MultiThreading

Object Orientated Programming in Java

Benjamin Kenwright

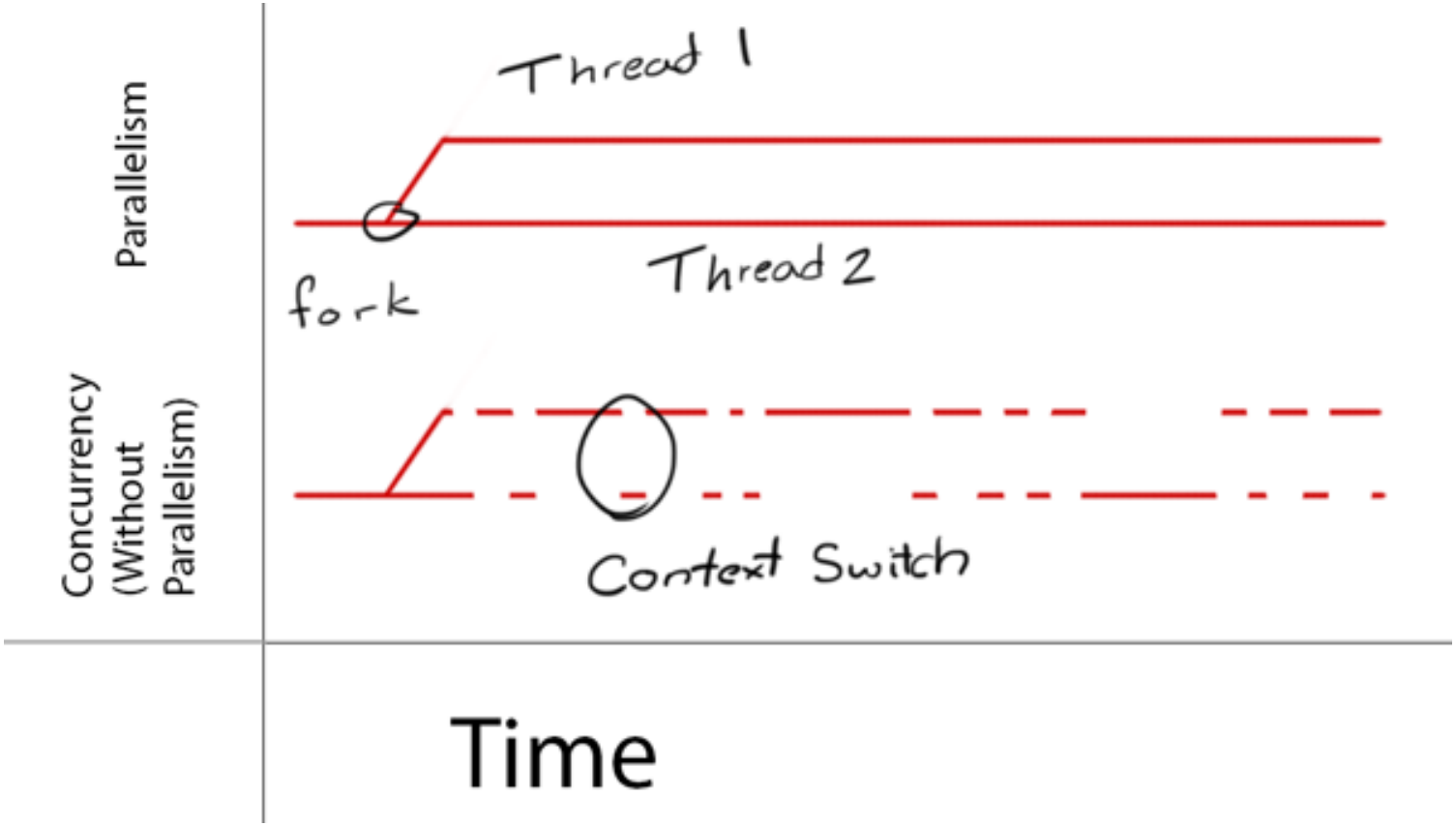
Outline

- Essential Java Multithreading
- Examples
- Today's Practical
- Review/Discussion

Why Multithreading?

