



# Multithreading

---

- ❖ Thread Properties
- ❖ Thread Priorities
- ❖ Cooperating and Selfish Threads
- ❖ Synchronization
- ❖ Animation
- ❖ Timers
- ❖ Threads and Swing
- ❖ Using Pipes for Communication between Threads



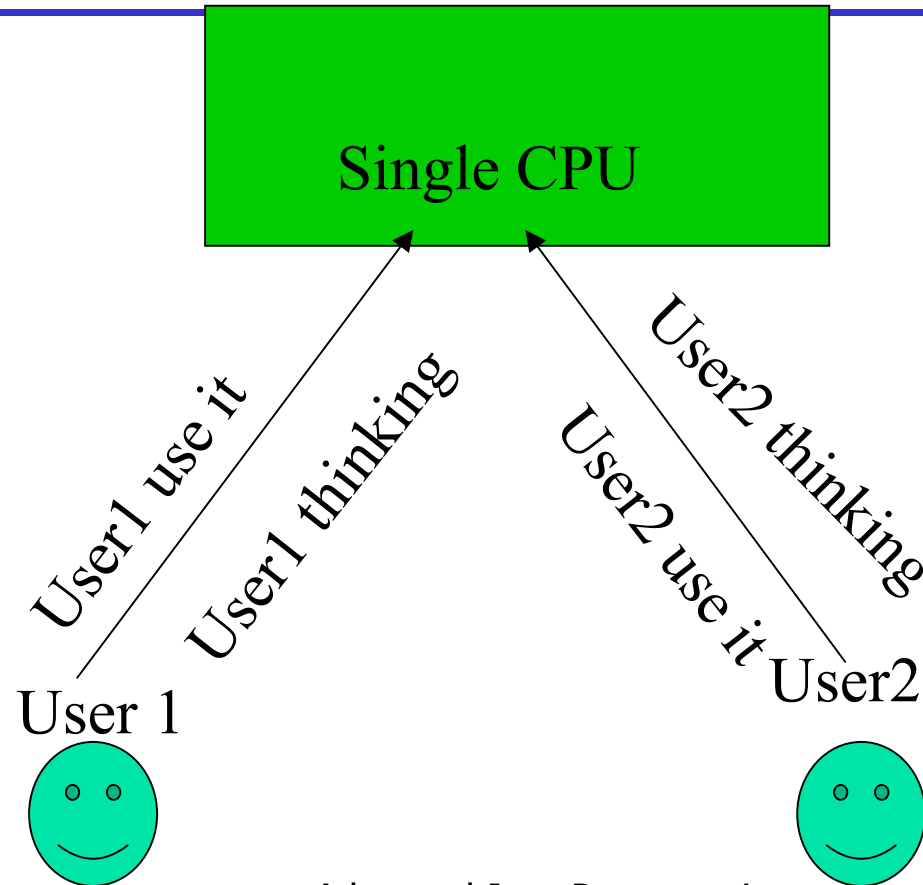
# Multitasking

---

- Multitasking: The ability to have more than one program working at what **seems like** the same time.
- Resource Distribution: The operating system is doling out resources to each program, giving the impression of parallel activity.



# Multitasking

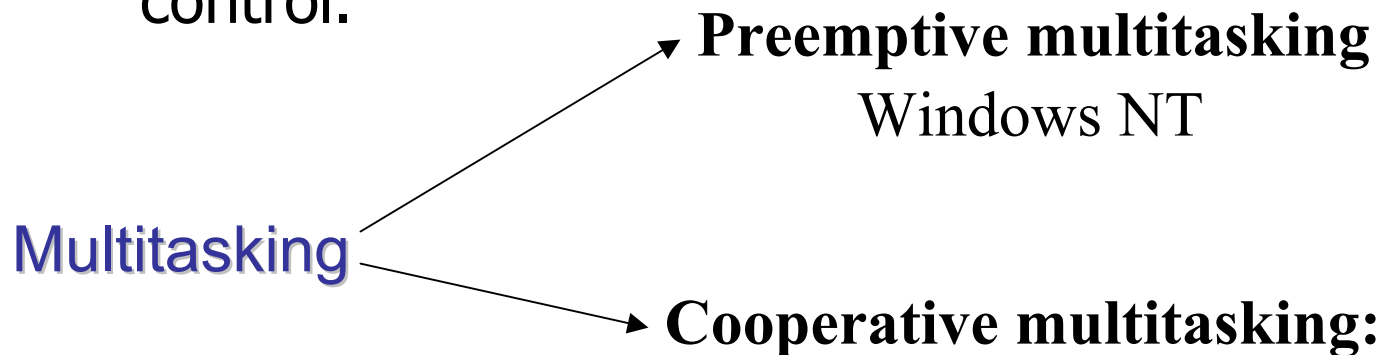




# Multitasking

---

- **Preemptive multitasking:** The operating system interrupts programs without consulting with them first.
- **Cooperative multitasking:** The operating system are only interrupted when they are willing to yield control.





# Multitasking

---

- **Preemptive multitasking is much more effective. With cooperative multitasking, a badly behaved program can hog everything.**



# Multithreading

---

- Multithreaded programs extend the idea of multitasking by taking one level lower: **individual programs** will appear to do **multiple tasks** at the **same time. Each task is usually called a thread.**
- Multithreaded programs: can run more than one thread at once.



