 counter=13
Thread 'first' i=2 counter=5
Thread 'first' i=3 counter=14
Thread 'first' i=4 counter=15
Thread 'first' i=5 counter=16
Thread 'first' i=6 counter=17
Thread 'first' i=7 counter=18
Thread 'first' i=8 counter=19
Thread 'first' i=9 counter=20
07:21 PM imac2:~/current/cs3773/week8$
```

Another danger with threading: deadlocks

Deadlock scenario:

- Thread A locks **object1**
- Thread B locks **object2**
- Thread A tries to lock **object1** and BLOCKS.
- Thread B tries to lock **object2** and BLOCKS



DeadlyEmbrace.java

```
public class DeadlyEmbrace implements Runnable {
    Object object1, object2;
    String name;
    DeadlyEmbrace(String name,Object object1,Object object2){
        this.name = name; this.object1 = object1; this.object2 = object2;
    }

    public void run(){
        System.out.println("Thread "+name+" starting up...");
        synchronized (object1) {
            System.out.println("Thread "+name+" grabbed "+object1);
            synchronized (object2) {
                System.out.println("Thread "+name+" grabbed "+object2);
            }
        }
        System.out.println("Thread "+name+" shutting down...");
    }

    public static void main(String[] args){
        Object a = new Object();
        Object b = new Object();
        new Thread(new DeadlyEmbrace("first",a,b)).start();
        new Thread(new DeadlyEmbrace("second",b,a)).start();
    }
```

Let's try it.
DeadlyEmbrace Run #1

07:38 PM imac2:~/current/cs3773/week8$ java DeadlyEmbrace

Thread first starting up...

Thread first grabbed java.lang.Object@c20e24

Thread first grabbed java.lang.Object@2e7263

Thread first shutting down...

Thread second starting up...

Thread second grabbed java.lang.Object@2e7263