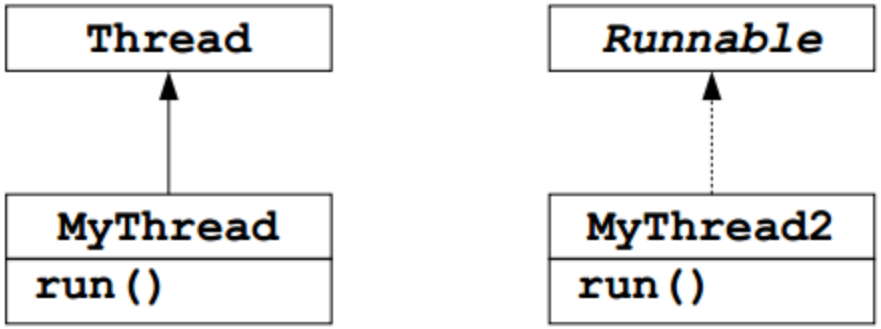
- What is the rational?
- Why make things complicated?
- What would happen if we didn’t have multithreading?

Concurrency & Parallelism



Threading

- Advantages & Disadvantages of Threads
- Java Threads
  - Class: java.lang.Thread
  - Interface: java.lang.Runnable
- Multithreaded Programming



Thread Definition

- *Definition*: A thread is a single sequential flow of control within a program (also called *lightweight process*)

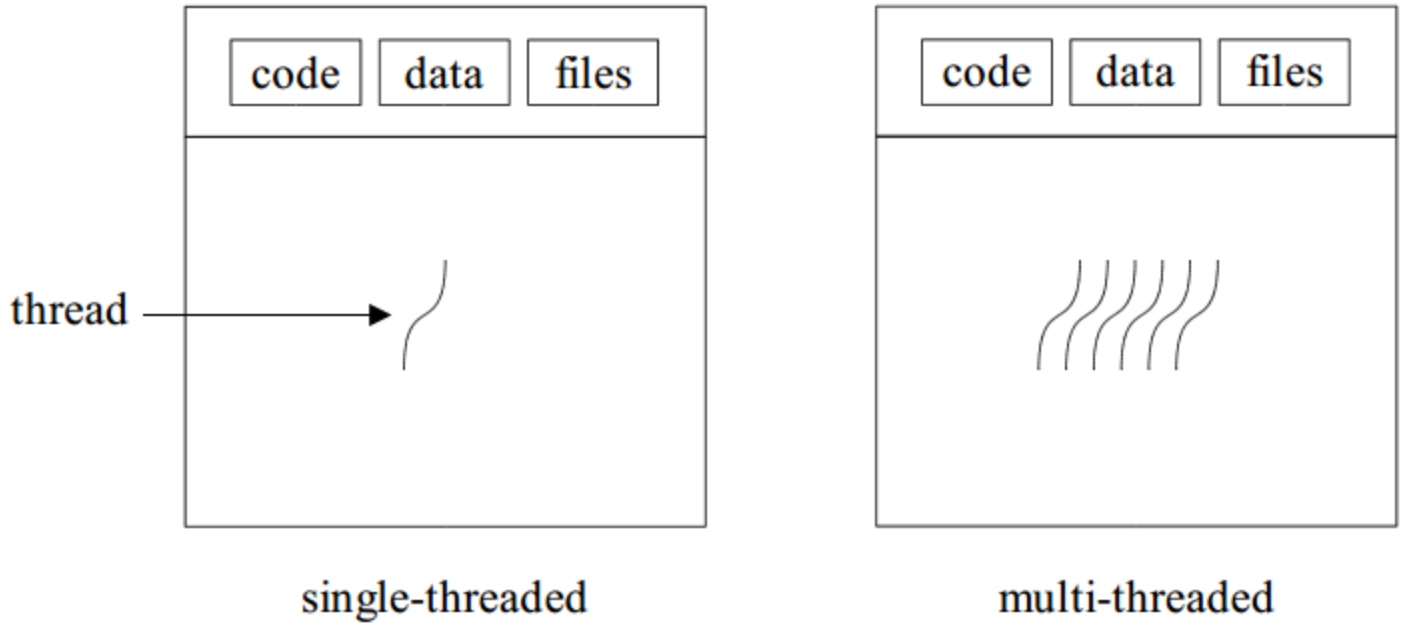
Thread

- Each thread acts like its own sequential program
  - Underlying mechanism divides up CPU between multiple threads
- Two types of multithreaded applications
  - Make many threads that do many tasks in parallel, i.e., no communication between the threads (GUI)
  - Make many threads that do many tasks concurrently, i.e., communication between the threads (data access)