# Multitasking and Multithreading

---

- The essential difference is that while each process has a complete set of its own variables, threads share the same data.
- All modern operating systems support multithreading.
- Multithreading is extremely useful in practice.
  - A browser should be able to deal with multiple hosts or open an e-mail or to view another page while downloading data.
  - The Java programming itself uses a thread to do garbage collection in the background.



# Single-threading

---

- Example: Bounce.java
- `public void bounce()`
- `{ draw();`
- `for (int i = 1; i <= 1000; i++)`
- `{ move();`
- `try { Thread.sleep(5); }`
- `catch(InterruptedException e) {}`
- `}`
- `}`



# Single-threading

---

- If you run the program, you can see that the ball bounces around nicely, but **it is completely takes over the application.** You can not interact with the program until the ball has finished bouncing.
- This is not a good situation in theory or in practice, and it is becoming more and more of a problem as networks become more central, it is all too common to be stuck in a time-consuming task that you would really like to interrupt.



# Multithreading

## Using Threads to Give Other Tasks a Chance

---

- Example: BounceThread.java

- class Ball extends Thread
- {
- .....
- public void run()
- { try
- { draw();
- for (int i = 1; i <= 1000; i++)
- { move();
- sleep(5);
- }
- }
- }
- catch (InterruptedException e) {}
- }



# Multithreading

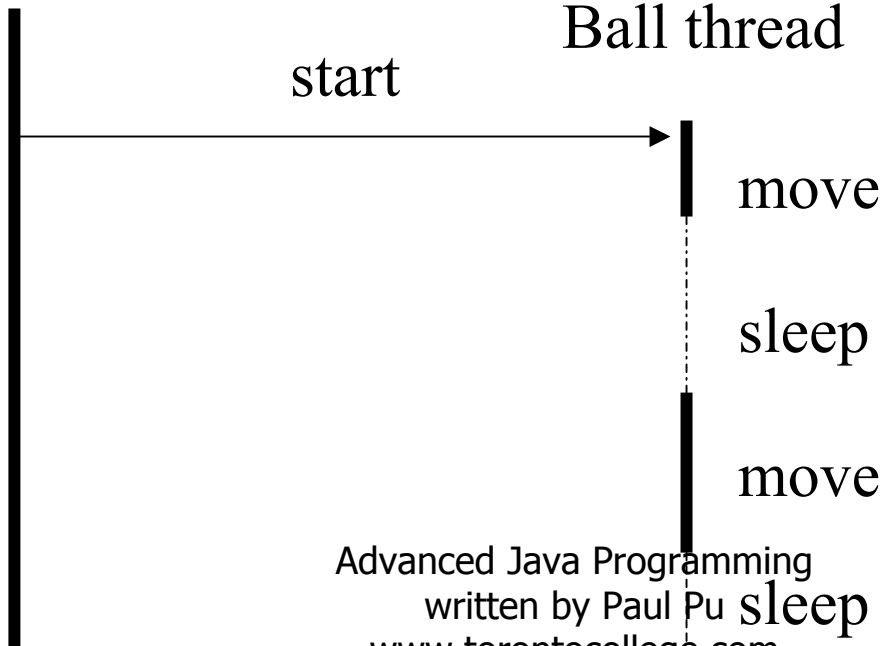
## Using Threads to Give Other Tasks a Chance

---

- In this program, we use two threads: one for the bouncing ball and another for the main thread that takes care of the user interface.

User interface thread

Ball thread





# Multithreading

## Using Threads to Give Other Tasks a Chance

---

- In the java programming language, a thread needs to tell the other threads when it is idle. So the other threads can grab the chance to execute the code in their run procedures. The usual way to do this is through the static sleep method.



# Multithreading

## Java.lang.Thread

---

- Thread() : constructs a new thread. You must start the thread to activate its run method.
- void run(): You must override this function and add the code that you want to have executed in the thread.
- void start(): starts this thread, causing the run() method to be called. This method will return immediately. The new thread runs concurrently.
- static void sleep(long millis): puts the currently executing thread to sleep for the specified number of milliseconds.



# Multithreading

## Java.lang.Thread

---

- boolean `isAlive()`: return true if the thread has started and has not yet terminated.
- void `suspend` : suspends this thread's execution. This method is deprecated.
- void `resume()`: resumes this thread. This method is only valid after `suspend()` has been invoked. This method is deprecated.
- void `stop()`: stops the thread. This method is deprecated.
- void `join()`: waits for the specified thread to cease to be "alive"
- void `join(long millis)`: waits for the specified thread to cease to be "alive" or for the specified amount of milliseconds to have passed.



# Multithreading

## Thread Properties

---

- Thread can be in one of four states:
  - new
  - runnable
  - blocked
  - dead



# Multithreading

## Thread Properties

---

- **New threads:** When you create a thread with the new operator. This thread is not yet running. It is in new state.
- **Runnable threads:** Once you invoke the start method, the thread is runnable. A runnable thread may not yet be running. It is up to operating system to give the thread time to run.



