

Threads



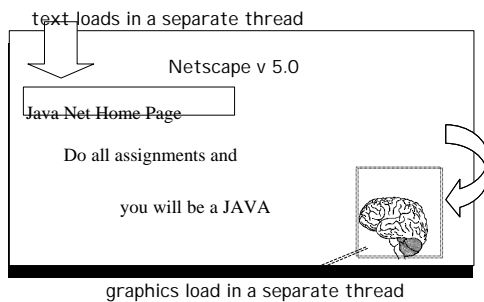
What is a Thread?

a flow of control within a program



sequential programs have one thread

Browsers run Multiple Treads



Why Threads?

- Provides parallel computation with low overhead
- Use threads when a program may need to wait for some resource
 - disk access, network connection
- When one thread is waiting, other can continue processing

Threads vs Processes

- A process has its own address space
 - In a multitasking operating system, each program is run as a separate process
 - Process switching has overhead
- A thread shares the address space of the the program that created it
 - minimal overhead with thread switching

the Thread class

- `System.out.println("hello");`
- `Thread.sleep(1000);`
- `System.out.println("world");`

puts currently executing thread to sleep for 1000 ms

static method

actually....

```
try {  
    Thread.sleep(1000); ←  
}  
catch (InterruptedException e) { }
```

Creating your own thread

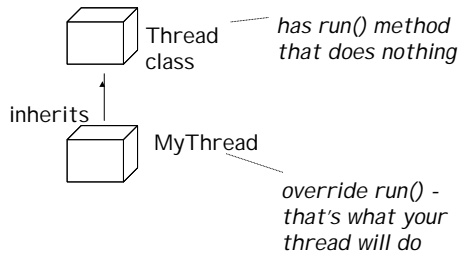
- Two ways to create a Java thread:
 - Extend the Thread class and override the run () method
 - Implement the Runnable interface
 - define run ()

either way, there is a thread object

whatever code is in run () will be run as a separate thread



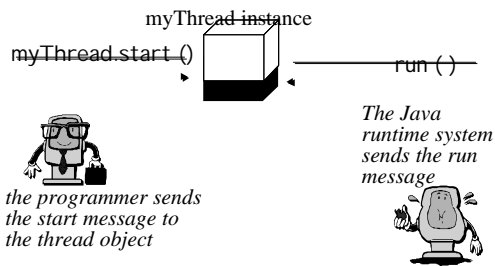
Subclass Thread



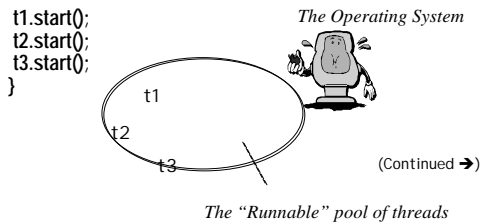
Subclassing Thread

```
class SimpleThread extends Thread {  
    private String internalName;  
  
    SimpleThread (String name) {  
        internalName = name;  
    }  
  
    public void run() {  
        for (int i=0; i<5; i++) {  
            System.out.println(internalName);  
        }  
    }  
}
```

Activating your thread



```
public class SimpleThreadTest1 {  
    public static void main (String argv[]) {  
        SimpleThread t1 = new SimpleThread("sun");  
        SimpleThread t2 = new SimpleThread("java");  
        SimpleThread t3 = new SimpleThread("beans");  
    }  
}
```

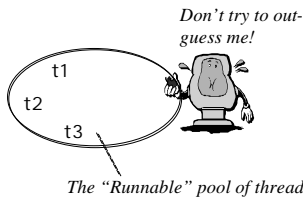


```
class SimpleThread extends Thread {
    String internalName;

    public void run() {
        for (int i=0; i<5; i++) {
            System.out.println(internalName);
        }
    }
}
```

Output:

```
sun
sun
java
java
sun
beans
beans
sun
java
java
beans
sun
beans
beans
```



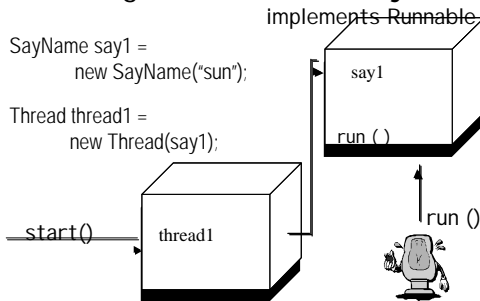
Option 2. Runnable interface

```
class SayName implements Runnable {
    private String internalName;

    SayName (String name) {
        internalName = name;
    }

    public void run() {
        for (int i=0; i<8; i++) {
            System.out.println(internalName);
        }
    }
}
```

