

GUIs and Threads

IAT351

Week 11 Lecture
26.03.2008

Lyn Bartram
lyn@sfu.ca



SCHOOL OF INTERACTIVE
ARTS + TECHNOLOGY

Today's agenda

- Administrivia
- Introduction to threads
 - Why do we need them?
 - How do they work?
- Multi-user programming, servers, etc.
- GUI programming and the Java system threads



Administrivia

- Assignments
- Workshop this Friday: work on project or assignment 5
- Final project and course schedule
 - Lyn is away April 2-10
 - Final projects demos and papers due Week 15
 - We need to set a date
- Teaching evaluations

Introduction to Concurrency

- Computers can perform many tasks concurrently – download a file, print a file, receive email, etc.
 - Each application and service is made up of one or more running processes, or tasks
 - Each process has its own distinct data and object space, called its *address space*
 - ensures that the data structures and state of the process remain intact from changes in other processes
 - Interact through well-defined communication mechanisms like message passing or shared address space
- Modern operating systems allow multiple processes to run “at the same time”
 - Sophisticated mechanisms for managing a process’s time on the CPU
 - *Scheduling* according to time and priority schemes
 - Multitasking



Multitasking vs. Multithreading

- Multitasking operating systems run multiple programs simultaneously.
 - O.S is responsible for splitting the time among the different programs that are running
- Once systems allowed different users to run programs at the same time it was a short step to letting the same user run multiple programs simultaneously.
- But:
 - It's complicated and expensive for 2 processes to interact closely
 - Sometimes, just like a user needs to run several things at once, a program needs to do several things simultaneously
- *Multithreading* enables a program to accomplish multiple simultaneous tasks.
 - Threading system provides support for scheduling and management
 - Developer has to write the code to deploy it



Introduction to Threads

- Sequential languages such as C and C++ do not allow the specification of concurrent processes.
 - Need a special threading subsystem
- Java (and C#), on the other hand, offers primitives for turning a program into separate, **independently-running** subtasks.
- Each of these independent subtasks is called a **thread**
- You program as if each thread runs by itself and has the CPU to itself.
- Each thread shares the address space of the process
 - Very efficient
 - Very dangerous!



Overview of Multitasking

- Each program has at least one thread within it.
- Single thread in a single process
- Process begins execution at a well-known point. (i.e main ())
- Execution of statements follow a predefined order.
- During execution process has access to certain data: thread's stack->local variables
object references->instance variables
class or object references-> static variables

A single threaded program

```
class ABC
{
  ....
  public void main(..)
  {
    ...
    ..
  }
}
```