# Multithreading

## Thread Properties

---

- Blocked threads: A thread enters the blocked state when one of the following actions occurs:
  - Someone calls the `sleep()` method of the thread
  - The thread calls an operation that is blocking on input/output.
  - The thread calls the `wait()` method
  - The thread tries to lock an object that is currently locked by another thread.
  - Someone calls the `suspend()` method of the thread.



# Multithreading

## Thread Properties

---

- Moving out of a Blocked State
  - If a thread has been put to sleep, the specified number of milliseconds must expire.
  - If a thread is waiting for the completion of an input or output operation, then the operation must have finished.
  - If a thread called wait, then another thread must call notify or notifyall.
  - If a thread is waiting for an object lock that was owned by another thread, then the other thread must have relinquished the lock.
  - If a thread has been suspended, then someone must call its resume method.



# Multithreading

## Thread Properties

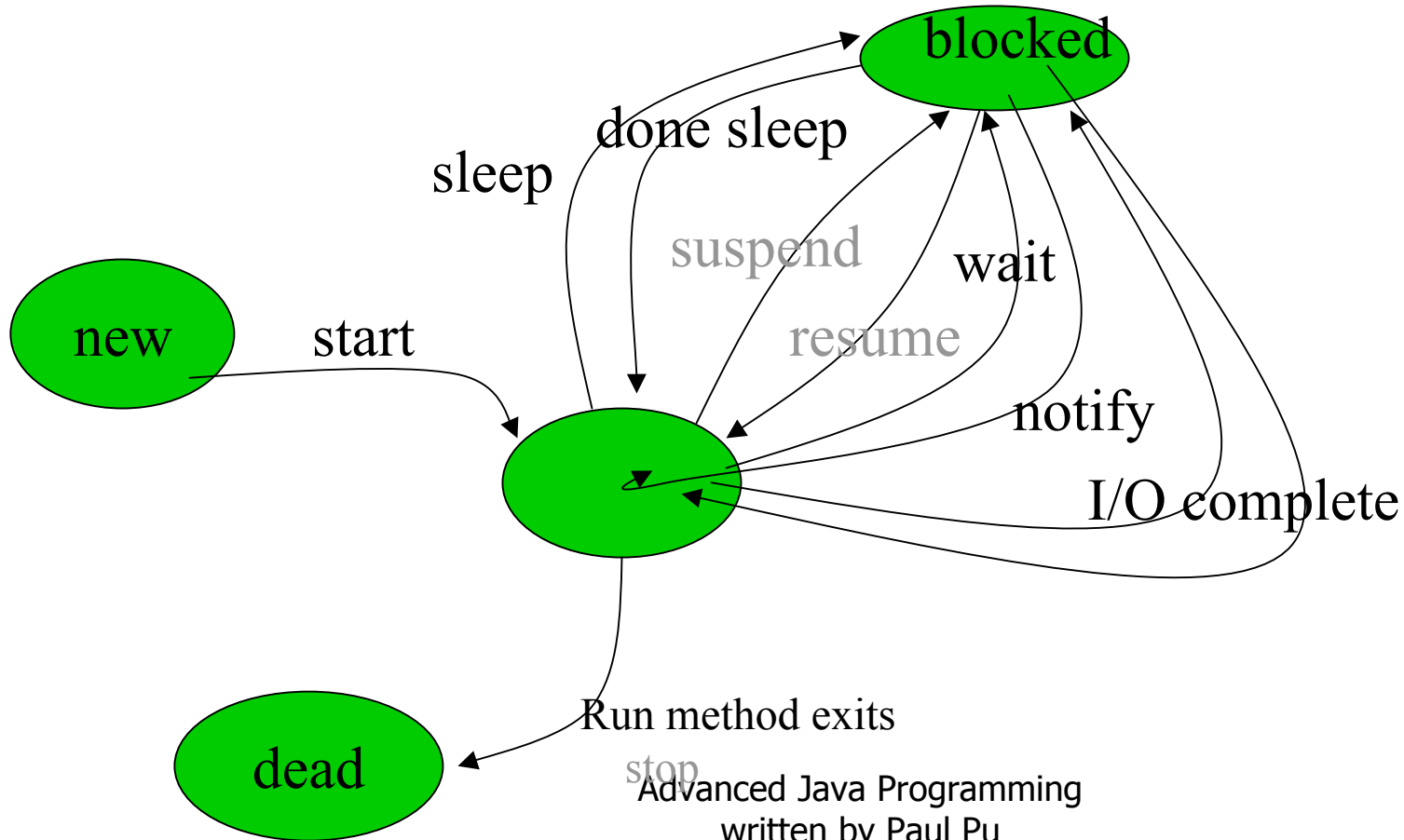
---

- Dead Threads:
  - It dies a natural death because the run method exits normally
  - It dies abruptly because an uncaught exception terminates the run method.



# Multithreading

## Thread Properties





# Multithreading

## Interrupting Threads

---

- When a thread is sleeping, it can not actively check whether it should terminate. This is where the interrupt method comes in. When interrupt method is called on a thread object that is currently blocked, the blocking call(such as sleep or wait) is terminated by an InterruptedException.