Working with Runnable objects



```
class SayNameTest {
    public static void main (String [] args ) {
        SayName say1 = new SayName("sun");
        SayName say2 = new SayName("java");
    }
}
```

```
Thread thread1 = new Thread(say1);
thread1.start();
```

```
Thread thread2 = new Thread(say2);
thread2.start(); }
```

```
>java SayNameTest
sun..java..java..java..java..java..java
..java..sun..sun..sun..sun..sun..sun..
```

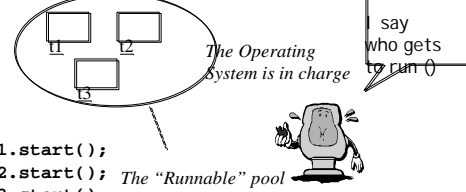
Thread Creation RULE

- Always create an instance of Thread
- Rule:
 - either use new with a subclass of Thread
 - **new MyThread();**
 - or use new with Thread (anObject)
 - **new Thread (anObject);**

must implement Runnable interface

Thread Execution

```
SimpleThread t1 = new SimpleThread("sun");
SimpleThread t2 = new SimpleThread("java");
SimpleThread t3 = new SimpleThread("beans");
```



```
t1.start();
t2.start();
t3.start();
```

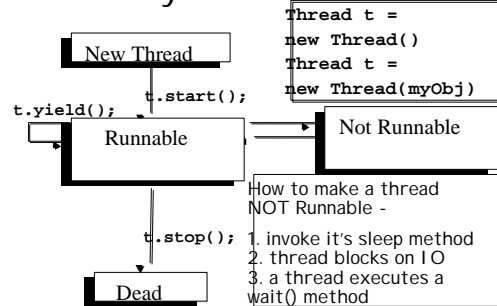
Hog Prevention with yield()

- Use yield to prevent a thread from hogging the CPU

```
public void run() {
    for (int i=0; i<8; i++) {
        System.out.println(internalName);
        Thread.yield();
    }
}
```

```
>java SayNameTest
sun..java..sun..java..sun..java..sun..java..
sun..java..sun..java..java..sun..java..sun..
```

Life Cycle of a Thread



Partitioning Work with Threads

Three Work Threads

```
t1.start(); // returns immediately
t2.start();
t3.start();
System.out.println("done");
```

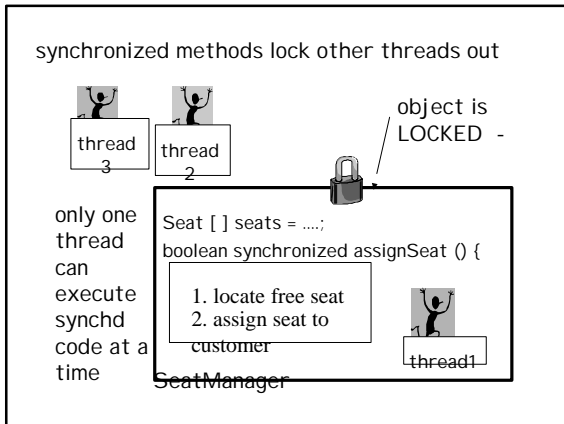
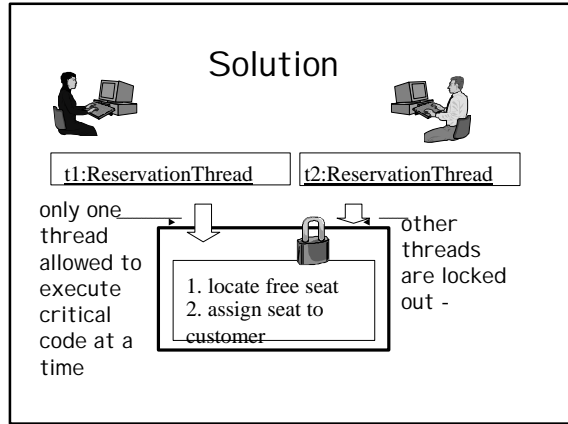
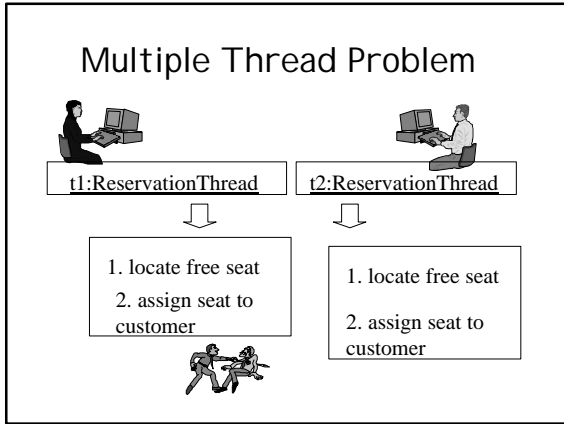
will print before threads complete their work