Thread second grabbed java.lang.Object@c20e24

Thread second shutting down...

07:38 PM imac2:~/current/cs3773/week8$

DeadlyEmbrace Run #2:

07:38 PM imac2:~/current/cs3773/week8$ java DeadlyEmbrace

Thread first starting up...

Thread second starting up...

Thread second grabbed java.lang.Object@2e7263

Thread first grabbed java.lang.Object@c20e24

Results of multiple runs

Results of multiple runs

Run 1 — OK

Results of multiple runs

Run 1 — OK

Run 2 — DEADLOCK

Results of multiple runs

Run 1 — OK

Run 2 — DEADLOCK

Run 3 — OK

Results of multiple runs

Run 1 — OK

Run 2 — DEADLOCK

Run 3 — OK

Run 4 — OK

Results of multiple runs

Run 1 — OK

Run 2 — DEADLOCK

Run 3 — OK

Run 4 — OK

Run 5 — OK

Results of multiple runs

Run 1 — OK

Run 2 — DEADLOCK

Run 3 — OK

Run 4 — OK

Run 5 — OK

Run 6 — OK

Results of multiple runs

Run 1 — OK

Run 2 — DEADLOCK

Run 3 — OK

Run 4 — OK

Run 5 — OK

Run 6 — OK

Run 7 — OK

Results of multiple runs

Run 1 — OK

Run 2 — DEADLOCK

Run 3 — OK

Run 4 — OK

Run 5 — OK

Run 6 — OK

Run 7 — OK

Run 8 — DEADLOCK

Results of multiple runs

Run 1 — OK

Run 2 — DEADLOCK

Run 3 — OK

Run 4 — OK

Run 5 — OK

Run 6 — OK

Run 7 — OK

Run 8 — DEADLOCK

Run 9 — OK

Results of multiple runs

Run 1 — OK

Run 2 — DEADLOCK

Run 3 — OK

Run 4 — OK

Run 5 — OK

Run 6 — OK

Run 7 — OK

Run 8 — DEADLOCK

Run 9 — OK

Run 10 — DEADLOCK

Results of multiple runs

Run 1 — OK

Run 2 — DEADLOCK

Run 3 — OK

Run 4 — OK

Run 5 — OK

Run 6 — OK

Run 7 — OK

Run 8 — DEADLOCK

Run 9 — OK

Run 10 — DEADLOCK

Run 11 — OK

Results of multiple runs

Run 1 — OK

Run 2 — DEADLOCK

Run 3 — OK

Run 4 — OK

Run 5 — OK

Run 6 — OK

Run 7 — OK

Run 8 — DEADLOCK

Run 9 — OK