# Multithreading

## Interrupting Threads

---

- `public void run(){`
- `Try`
- `{ ....`
- `While(more work to do)`
- `{ do more work}`
- `}`
- `Catch(InterruptedException e){`
- `//thread was interrupted during sleep or wait`
- `}`
- `// exit run method and terminate thread`
- `}`



# Multithreading

## Interrupting Threads

---

- `void interrupt()`: send an interrupt request to a thread. The “interrupted” status of thread is set to true. If the thread is currently blocked by a call to sleep or wait, an `InterruptedException` is thrown.
- Static boolean `interrupted()` : tests whether or not the current thread has been interrupt.
- Boolean `isInterrupted()`: tests whether or not a thread has been interrupted.



# Thread Priorities

---

- Every thread has a priority. You can increase or decrease the priority of any thread with the `setPriority` method:
  - `MIN_PRIORITY`      1
  - `MAX_PRIORITY`     10
  - `MORM_PRIORITY`   5
- Whenever the thread-scheduler has a chance to pick a new thread, it generally picks the highest-priority thread that is currently runnable.



# Thread Priorities

---

- The highest-priority runnable thread keeps running until
  - It yields by calling the `yield` method\
  - It ceases to be runnable (either by dying or by entering the blocked state)
  - A higher-priority thread has become runnable (sleep back, IO complete, someone call its `notify` method)
- The lower-priority threads would have no chance to run if the express threads had called `yield` instead of `sleep`.



# Thread Priorities

---

- Example: BounceExpress.java
- addButton(p, "Start",
- new ActionListener()
- { public void actionPerformed(ActionEvent evt)
- { for (int i = 0; i < 5; i++)
- { Ball b = new Ball(canvas, Color.black);
- b.setPriority(Thread.NORM\_PRIORITY);
- b.start();
- }
- }
- });



# Thread Priorities

---

- addButton(p, "Express",
- new ActionListener()
- { public void actionPerformed(ActionEvent evt)
- { for (int i = 0; i < 5; i++)
- { Ball b = new Ball(canvas, Color.red);
- b.setPriority(Thread.NORM\_PRIORITY + 2);
- b.start();
- }
- }
- }
- });



# Thread Priorities

---

- `java.lang.Thread`
  - `Void setPriority(int newPriority)`
  - `Static int MIN_PRIORITY`
  - `Static int NORM_PRIORITY`
  - `Static int MAX_PRIORITY`
  - `Static void yield()`: Causes the currently executing thread to yield. If there are other runnable threads whose priority is at least as high as the priority of this thread, they will be scheduled next.



# Selfish Thread

---

- Exampe: BounceSelfish.java
- class SelfishBall extends Ball
- { public SelfishBall(JPanel b, Color c) { super(b, c); }
  
- public void run()
- { draw();
- for (int i = 1; i <= 1000; i++)
- { move();
- long t = System.currentTimeMillis();
- while (System.currentTimeMillis() < t + 1)
- ;
- }
- }



# Thread Groups

---

- `String groupName=myGroup;`
- `ThreadGroup g= new ThreadGroup(groupName)`
- `Thread t=new Thread(g,threadName)`



# Multithreading

---

- Lab1: Type and execute the following codes:

```
public class RunnablePotato implements Runnable {
    public void run() {
        while (true)
            System.out.println(Thread.currentThread().getName());
    }
}

public class PotatoThreadTester {
    public static void main(String argv[]) {
        RunnablePotato aRP = new RunnablePotato();
        new Thread(aRP, "one potato").start();
        new Thread(aRP, "two potato").start();
    }
}
```



# Multithreading

---

- Lab2: Rewrite your Dog class as a multithreaded program, and then execute it.
- Please finish the them at lab time and hand in



# Multithreading

Thread safe

---

- **The Problem with Parallelism**

- **if (crucialValue > 0) { ... // think about what to do**
- **crucialValue += 1; }**

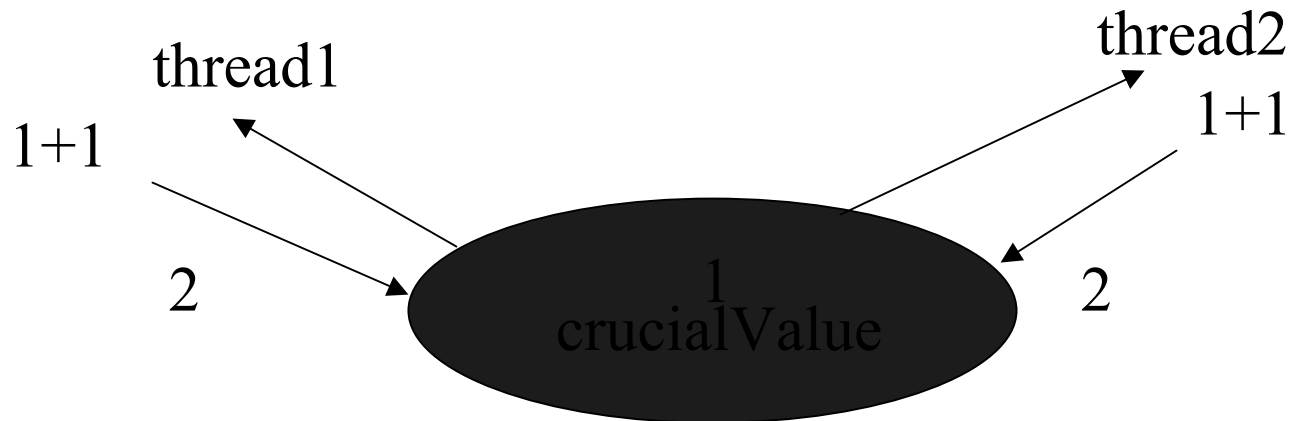
*critical sections*



# Multithreading

## Thread safe

- Even the one line `crucialValue += 1` involves three steps:
- get the current value, add one to it, and store it back.