Three Work Threads

```
t1.start(); // returns immediately
t2.start();
t3.start();
t1.join(); // main thread will block until t1 is no longer alive
t2.join();
t3.join();
System.out.println("done");
```

will print after all threads complete their work

Threads II Synchronization



```
class Account {
    private double balance;

    public Account (double initialDeposit) {
        balance = initialDeposit;
    }
    public synchronized double getBalance () {
        return balance;
    }
    public synchronized void deposit (double amount) {
        balance += amount;
    }
}
```

what is locked?

```
class Account {
    synchronized double getBalance() {
        return balance;
    }
}
```

Account a1 = new Account(200); a1 has

Account a2 = new Account(500); a2 has

the object instance is locked

when a thread enters synchronized code -- all other threads trying to access other synchronized code of that object, are blocked!

```
class Agent { ... }
a = new Agent();

synchronized boolean foo() {
    //code
}

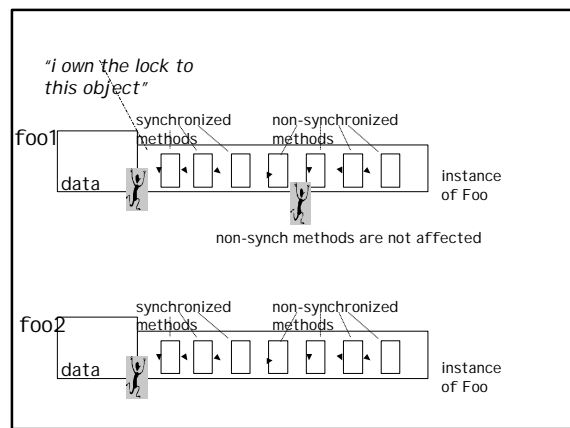
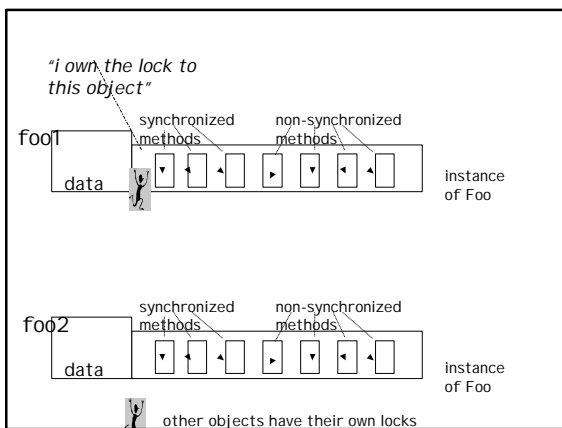
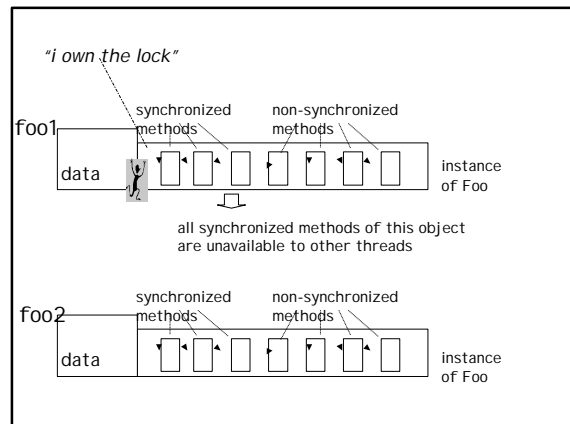
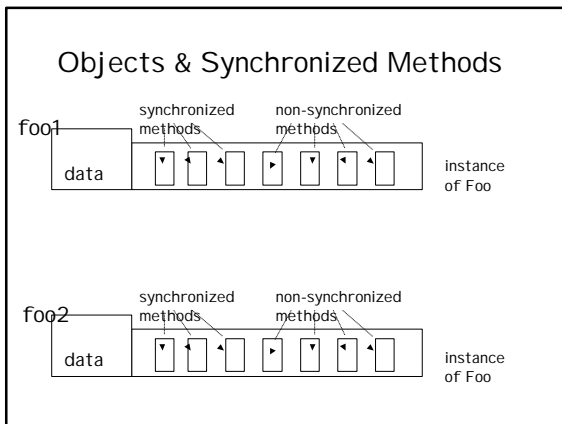
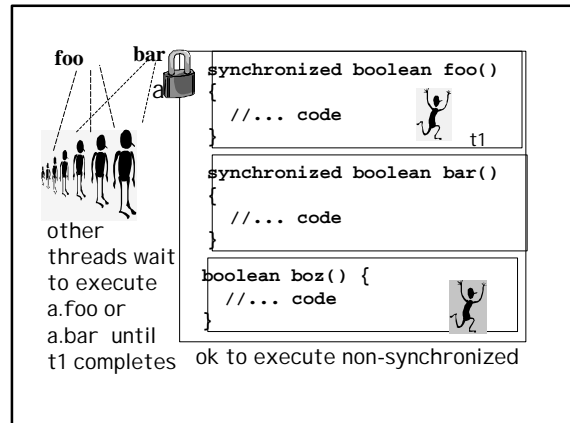
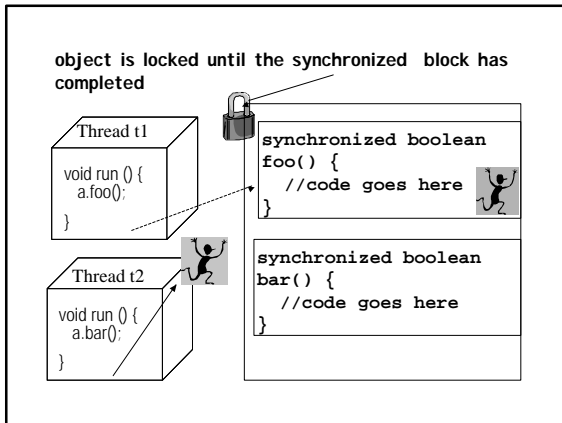
synchronized boolean bar() {
    //code
}
```