Advantages/Disadvantages

- Advantages
  - Responsiveness, e.g., of user interfaces
  - Resource sharing
  - Economy
  - Utilization of multiprocessor hardware architectures
- Disadvantages
  - More complicated code
  - Deadlocks (very hard to debug logical program errors)

Single & Multithreaded Processes



User and Kernel Threads

- Thread management done by user-level threads library

  Examples

  - POSIX *Pthreads* (e.g., Linux and NT)
  - Mach *C-threads* (e.g., MacOS and NeXT)
  - Solaris *threads*

- Supported by the kernel

  Examples

  - Windows 95/98/NT/2000/XP
  - Solaris
  - TRU64 (one of HP's UNIX)

Java Threads

- Java threads may be created by
  - Extending **Thread** class
  - Implementing the **Runnable** interface

Class **Thread**

- The simplest way to make a thread
- Treats a thread as an object
- Override the **run()** method, i.e., the thread's "main"
  - Typically a loop
  - Continues for the life of the thread
- Create **Thread** object, call method **start()**
- Performs initialization, call method **run()**
- Thread terminates when **run()** exits

Extending the **Thread** Class

```
class Worker extends Thread {
   public void run() {
       System.out.println("I\'m a worker thread");
    } // thread is dead
}


public class First{
    public static void main (String args[]){
          Worker runner = new Worker();
          runner.start();
          System.out.println("I\'m the main thread");
    } // main thread alive until all children are dead
}
```

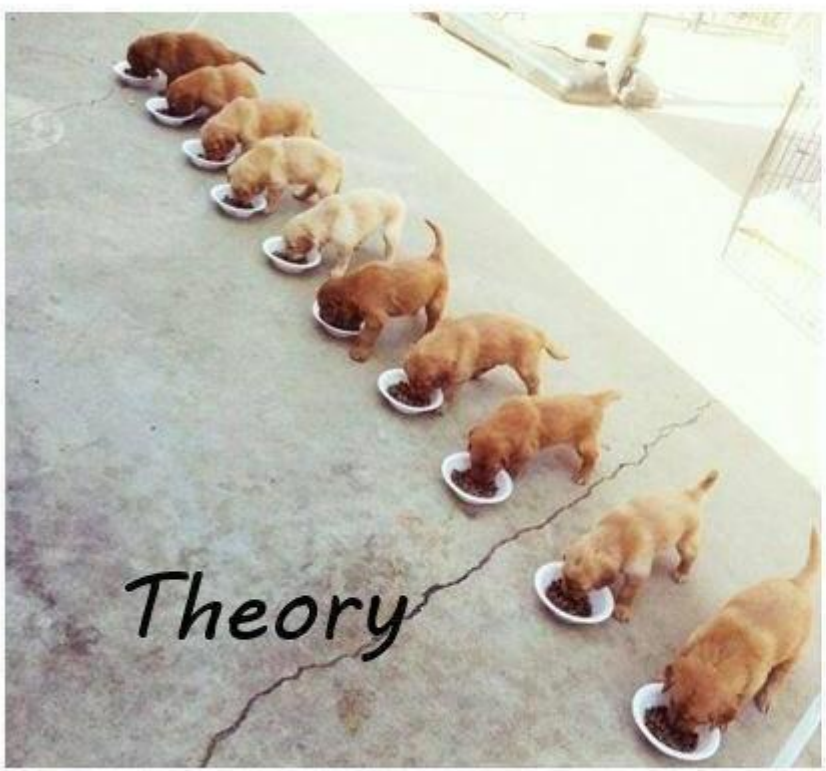
Extending the **Thread** Class Example

```
class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str);
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {
                sleep((int)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
        System.out.println("DONE! " + getName());
    }
}

class ThreadTest {
    public static void main (String[] args) {
        new SimpleThread("Hello").start();
        new SimpleThread("Goodbye").start();
    }
}
```

```
G:\>java -cp . ThreadTest
0 Goodbye
0 Hello
1 Goodbye
2 Goodbye
1 Hello
3 Goodbye
2 Hello
3 Hello
4 Goodbye
4 Hello
5 Goodbye
6 Goodbye
5 Hello
7 Goodbye
6 Hello
8 Goodbye
7 Hello
9 Goodbye
DONE! Goodbye
8 Hello
9 Hello
DONE! Hello
```

javac ThreadTest.java
java -cp . ThreadTest

Multithreaded Programming



Sharing Resources

- *Single threaded programming*: you own everything, no problem with sharing
- *Multi-threaded programming*: more than one thread may try to use a shared resource at the same time
  - Add and withdraw from a bank account
  - Using the speakers at the same time, etc.
- Java provides locks, i.e., monitors, for objects, so you can wrap an object around a resource
  - First thread that acquires the lock gains control of the object, and the other threads cannot call synchronized methods for that object