# Multithreading

## Thread safe

---

- `public class ThreadCounter {`
- `int crucialValue;`
- `public void countMe() {`
- `crucialValue += 1;`
- `}`
  
- `public int howMany() {`
- `return crucialValue; }`
- `}`



# Multithreading

## Thread safe

---

- Why should have problem?
- Java allows you to fix this situation:
- `public class SafeThreadCounter {`
- `int crucialValue;`
- `public synchronized void countMe() {`
- `crucialValue += 1;`
- `}`
- `public int howMany() {`
- `return crucialValue;`
- `}`
- `}`



# Multithreading

Thread safe

---

- The **method howMany()** in the last example doesn't need to be synchronized because it simply returns the current value of an instance variable.



# Multithreading

## thread safe /synchronized keyword

---

- The **synchronized keyword** tells Java to make the block of code in the method **thread safe**. This means that only one thread will be allowed inside this method at once, and others will have to wait until the currently running thread is finished with it before they can begin running it.

# Multithreading

## thread safe /synchronized keyword

---

- 1: public class Point { //redefines class Point from package java.awt
- 2: private float x, y; //OK since we're in a different package here
- 3:
- 4: public float x() { // needs no synchronization
- 5: return x;
- 6: }
- 7:
- 8: public float y() { // ditto
- 9: return y;
- 10: }
- 11: ... // methods to set and change x and y
- 12: }



# Multithreading

## thread safe /synchronized keyword

---

- 13:
- 14: public class UnsafePointPrinter {
- 15: public void print(Point p) {
- 16: System.out.println("The point's x is " +  
p.x()
- 17: + " and y is " + p.y() +  
".");
- 18: }
- 19: }



# Multithreading

## thread safe /synchronized keyword

---

- The methods analogous to `howMany()` are `x()` and `y()`. They need no synchronization because they just return the values of member variables. It is the responsibility of the caller of `x()` and `y()` to decide whether it needs to synchronize itself-and in this case, it does. Although the method `print()` simply reads values and prints them out, it reads *two* values. This means that there is a chance that some other thread, running between the call to `p.x()` and the call to `p.y()`, could



# Multithreading

## thread safe /synchronized keyword

---

- public class TryAgainPointPrinter {
- public void print(Point p) {
- float safeX, safeY;
  
- synchronized(this) {
- safeX = p.x();     // these two lines now
- safeY = p.y();     // happen atomically
- }
- System.out.print("The point's x is " + safeX
- + " y is " + safeY);
- }
- }



# Multithreading

## thread safe /synchronized keyword

---

- have changed the value of x and y stored inside the Point p. Remember, you don't know how many other threads have a way to reach and call methods in this Point object! "Thinking multithreaded" comes down to being careful any time you make an assumption that something has *not* happened between two parts of your program (even two parts of the same line, or the same expression, such as the string + expression in this example).



# Multithreading

## thread safe /synchronized keyword

---

- **TryAgainPointPrinter**
- You could try to make a safe version of print() by simply adding the synchronized keyword modifier to it, but instead, let's try a slightly different approach:



# Multithreading

## thread safe /synchronized keyword

---

- The synchronized statement takes an argument that says what object you would like to lock to prevent more than one thread from executing the enclosed block of code at the same time. Here, you use this (the instance itself), which is exactly the object that would have been locked by the synchronized method as a whole if you had changed print() to be like your safe countMe() method. **You have an added bonus with this new form of synchronization: You can specify exactly what part of a method needs to be safe, and the rest can be left unsafe.**

# Multithreading

## thread safe /synchronized keyword

---

- `public class Point {`
- `private float x, y; . . . // the x() and y() methods` `public synchronized`
- `void setXAndY(float newX, float newY) {`
- `x = newX;`
- `y = newY;`
- `}`
- `}`



# Multithreading

## thread safe /synchronized keyword

---

- **SafePointPrinter**
- The astute reader, though, may still be worried by the last example. It seems as if you made sure that no one executes *your* calls to `x()` and `y()` out of order, but have you prevented the **Point p** from changing out from under you? **If the answer is no**, you still have not completely solved the problem. It turns out that you really do need the full power of the synchronized statement:



# Multithreading

## thread safe /synchronized keyword

---

- public class SafePointPrinter {
- public void print(Point p) {
- float safeX, safeY;
- synchronized(p) {     // no one can change p
- safeX = p.x();     // while these two lines
- safeY = p.y();     // are happening atomically
- }
- System.out.print("The point's x is " + safeX
- + " y is " + safeY);
- }
- }