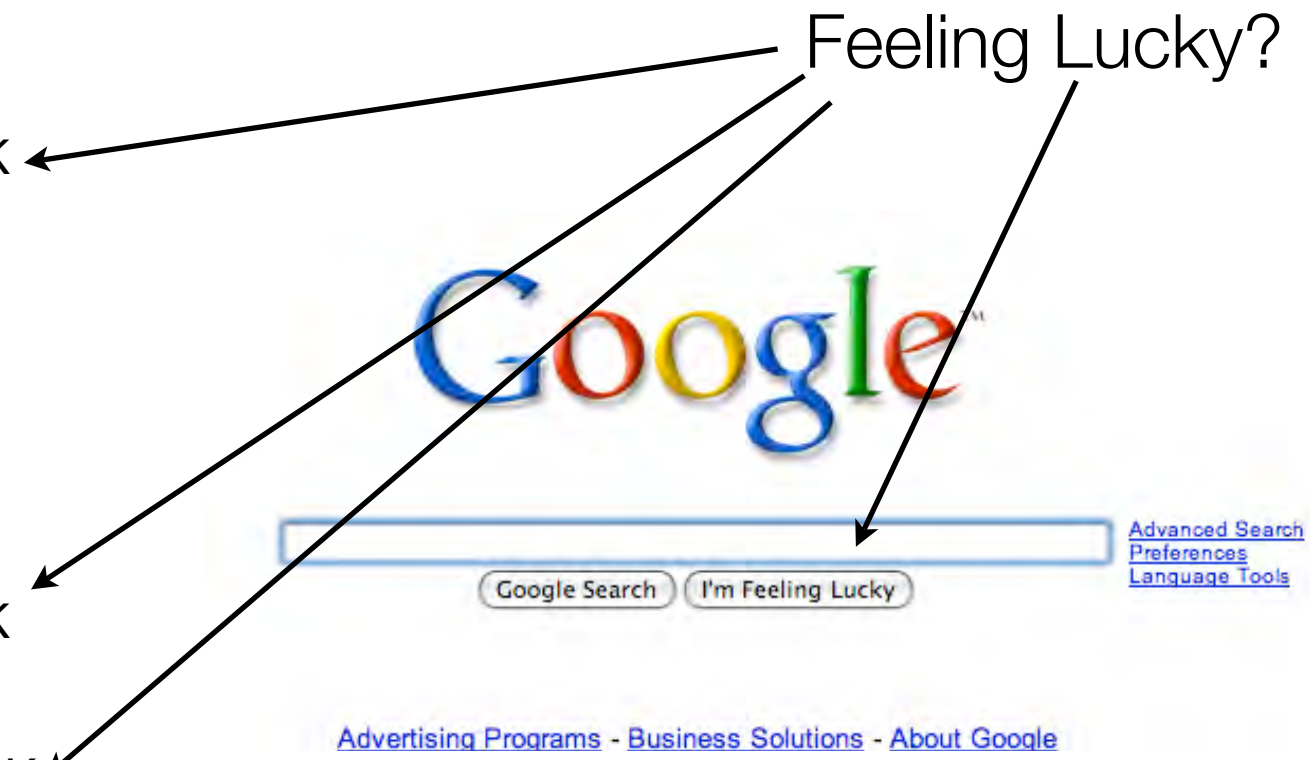
Run 10 — DEADLOCK

Run 11 — OK

Run 12 — OK

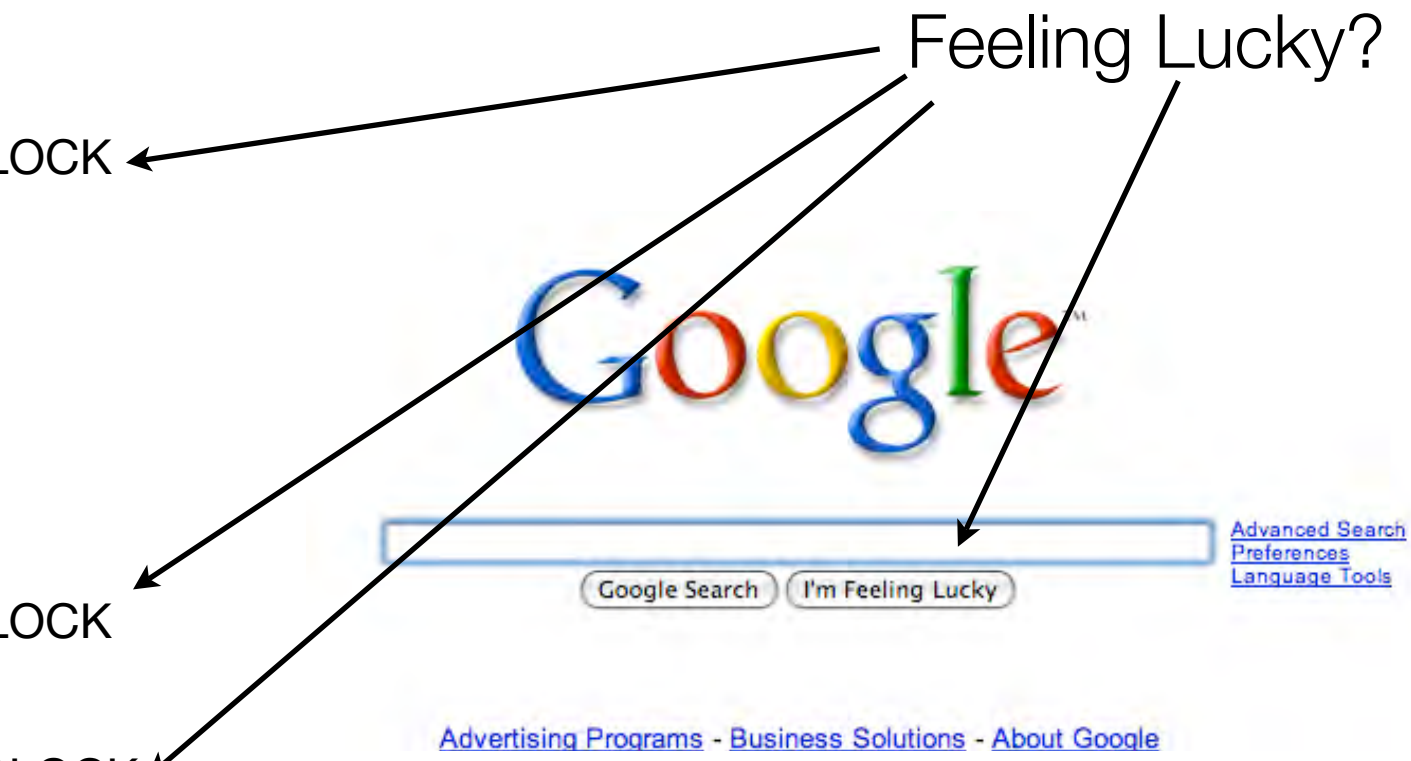
Results of multiple runs

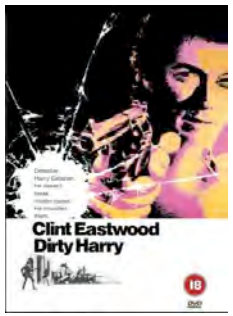
Run 1 — OK
Run 2 — DEADLOCK
Run 3 — OK
Run 4 — OK
Run 5 — OK
Run 6 — OK
Run 7 — OK
Run 8 — DEADLOCK
Run 9 — OK
Run 10 — DEADLOCK
Run 11 — OK
Run 12 — OK



Results of multiple runs

Run 1 — OK
Run 2 — DEADLOCK
Run 3 — OK
Run 4 — OK
Run 5 — OK
Run 6 — OK
Run 7 — OK
Run 8 — DEADLOCK
Run 9 — OK
Run 10 — DEADLOCK
Run 11 — OK
Run 12 — OK





I know what you're thinking. "Did he fire six shots or only five?" Well, to tell you the truth, in all this excitement I kind of lost track myself. But being as this is a .44 Magnum, the most powerful handgun in the world, and would blow your head clean off, you've got to ask yourself a question: Do I feel lucky?

You are responsible for these bold, underlined Java Reserved Words.

---

abstract	**do**	**import**	**short**	**volatile**
assert	**double**	**instanceof**	**static**	**while**
boolean	**else**	**int**	**strictfp**	
break	**enum**	**interface**	**super**	
byte	**extends**	**long**	**switch**	
case	**final**	**native**	**synchronized**	
catch	**finally**	**new**	**this**	
char	**float**	**package**	**throw**	
class	**for**	**private**	**throws**	
[const]	**[goto]**	**protected**	transient	
continue	**if**	**public**	**try**	
default	**implements**	**return**	**void**	

http://java.sun.com/docs/books/tutorial/java/nutsandbolts/_keywords.html

Serialization & Transient Variables

Java makes it easy to put data into a file.

java.io.Writer — Abstract class for writing to character streams

- Implements write(char[],int, int), flush() and close()

System.out — a java.io.PrintStream