Locks

- One lock pr. object for the object's methods
- One lock pr. class for the class' static methods

- Typically data is private, only accessed through methods
  - Must be private to be protected against concurrent access
- If **a method is synchronized**, entering that method acquires the **lock**
  - No other thread can call *any* synchronized method for that object until the lock is released

Sharing Resources, cont.

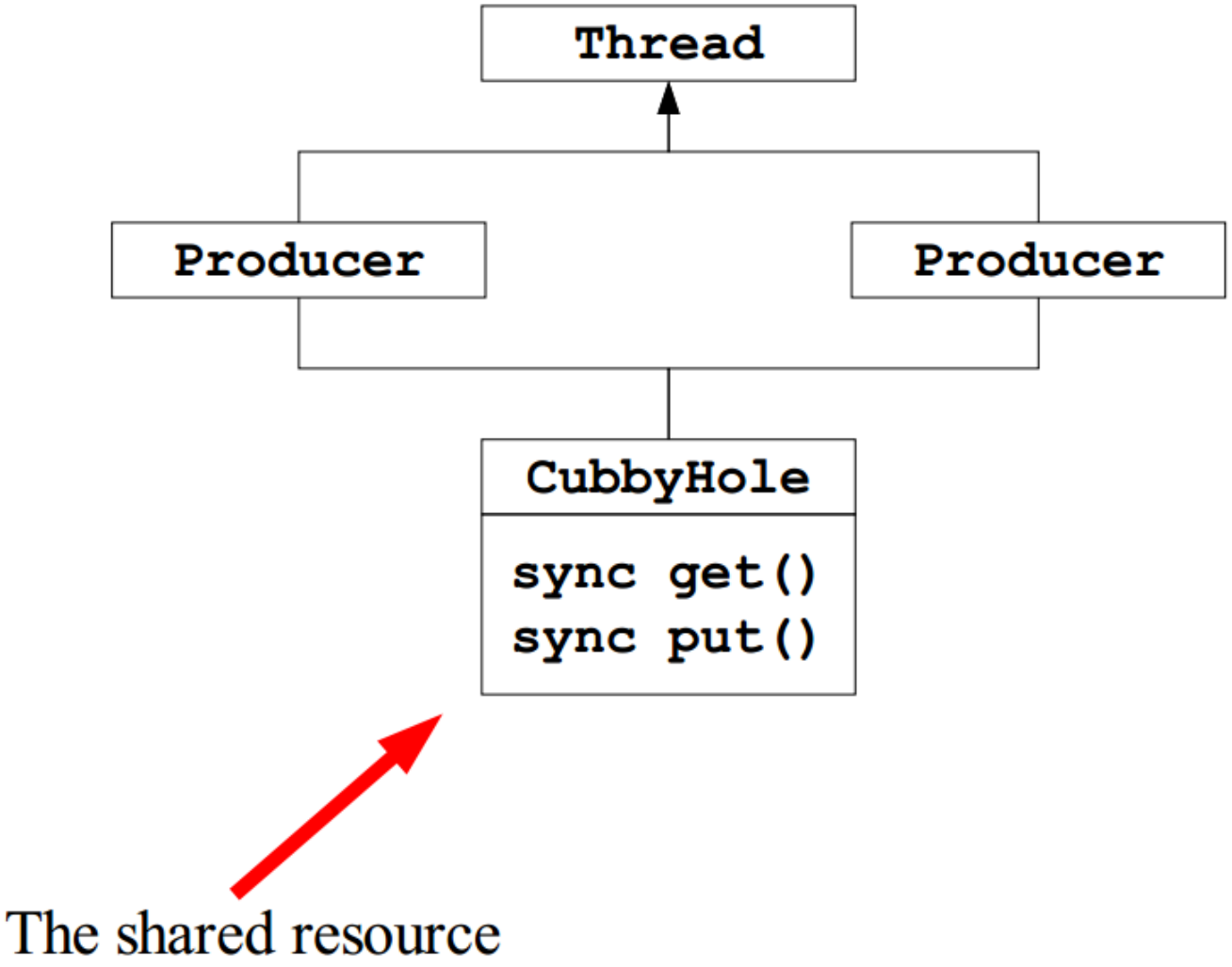
- Only one synchronized method can be called at any time for a particular object

```
synchronized void foo() {/*..*/}
synchronized void bar() {/*..*/}
```

Sharing Resources, cont.