# Multithreading

## thread safe /synchronized keyword

---

- Now you've got it! You actually needed to protect the Point p from changes, so you lock it by providing it as the argument to your synchronized statement. Now when x() and y() are called together, they can be sure to get the current x and y of the Point p, without any other thread being able to call a modifying method between. **You're still assuming, however, that the Point p has properly protected *itself*. You can always assume this about system classes-but *you* wrote this Point class. You can make sure it's okay by writing the only method that can change x and y inside p yourself:**



# Multithreading

## thread safe /synchronized keyword

---

- By making synchronized the only "set" method in Point, you guarantee that any other thread trying to grab the Point p and change it out from under you has to wait. You've locked the Point p with your synchronized(p) statement, and any other thread has to lock the same Point p via the implicit synchronized(this) statement that is executed when p enters setXAndY().
- **So at last you are thread safe.**



# Multithreading

## thread safe /synchronized keyword

---

- Example: UnsynchBankTest.java

- ```
public void transfer(int from, int to, int amount)
```
- ```
    {  if (accounts[from] < amount) return;
```
- ```
        accounts[from] -= amount;
```
- ```
        accounts[to] += amount;
```
- ```
        ntransacts++;
```
- ```
        if (ntransacts % NTEST == 0) test();
```
- ```
    }
```



# Multithreading

## thread safe /synchronized keyword

---

- `class TransferThread extends Thread`
- `{ public TransferThread(Bank b, int from, int max)`
- `{ bank = b;`
- `fromAccount = from;`
- `maxAmount = max;`
- `}`



# Multithreading

## thread safe / synchronized keyword

---

```
■ public void run()  
■     { try  
■         { while (!interrupted())  
■             { int toAccount = (int)(bank.size() *  
Math.random());  
■                 int amount = (int)(maxAmount *  
Math.random());  
■                 bank.transfer(fromAccount, toAccount,  
amount);  
■                 sleep(1);  
■             }  
■         }  
■     catch (InterruptedException e) {}  
■ }
```



# Multithreading

## thread safe /synchronized keyword

---

- Example: SynchBankTest.java: Synchronizing Access to Shared Resources
- `public synchronized void transfer(int from, int to, int amount)`
- `{ if (accounts[from] < amount) return;`
- `accounts[from] -= amount;`
- `accounts[to] += amount;`
- `ntransacts++;`
- `if (ntransacts % NTEST == 0) test();`
- `}`



# Multithreading

## wait and notify

---

- public synchronized void transfer(int from, int to, int amount)
- { try
- { while (accounts[from] < amount)
- **wait();**
- accounts[from] -= amount;
- accounts[to] += amount;
- ntransacts++;
- **notifyAll();**
- if (ntransacts % NTEST == 0) test();
- }
- catch(InterruptedException e) {}
- }



# Summary of How the synchronization mechanism works

---

- To call a synchronized method, the thread locks the object.
- When a thread executes a call to wait, it surrenders the object lock and enters a wait list
- To remove a thread from the wait list, some other thread must make a call to notifyall or notify



# Multithreading

## Lab 3

---

- Which one statement below is true concerning the following code?
- 1. class Greebo extends java.util.Vector
- 2. implements Runnable {
- 3.     public void run(String message) {
- 4.         System.out.println("in run() method: " +
- 5.         message);
- 6.     }
- 7. }
- 8.
- 9. class GreeboTest {
- 10.     public static void main(String args[]) {
- 12.         Greebo g = new Greebo();
- 13.         Thread t = new Thread(g);
- 14.         t.start();
- 15.     }
- 16. }



# Multithreading

## Lab 3

---

- A) There will be a compiler error, because class Greebo does not correctly implement the Runnable interface.
- B) There will be a compiler error at line 13, because you cannot pass a parameter to the constructor of a Thread.
- C) The code will compile correctly but will crash with an exception at line 13.
- D) The code will compile correctly but will crash with an exception at line 14.
- E) The code will compile correctly and will execute without throwing any exceptions.



# Multithreading

## Lab 3

---

- Which one statement below is always true about the following application?
  - 1. class HiPri extends Thread {
  - 2. HiPri() {
  - 3.     setPriority(10);
  - 4. }
  - 5.
  - 6. public void run() {
  - 7.     System.out.println(
  - 8.         "Another thread starting up.");
  - 9.     while (true) { }
  - 10. }
  - 11.



# Multithreading

## Lab 3

---

- 12. `public static void main(String args[]) {`
- 13. `HiPri hp1 = new HiPri();`
- 14. `HiPri hp2 = new HiPri();`
- 15. `HiPri hp3 = new HiPri();`
- 16. `hp1.start();`
- 17. `hp2.start();`
- 18. `hp3.start();`
- 19. `}`
- 20. `}`



# Multithreading

## Lab 3

---

- A) When the application is run, thread hp1 will execute; threads hp2 and hp3 will never get the CPU.
- B) When the application is run, all three threads (hp1, hp2, and hp3) will get to execute, taking time-sliced turns in the CPU.
- C) Either A or B will be true, depending on the underlying platform.