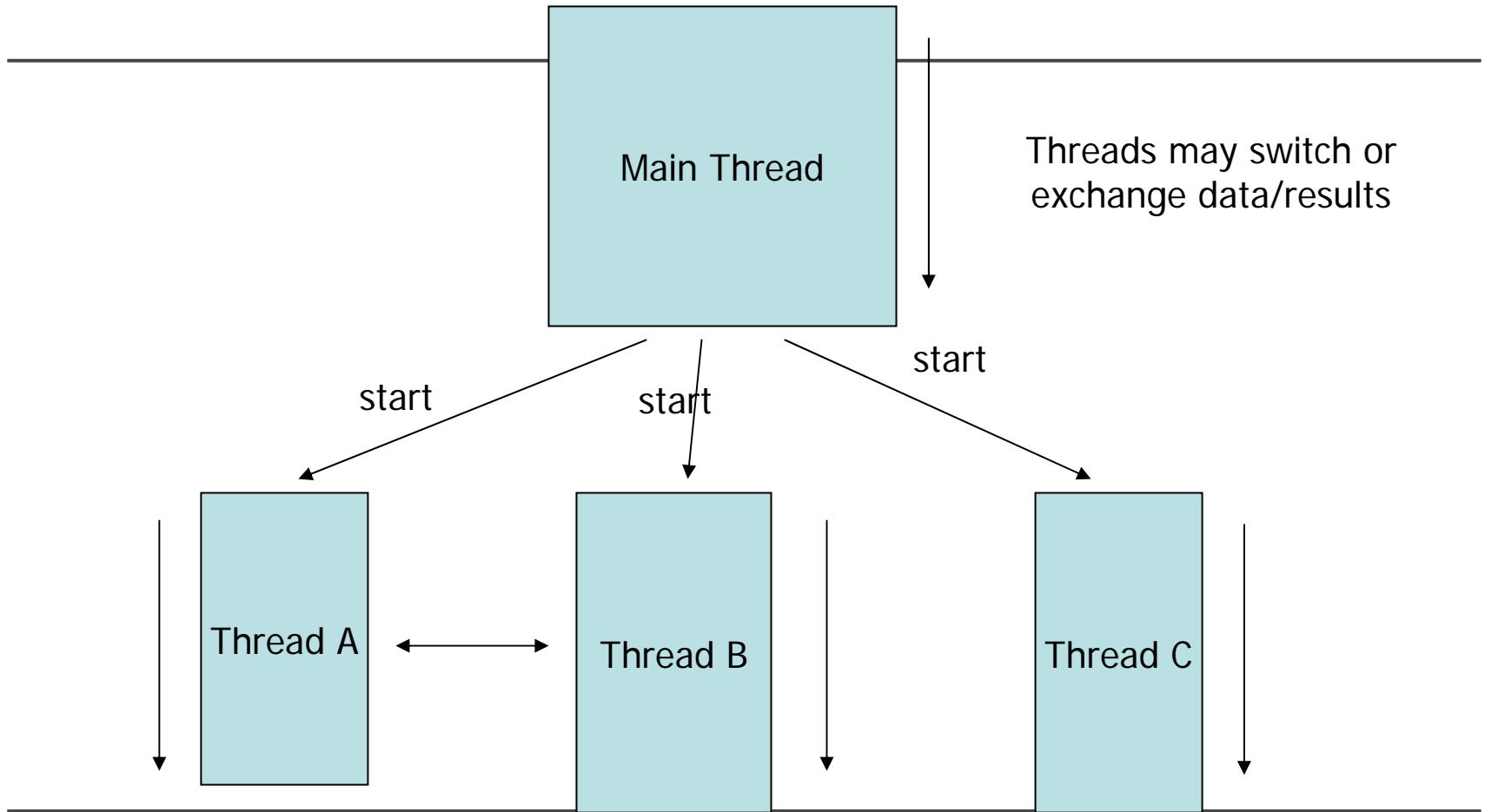
begin

body

end

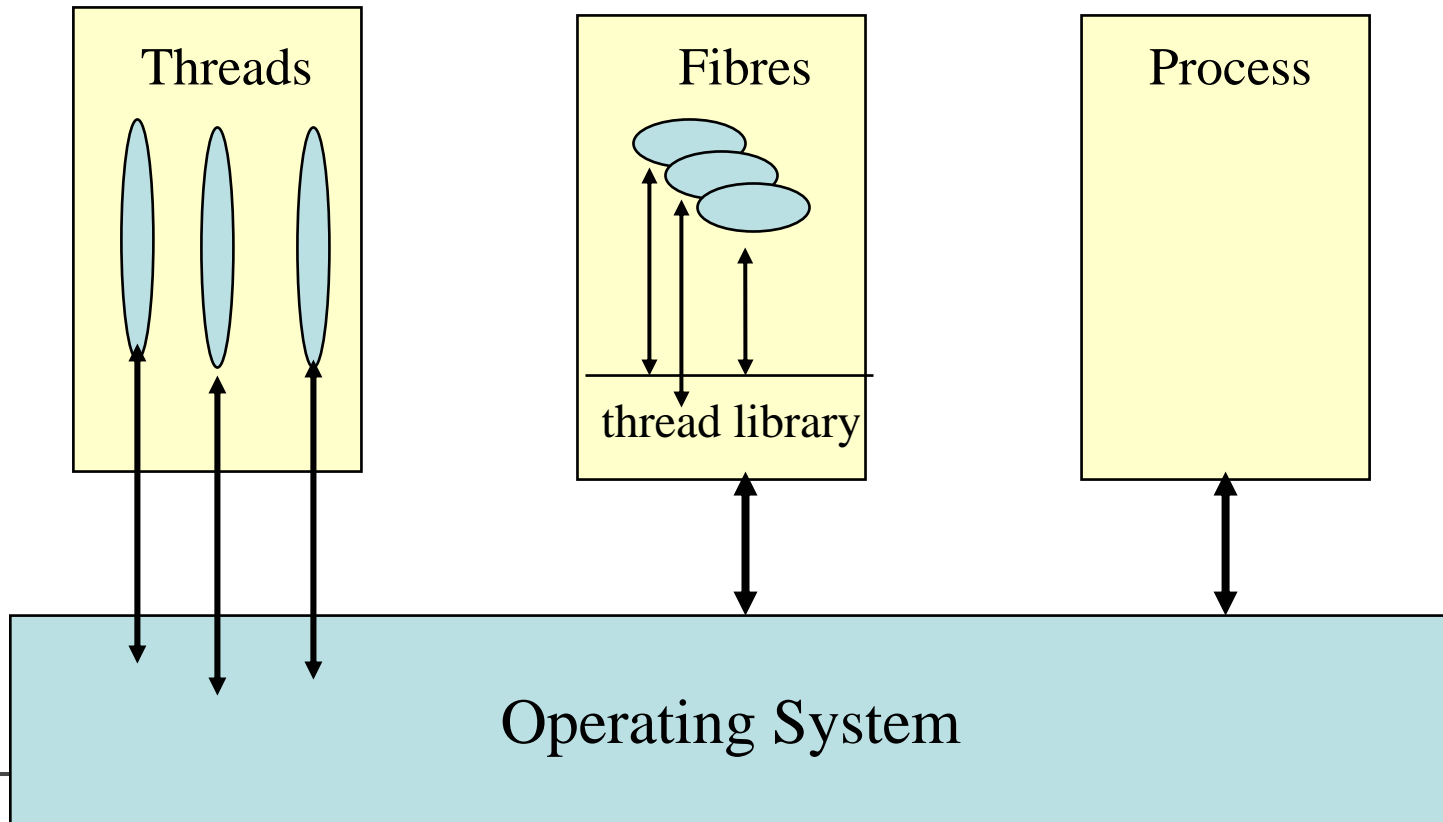


A Multithreaded Program



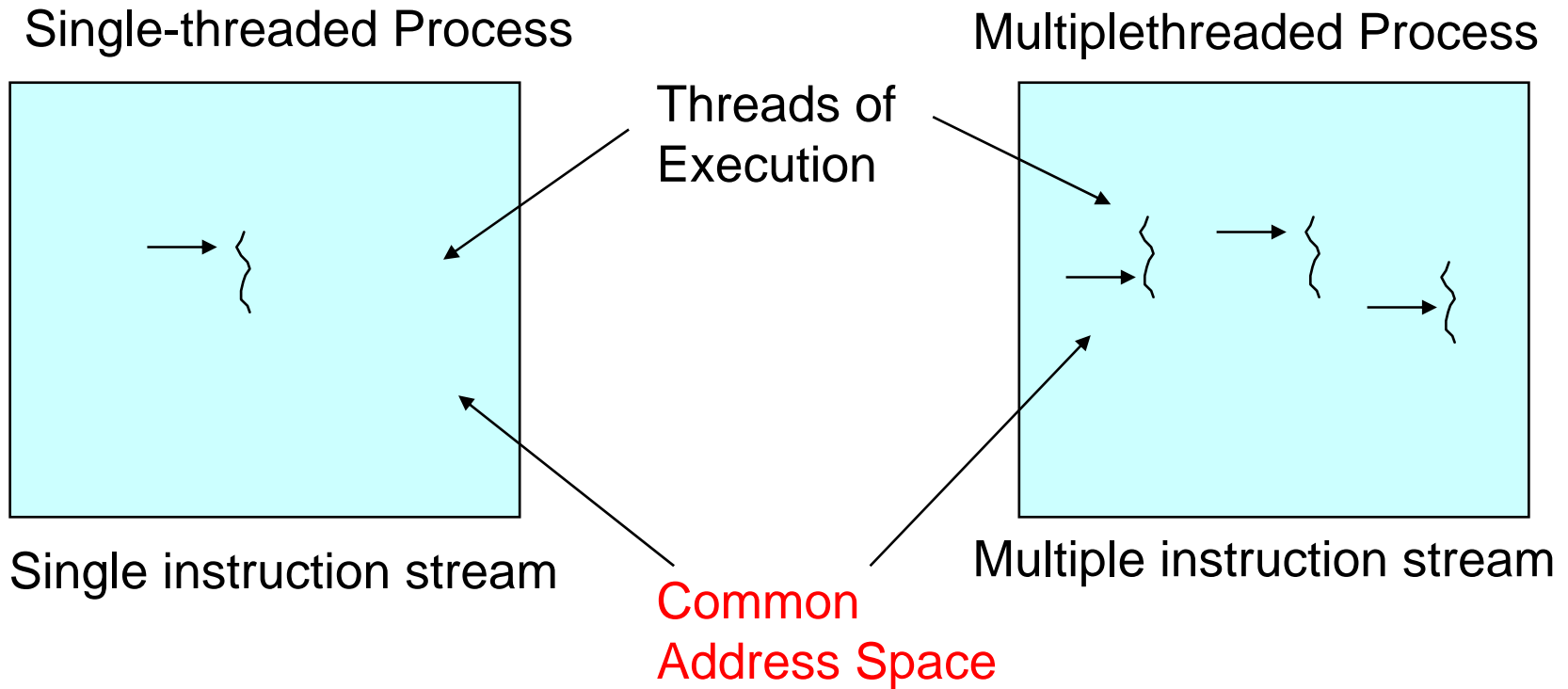
Concurrency Models I

- Processes versus Threads



Single and Multithreaded Processes