a Thread

```
void run () {
    a.foo();
}
```

a Thread

```
void run () {
    a.bar();
}
```



What about variables?

- Variables cannot be synchronized
- However you can control access if you:
 - declare variables private
 - provide accessor methods

```
class Account {
    private double balance;
    synchronized void updateBalance (double amt) {
        balance += amt;
    }
}
```

Synchronized statements

- Allows locking an object without a synchronized method
- Form:

```
synchronized (<some object>) {
    statements;
}
```

some object to lock

Control array access

```
public static void abs (int[] values) {
    synchronized (values) {
        for (int i=0; i<values.length; i++) {
            if (values[i] < 0)
                values[i] = -values[i];
        }
    }
}
```

locks the array values

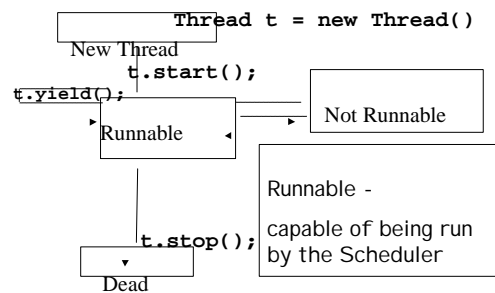
any object may serve as the lock object

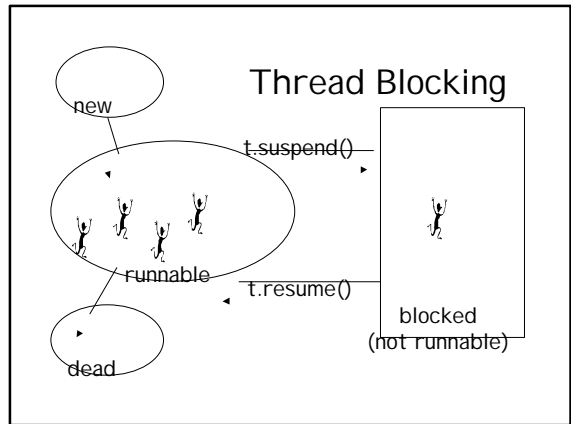
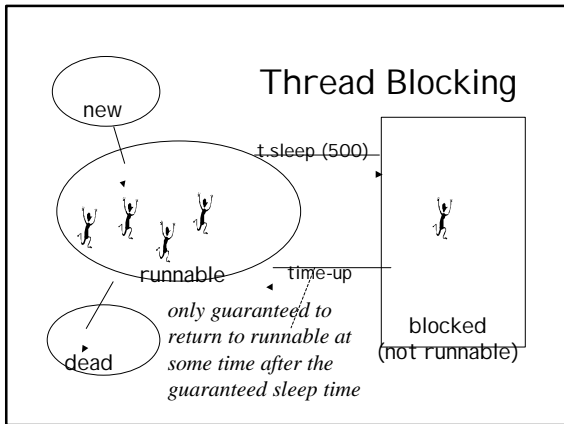
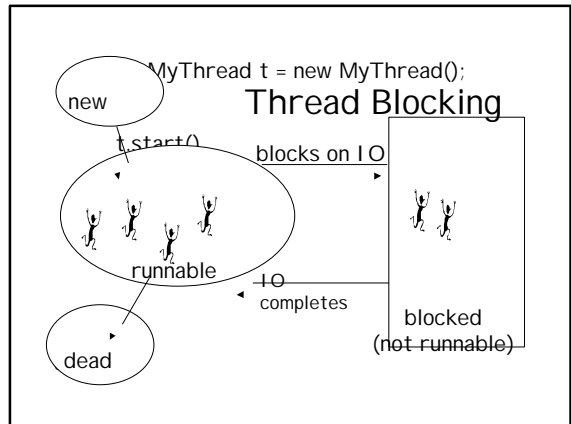
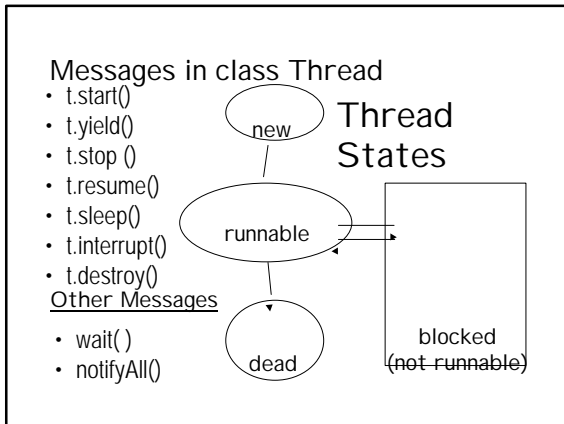
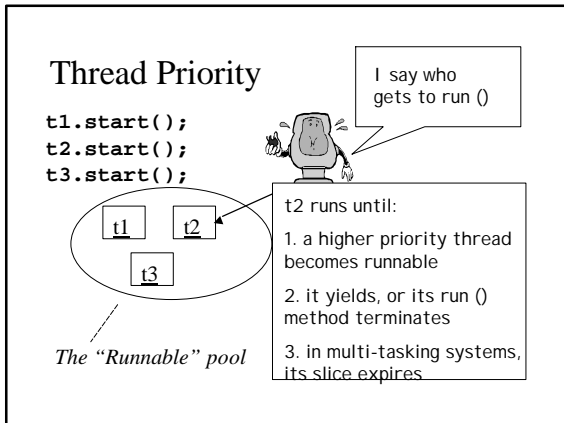
Synchronized Summary