# Multithreading

## Lab 3

---

- True or false: A thread wants to make a second thread ineligible for execution. To do this, the first thread can call the `yield()` method on the second thread.
- A. True
- B. False



# Multithreading

## Lab 3

---

- A thread's run() method includes the following lines:
  1. try {
  2. sleep(100);
  3. } catch (InterruptedException e) { }
- Assuming the thread is not interrupted, which one of the following statements is correct?



# Multithreading

## Lab 3

---

- A) The code will not compile, because exceptions may not be caught in a thread's run() method.
- B) At line 2, the thread will stop running. Execution will resume in, at most, 100 milliseconds.
- C) At line 2, the thread will stop running. It will resume running in exactly 100 milliseconds.
- D) At line 2, the thread will stop running. It will resume running some time after 100 milliseconds have elapsed.



# Multithreading

## Lab 3

---

- A monitor called `mon` has 10 threads in its waiting pool; all these waiting threads have the same priority. One of the threads is `thr1`. How can you notify `thr1` so that it alone moves from the Waiting state to the Ready state?
- A) Execute `notify(thr1);` from within synchronized code of `mon`.
- B) Execute `mon.notify(thr1);` from synchronized code of any object.
- C) Execute `thr1.notify();` from synchronized code of any object.
- D) Execute `thr1.notify();` from any code (synchronized or not) of any object.
- E) You cannot specify which thread will get notified.



# Multithreading

## Lab 3

---

- If you attempt to compile and execute the application listed below, will it ever print out the message In xxx?
  - 1. class TestThread3 extends Thread {
  - 2. public void run() {
  - 3. System.out.println("Running");
  - 4. System.out.println("Done");
  - 5. }
  - 6.
  - 7. private void xxx() {
  - 8. System.out.println("In xxx");
  - 9. }



# Multithreading

## Lab 3

---

- 10.
- 11. `public static void main(String args[]) {`
- 12.     `TestThread3 ttt = new TestThread3();`
- 13.     `ttt.xxx();`
- 14.     `ttt.start();`
- 12.     `}`
- 13.     `}`



# Multithreading

## Lab 3

---

- A) Yes
- B) No



# Multithreading

## DeadLocks

---

- The synchronization feature is convenient and powerful.
- But it cannot solve all problems that might arise in multithreading.
- Account 1: \$2000
- Account 2: \$3000
- Thread 1: Transfer \$3,000 from Account 1 to Account 2
- Thread 2: Transfer \$4000 from account 2 to Account 1
- This will cause **deadlock**.



# Multithreading DeadLocks

---

- Creating a deadlock
  - SynchronBankTest Program: change the transaction limit from 10,000 to 14,000
  - Or SynchronBankTest Program: change notifyall to notify



# Multithreading

## DeadLocks

---

- public class Deadlock implements Runnable{
- private Object firstResource;
- private Object secondResource;
- public static void main(String[] args) {
- Object a="Resource A";
- Object b="Resource B";
- Thread t1=new Thread(new Deadlock(a,b));
- Thread t2=new Thread(new Deadlock(b,a));
- t1.start();
- t2.start();
- }



# Multithreading

## DeadLocks

---

- `public Deadlock(Object first, Object second) {`
- `firstResource=first;`
- `secondResource=second;`
- `}`



# Multithreading

## DeadLocks

---

- `public void run(){`
- `for(;;){`
- `System.out.println(`
- `Thread.currentThread().getName()+"Looking for lock on`  
`"+firstResource);`
- `synchronized(firstResource){`
- `System.out.println(`
- `Thread.currentThread().getName()+"Obtained lock on`  
`"+firstResource);`
- `System.out.println(`
- `Thread.currentThread().getName()+"Looking for lock on`  
`"+secondResource);`



# Multithreading DeadLocks

---

- synchronized(secondResource){
- System.out.println(  
■         Thread.currentThread().getName()+"Obtained lock on  
■     "+secondResource);
- System.out.println(  
■         Thread.currentThread().getName()+"Looking for lock on  
■     "+secondResource);
- try {Thread.sleep(100);}
- catch(InterruptedException ex) {}
- }
- }
- }
- }
- }



# Multithreading

## Why the stop and suspend Methods are Deprecated

---

- Stop: When a thread is stopped, it immediately gives up the locks on all objects that it has locked. This can leave objects in an inconsistent state.
- Suspend: If you suspend a thread that owns a lock to an object, then the object is unavailable until the thread is resumed.



# Multithreading

## Scheduling Implementations

---

- Historically, two approaches have emerged for implementing thread schedulers
  - Preemptive scheduling
  - Time-sliced

In preemptive scheduling, there are only two ways for a thread to leave the Running state without explicitly calling a thread-scheduling methods such as `wait()` or `suspend()`:

- ✦ It can cease to be ready to execute (for example, by calling a blocking I/O method)
- ✦ It can get moved out of the CPU by a higher-priority thread that becomes ready to execute



# Multithreading

## Scheduling Implementations

---

- With time slicing, a thread is only allowed to execute for a limited amount of time. It is then moved to the Ready state. Where it must contend with all the other ready threads.
- Time slicing insures against the possibility of a single high-priority thread getting into the Running state and never getting out. Preventing all other threads from doing their jobs. Unfortunately, time slicing creates a non-deterministic system; at any moment, you cannot be certain which thread is executing or for how long it will continue execute.