- Efficiency
  - Memory: Each object has a lock implemented in **Object**
  - Speed and Overhead (e.g., calling)
    - Older standard Java libraries used synchronized a lot, did not provide any alternatives

Sharing Resources, Example



Sharing Resources, Example

```
public class CubbyHole {
   private int contents;
   private boolean available = false;
   public synchronized int get() {
       while (available == false) {
              try { wait(); } ... }
       available = false;
       notifyAll();
       return contents;
   }
   public synchronized void put(int value) {
       while (available == true) {
              try { wait(); ...} }
       contents = value;
       available = true;
       notifyAll();
   }
}
```

Sharing Resources, Example cont.

```
public class Producer extends Thread {
   private CubbyHole cubbyhole;
   private int number;
   public Producer(CubbyHole c, int number) {
       cubbyhole = c;
       this.number = number;     }
   public void run() {
       for (int i = 0; i < 10; i++) {
         cubbyhole.put(i);
       System.out.println(
       "Producer #" + this.number + " put: " + i);
       try {sleep((int)(Math.random() * 100));
       } catch (InterruptedException e) { } }
   }
}
```

Sharing Resources, Example cont.

```
public class Consumer extends Thread {
   private CubbyHole cubbyhole;
   private int number;
   public Consumer(CubbyHole c, int number) {
       cubbyhole = c;
       this.number = number;
   }
   public void run() {
       int value = 0;
       for (int i = 0; i < 10; i++) {
         value = cubbyhole.get();
         System.out.println(
        "Consumer #" + this.number + " got: " + value);
       }
   }
}
```

Sharing Resources, Example cont.

```
public class ProducerConsumerTest {
   public static void main(String[] args) {
       CubbyHole c = new CubbyHole();
       Producer p1 = new Producer(c, 1);
       Consumer c1 = new Consumer(c, 1);
       p1.start();
       c1.start();
   }
}
```

The **Runnable** Interface

- To inherit from an exising object and make it a thread, implement the **Runnable** interface
- A more classical, function-oriented way to use threads

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

The **Runnable** Interface, cont .

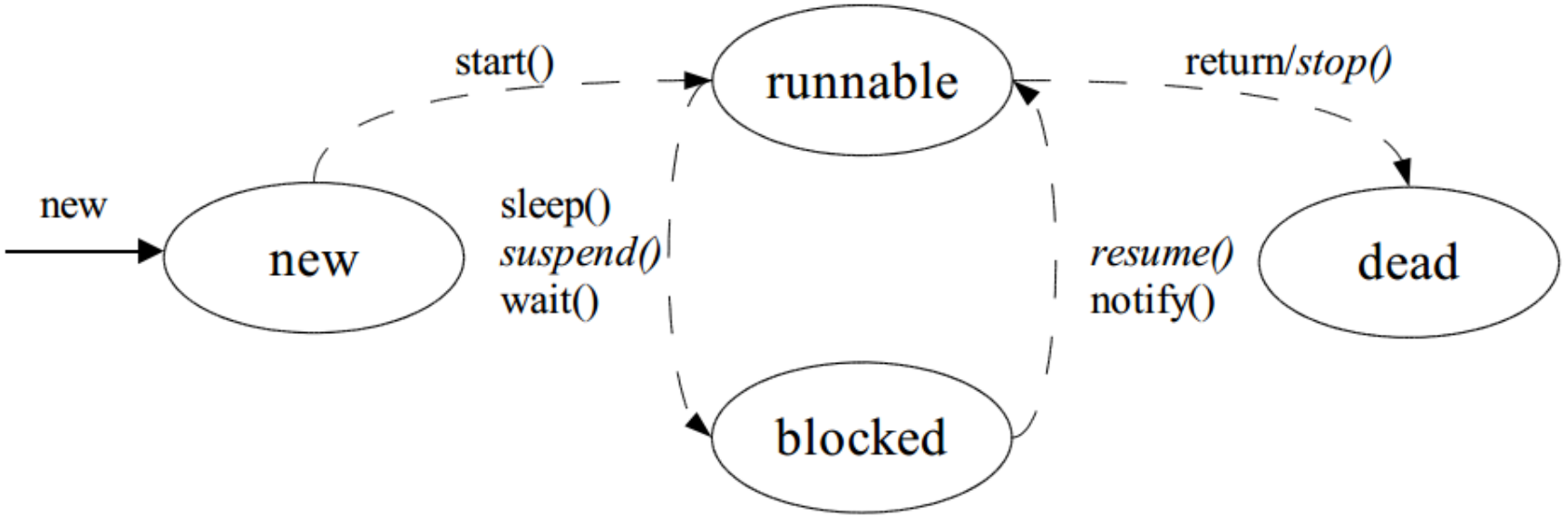
```
class Worker implements Runnable{
   public void run(){
       System.out.println("I\'m a worker thread");
   }
}

public class Second{
    public static void main(String args[]) {
          Runnable runner = new Worker();
          Thread thrd = new Thread(runner);
          thrd.start();
          System.out.println("I\'m the main thread");
    }
}
```