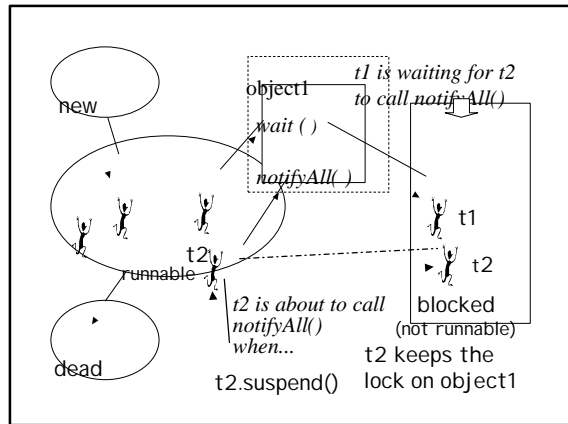
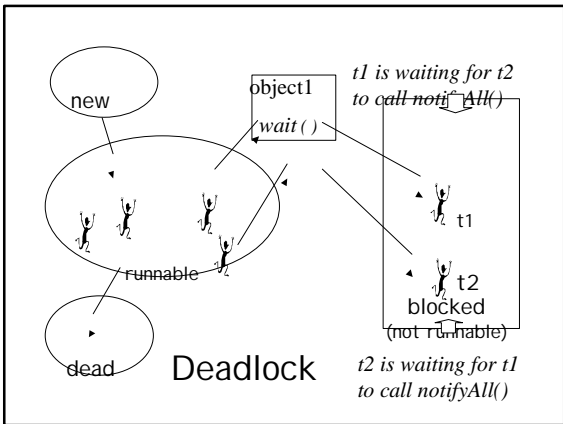
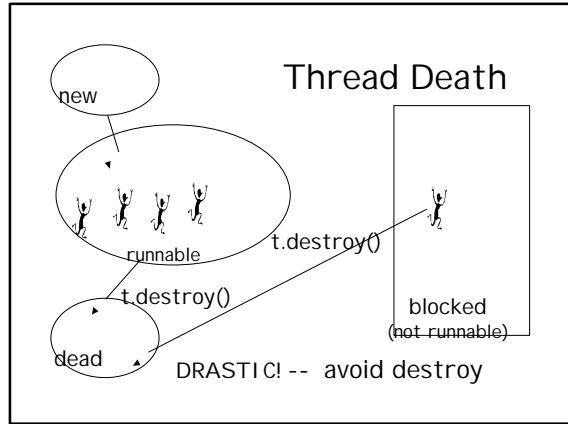
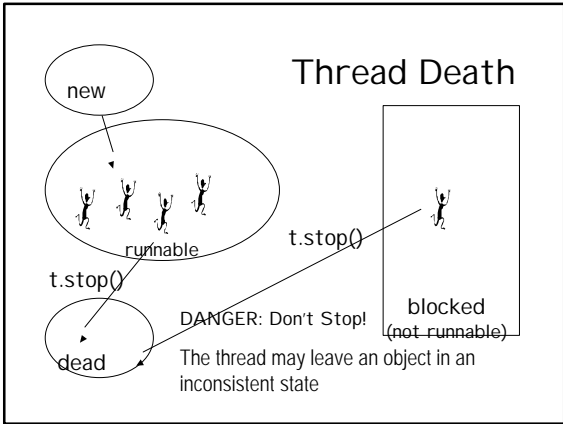
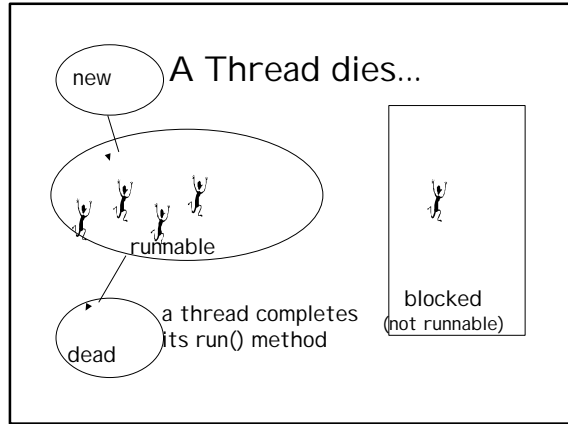
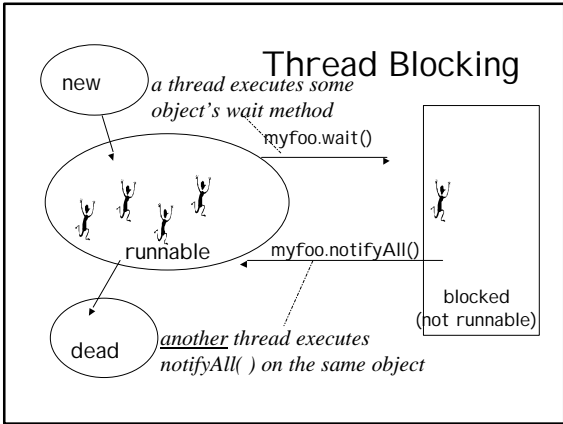
- Synchronized methods or code blocks LOCK an object
- Synchronization permits multiple threads to act concurrently without interfering with each other