# Multithreading

## Scheduling Implementations

---

- It is natural to ask which implementation Java uses.
- The answer is that it depends on the platform; the java specification gives implementations a lot of leeway.



# Multithreading

## Monitors, wait(), notify

---

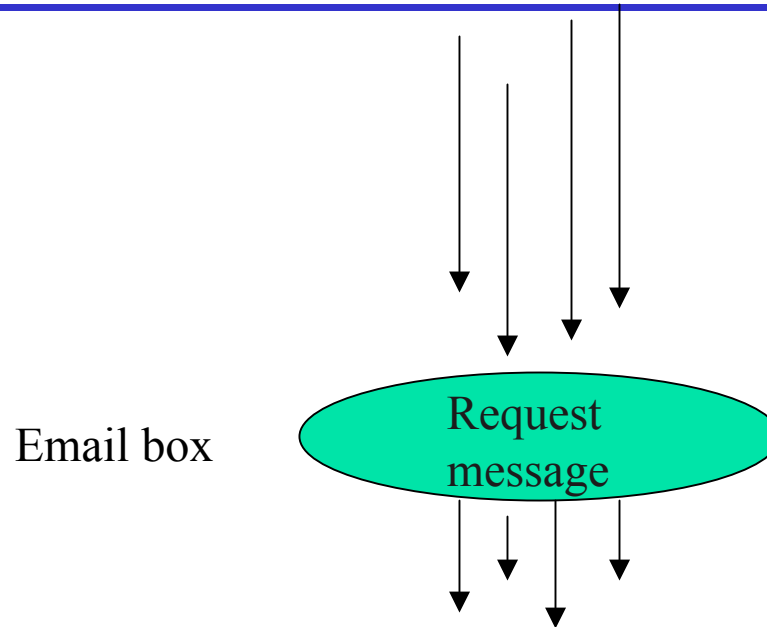
- A monitor is an object that can block and revive threads.
- The reason for having monitors is that sometimes a thread cannot perform its job until an object reaches a certain state



# Multithreading

## Monitors, wait(), notify

---



Logical:

A thread will come to check request. When request is true, the thread should write message to System.out, and then set request to false.

Setting request to false indicates that the mailbox object is ready to handle another request



# Multithreading

## Monitors, wait(), notify

---

- package mailbox;
- class Mailbox {
- public boolean request;
- public String message;
- }



# Multithreading

## Monitors, wait(), notify

```
■ package mailbox;
■ public class Consumer extends Thread {
■     private Mailbox myMailbox;
■     public Consumer(Mailbox box) {
■         this.myMailbox=box;
■     }
■     public void run() {
■         while (true) {
■             10         if(myMailbox.request){
■             11             System.out.println(myMailbox.message);
■                 myMailbox.request=false;
■             }
■         try {
■             sleep(50);
■         }
```



# Multithreading

## Monitors, wait(), notify

---

- catch(InterruptedException e) { }
- }
- }
- }



# Multithreading

## Monitors, wait(), notify

- 
- package mailbox;
  - public class Producer extends Thread {
  - private Mailbox myMailbox;
  - public Producer(Mailbox box) {
  - this.myMailbox=box;
  - }
  - public void run() {
  - while (true) {
  - if(!myMailbox.request){
  - myMailbox.message="This is my letter:"+Math.round(Math.random()\*100);
  - myMailbox.request=true;
  - }
  - try { sleep(5000);}
  - }
  - }
  - catch(InterruptedException e) { }
  - }
  - }
  - }



# Multithreading

## Monitors, wait(), notify

---

- The problems:
  - The consumer class accesses data internal to the Mailbox class, introducing the possibility of corruption. On a time-sliced system, the consumer thread could just possibly be interrupted between lines 10 and 11. The interrupting thread could just possibly be a client that sets message to its own message (ignoring the convention of checking request to see if the handler is available). The consumer thread would send the wrong message.
  - The choice of 50 milliseconds for the delay can never be ideal. Some times, 50 milliseconds will be too long, and clients will receive slow service; sometimes 50 milliseconds will be too frequent, and cycles will be wasted.



# Multithreading

## Monitors, wait(), notify

---

- Solutions:

Ideally, these problems would be solved by making some modifications to the Mailbox class:

- ✍ The mailbox should be able to protect its data from irresponsible clients
- ✍ If the mailbox is not available (this is, if the request flag is already set, then a client consumer should not have to guess how long to wait before checking the flag again. The handler should tell the client when the time is right.



# Multithreading

## Monitors, wait(), notify

---

- package mailbox;
- public class CheckMail {
  - public static void main(String[] args) {
  - Mailbox myMailbox=new Mailbox();
  - Producer p=new Producer(myMailbox);
  - Consumer c=new Consumer(myMailbox);
  - p.start();
  - c.start();
  - }
  - }



# Multithreading

## The Class Lock

---

- Class X{
- static int x,y;
- static synchronized void foo(){
- x++;
- y++;
- }
- }