```
java.lang.Object
  java.io.OutputStream
      java.io.FilterOutputStream
          java.io.PrintStream
```

java.io.FileWriter — Convenience class for writing character files

```
java.lang.Object
  java.io.Writer
      java.io.OutputStreamWriter
          java.io.FileWriter
```

Creating a file with a buffered writer:

```
import java.io.*;

public class Writer {
    public static void main(String[] args){
     try {
         FileWriter out =
               new FileWriter("outfile.txt");
         out.write("In the file the bytes go!\n");
         out.close();
     } catch (java.io.IOException e){
         e.printStackTrace();
     }
    }
}
```

You can be a little more efficient with BufferedWriter:

java.io.BufferedWriter — More efficient; saves up writes.

```
java.lang.Object
  java.io.Writer
      java.io.BufferedWriter
```

Creating a file with a BufferedWriter:

```
import java.io.*;

public class Writer {
    public static void main(String[] args){
     try {
         BufferedWriter out =
                 new BufferedWriter(
                      new FileWriter("outfile.txt"));
         out.write("In the file the bytes go!\n");
         out.close();
     } catch (java.io.IOException e){
         e.printStackTrace();
     }
    }
}
```

PrintWriter gives you additional methods for "printing"

append(char c)

printf()

println()

write(char[] buf)

write(char[] buf,int offset, int len)

write(int c)

write(String s)

write(String s, int offset, int len)

http://java.sun.com/j2se/1.5.0/docs/api/java/io/PrintWriter.html

But what if you want to put an object into a file?

First, you need to write the object into the file:

```
public class NamedLocation {
    String name;
    double x;
    double y;
    . . .
}
```

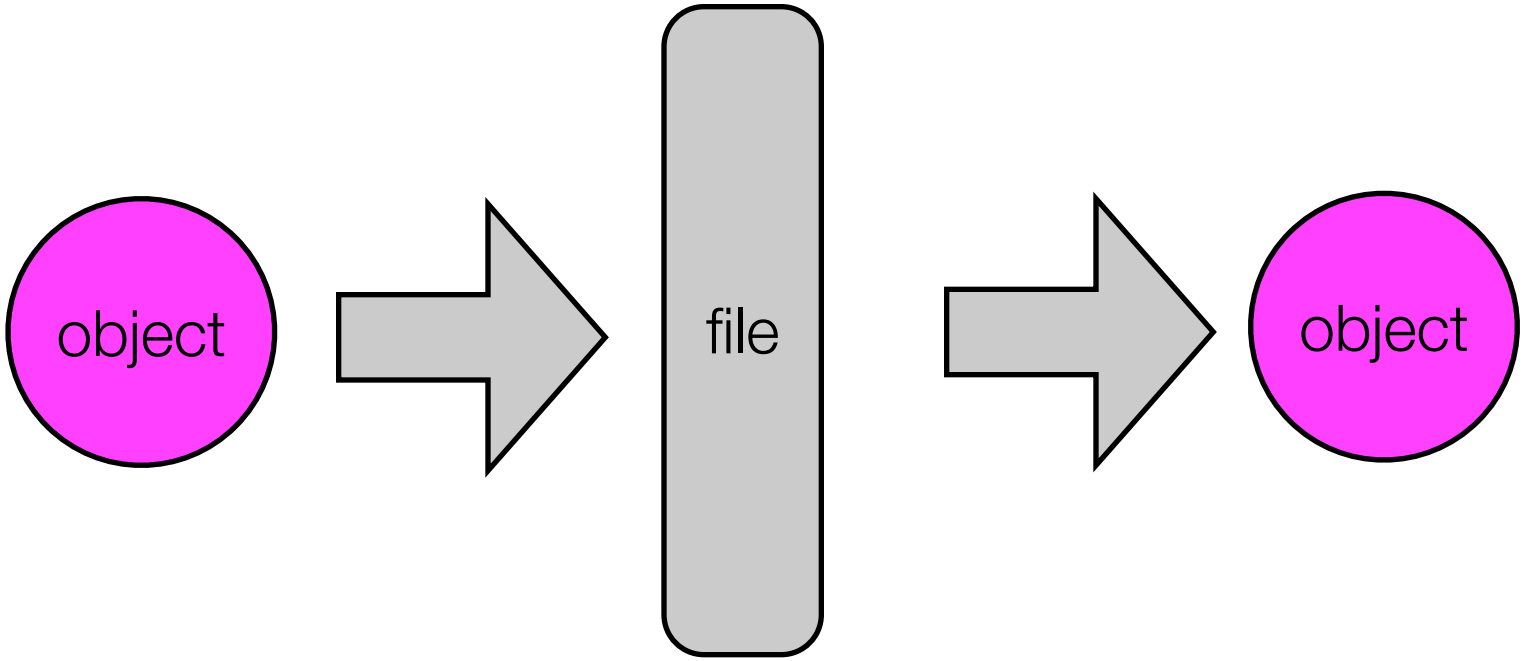
We would need to put into the file:

```
"NamedLocation"
name
x
y
```

… And we need a way to read it out.

"Serialization" turns objects into byte streams.

"Serialization" — Literally turns the object into a "series of bytes."



Java gives us serialization "for free."

To get serialization, just implement **Serializable**:

```
  class NamedLocation implements Serializable {
      String name;
      final double x;
      final double y;
      NamedLocation(String name,double x,double y){
          this.name = name;
          this.x = x;
          this.y = y;
      }
      public String toString(){
       return "NamedLocation[name="+name+" x="+x+" y="+y+"]";
      }
}
```

You write serialized objects to an

```
public class SerializationDemo {
    public static void main(String[] args){
        NamedLocation loc = new NamedLocation("MyName",10,20);

        System.out.println("Writing loc="+loc+" to the file...");

        try {
            ObjectOutputStream oos = new ObjectOutputStream(
                     new FileOutputStream("myfile.out"));
            oos.writeObject(loc);
            oos.close();
        } catch (java.io.IOException e){
            e.printStackTrace();
        }
    }
}

$ java SerializationDemo
Writing loc=NamedLocation[name=MyName x=10.0 y=20.0] to the file...
$
```

What's in myfile.out?

```
$ strings myfile.out
NamedLocation
namet
Ljava/lang/String;xp@$
MyName
$

$ hexdump myfile.out
0000000 ac ed 00 05 73 72 00 0d 4e 61 6d 65 64 4c 6f 63
0000010 61 74 69 6f 6e ee 6f be d3 d8 25 8a 9f 02 00 03
0000020 44 00 01 78 44 00 01 79 4c 00 04 6e 61 6d 65 74
0000030 00 12 4c 6a 61 76 61 2f 6c 61 6e 67 2f 53 74 72
0000040 69 6e 67 3b 78 70 40 24 00 00 00 00 00 00 40 34
0000050 00 00 00 00 00 00 74 00 06 4d 79 4e 61 6d 65
000005f
09:31 PM imac2:~/current/cs3773/week8$

$ cat myfile.out
NamedLocation?o???%??DxDyLnametLjava/lang/String;xp@$@4tMyName
$
```

Read with ObjectInputStream

```
import java.io.*;

public class ReadingDemo {
    public static void main(String[] args){

        try {
            ObjectInputStream iis = new ObjectInputStream(
                    new FileInputStream("myfile.out"));
            Object obj = iis.readObject();
            System.out.println("obj="+obj);
            iis.close();
        } catch (java.io.IOException e){
            e.printStackTrace();
        } catch (java.lang.ClassNotFoundException e){
            e.printStackTrace();
        }
    }
}

$ java ReadingDemo
obj=NamedLocation[name=MyName x=10.0 y=20.0]
$
```

So what does **transient** do?

transient prevents variables from being written when an object is serialized.

Change NamedLocation to this:

```
class NamedLocation implements Serializable {
    transient String name;
    final double x;
    final double y;
    NamedLocation(String name,double x,double y){
        this.name = name;
        this.x = x;
        this.y = y;
    }
    public String toString(){
     return "NamedLocation[name="+name+" x="+x+" y="+y+"]";
    }
}
```

… And the **name** won't be saved.

Using volatile to avoid saving data can dramatically save time & space

```
private int value;
private transient String name;
private Date    timeStamp;
private transient JPanel panel;
```

Serialization Comparison

	`TestObject`	`TestObjectTrans`
Save Time	990 ms	110 ms
Load Time	3,680 ms	1,040 ms
File Size	91.7K	1.6K

Note: If you do not save the transients, you need to initialize them when the object is read!
See http://java.sun.com/docs/books/performance/1st_edition/html/JPIOPerformance.fm.html

You are now responsible for all of the Java Reserved Words.

---

abstract	**do**	**import**	**short**	**volatile**
assert	**double**	**instanceof**	**static**	**while**
boolean	**else**	**int**	**strictfp**	
break	**enum**	**interface**	**super**	
byte	**extends**	**long**	**switch**	
case	**final**	**native**	**synchronized**	
catch	**finally**	**new**	**this**	
char	**float**	**package**	**throw**	
class	**for**	**private**	**throws**	
[const]	**[goto]**	**protected**	**transient**	
continue	**if**	**public**	**try**	
default	**implements**	**return**	**void**	

http://java.sun.com/docs/books/tutorial/java/nutsandbolts/_keywords.html