Threads III Thread States

Life Cycle of a Thread





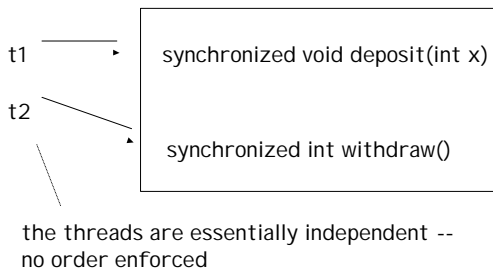


Thread Coordination with Notify and Wait

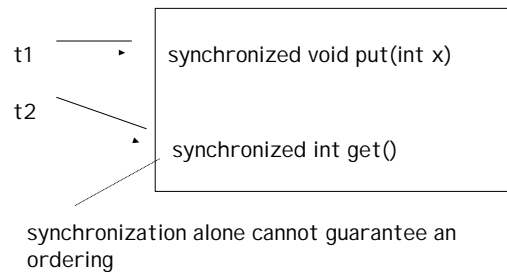
Objects and Threads

- Each object has a lock that can be obtained and released
 - Java uses an object's synchronized methods to control thread access the object
- Each object also has a waiting area for threads that need something other threads can provide
 - wait and notify

synchronization



what if ordering required? (put must precede get)



coordination solution #1

```
int value = -1; // no value yet

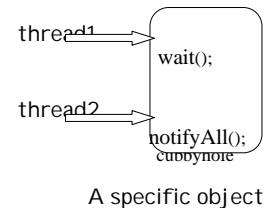
synchronized void put(int x) {
    value = x;
}

synchronized int get() {
    while (value == -1)
        { sleep(500); }
    return value; }

```

notify and wait

- wait ()
 - wait for in an object's waiting area
 - hope that another thread will execute notify or notifyAll
- notifyAll()
 - wake up all threads waiting in the object's waiting area

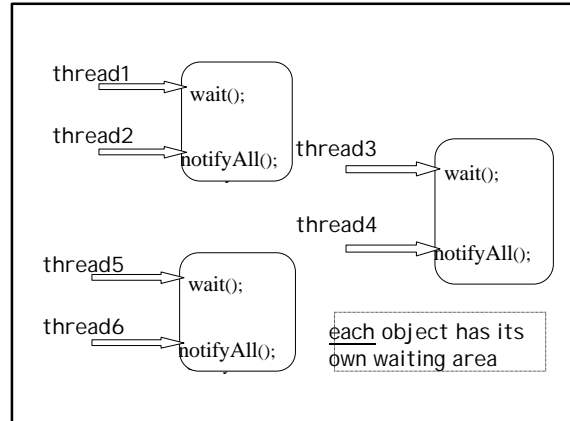


notify and wait (methods in class Object)

- All object's understand notify(), notifyAll() and wait()
- Use notify & wait in your code if:
 - you have get & put methods
 - you want consumer threads to wait on producer threads
 - you want producer threads to notify consumers



wait and notify is a communication mechanism



coordination solution #2

```
int value = -1; // no value yet

synchronized void put(int x) {
    value = x; notifyAll();
}

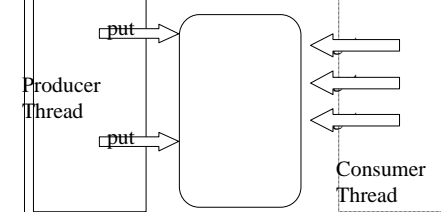
synchronized int get() {
    while (value == -1)
        { wait(); }
    return value; }

```

guarantees that a put will always occur before a get.