The **Runnable** Interface, cont.

```
class SimpleRunnable implements Runnable {
   private String myName;    private Thread t;
   SimpleRunnable (String name) {
        myName = name; t = new Thread(this); t.start();
   }
   public void run() {
        for (int i = 0; i < 10; i++) {
           System.out.println(i + " " + myName);
           try {
                 t.sleep((long)(Math.random() * 1000));
             } catch (InterruptedException e) {}
         }
         System.out.println("DONE! " + myName);
     }
}
public class TwoRunnableDemo {
   public static void main (String[] args)
   {        SimpleRunnable runner1 = new SimpleRunnable
   ("Jamaica");        SimpleRunnable runner2 = new
   SimpleRunnable("Fiji");    }
```

Java Thread Management

- *suspend()* – suspends execution of the currently running thread
- *sleep()* – puts the currently running thread to sleep for a specified amount of time
- *resume()* – resumes execution of a suspended thread.
- *stop()* – stops execution of a thread.



Synchronized Fields and Constructors

Class or object fields cannot be synchronized.

```
public class DataFields{
  /** A synchronized object field not allowed */
  private synchronized int x;
  /** A synchronized class field not allowed */
  public static synchronized int y;
}
```

- Constructors cannot be synchronized.

```
public class DataFields{
  public synchronized DataFields(){// not allowed }
  public static synchronized void staticMethod(){
    System.out.println("I'm in sync"); // allowed
  }
}
```

Issues

- Thread priority
- Thread groups
- Daemon (unix term)
  - similar to a service (on Win32)
- Deadlock
  - Very hard to detect logical errors in programs

Deadlocks

```
public class TwoResources {
   private int contentsA = 10;
   private int contentsB = 20;
   private boolean availableA = true;
   private boolean availableB = true;
   public synchronized int getA() {
        while (availableA == false) {
              try { wait(); } ... }
        // snip see CubbyHole
}
   public synchronized void putA(int value) {
        while (availableA == true) {
              try { wait(); ...} }
        // snip see CubbyHole
   }
    // ditto for B resource
}
```

Deadlocks, cont.

```
public class TRConsumer extends Thread {
  // start thread in constructor
  private TwoResources tr;
  public void getAthenB(){
    int a = tr.getA();  sleepy(2000);
    int b = tr.getB();
  }
  public void getBthenA(){
    int b = tr.getB(); sleepy(2000);
    int a = tr.getA();
  }
  public static void createDeadlock(){
    TwoResources tr = new TwoResources();
    TRConsumer one = new TRConsumer(tr, "A"); // A B
    TRConsumer two = new TRConsumer(tr, "B"); // B A
  }
}
```

Summary

- Overview Multithreading with Java
- *Single-threaded programming*: live by all by your self, own everything, no contention for resources
- *Multithreading programming*: suddenly “others” can have collisions and destroy information, get locked up over the use of resources
- Multithreading is built-into the Java programming language
- Multithreading makes Java programs complicated
  - ▷Multithreading is by nature difficult, e.g., deadlocks

This Week

- Read Associated Chapters
- Review Slides
- Java Exercises

Questions/Discussion