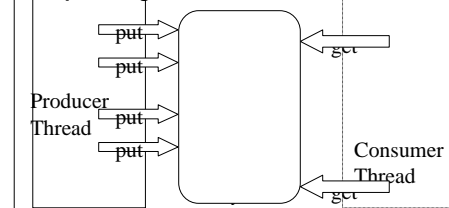
no busy wait loop

Speedy Consumer Problem



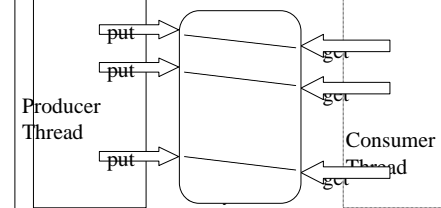
consumer gets duplicates!

Speedy Producer Problem



consumer misses values!

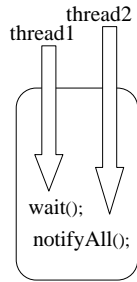
I deal



each put followed by a get
How can we guarantee it??

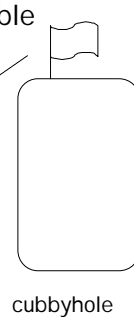
notify and wait - the difficulty

if thread2 goes first,
thread1 will miss what
thread2 did
thread1 may wait forever
needed: notify & wait & a
flag called available



notify and wait available

we set this flag when a
producer has deposited
something for a consumer to
get



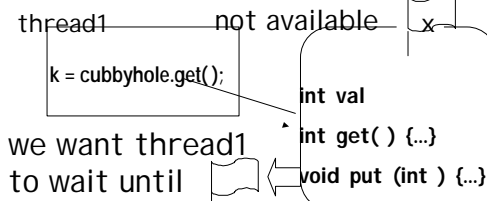
notify and wait not available

indicates the cubbyhole object
has no value -- nothing to get

initial state

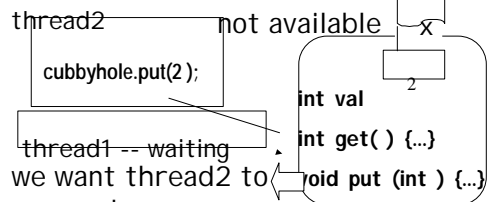
`boolean available = false;`

cubbyhole



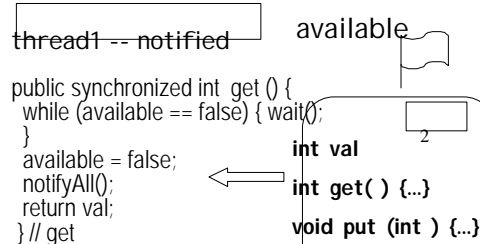
we want thread1
to wait until

```
public synchronized int get () {
    while (available == false) { wait(); }
    available = false; // because I got it!
    notifyAll(); // other deliveries may be waiting
    return val;
} // get
```



we want thread2 to
proceed...

```
public synchronized void put (int i) {
    while (available == true) { wait(); }
    val = i;
    available = true;
    notifyAll();
}
```



thread1 proceeds...

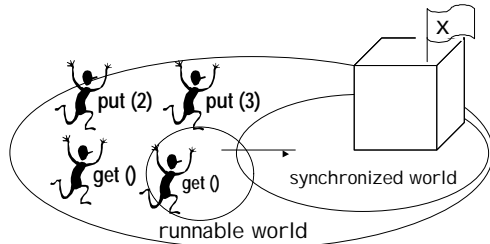
```

public synchronized int get () {
    while (available == false) {
        try { wait(); }
        catch (InterruptedException e) {}
    }
    available = true;
    notifyAll();
    return val;
} // get

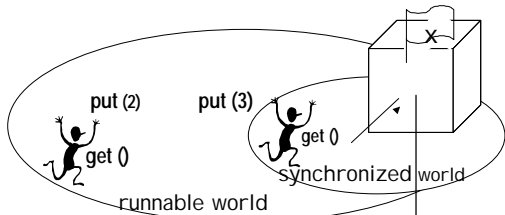
public synchronized void put (int i) {
    while (available == true) {
        try { wait(); }
        catch (InterruptedException e) {}
    }
    val = i;
    available = false;
    notifyAll();
} // put

```

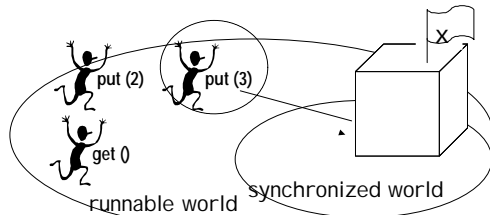
The Complete Code!



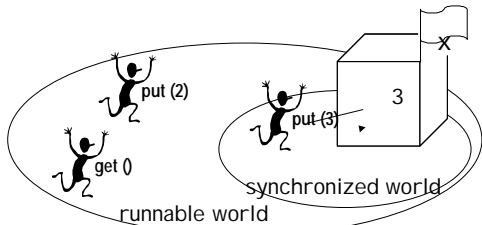
One runnable thread gets picked to execute



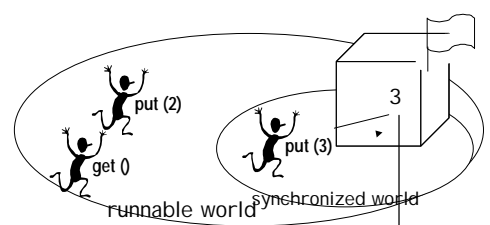
As long as the thread is executing a synchronized method no other thread can enter synchronized world



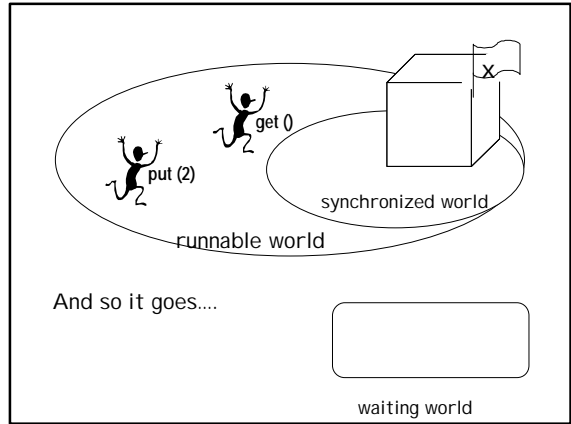
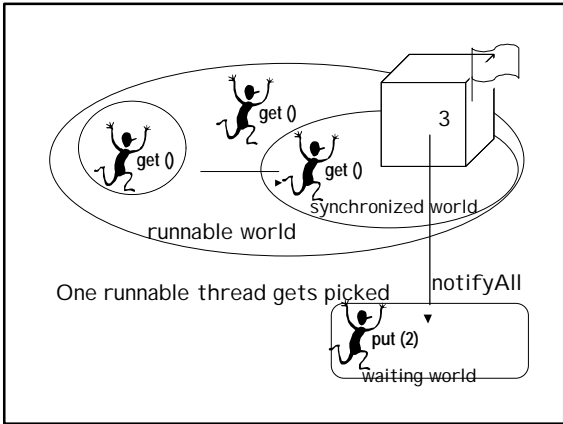
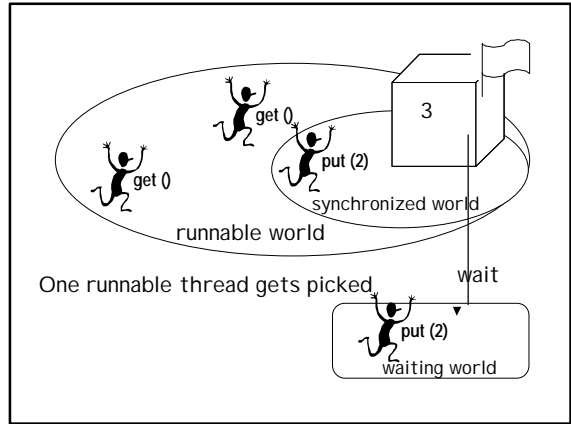
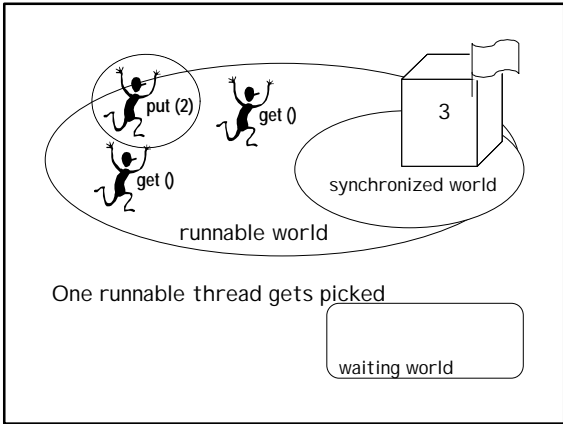
New runnable thread gets picked



New runnable thread gets picked



Waiting threads get notified =are made runnable
not guaranteed to get in!



END