threads are light-weight processes within a process

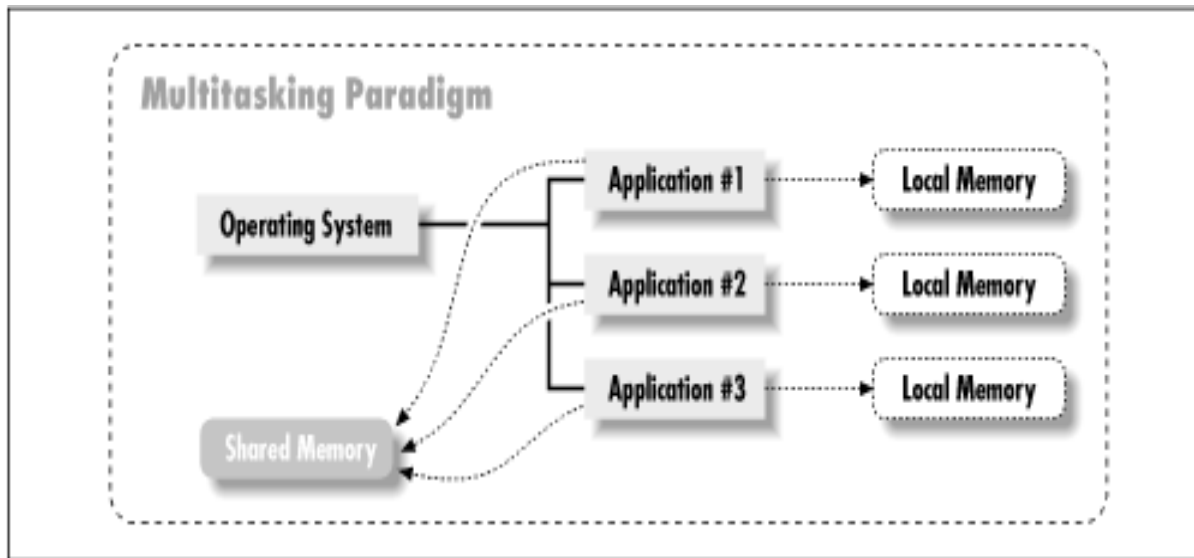


What's a Thread?

- **A *process* is an executing program**
 - memory allocated by OS
 - usually no memory sharing between processes
- **A *thread* is a single sequential flow of control**
 - runs in the address space of a process
 - has its own program counter and its own stack frame
- A thread is a path of execution through a program.
- Single threaded programs->one path of execution
- Multiple threaded programs -> two or more
- Threads run in methods or constructors. The threads go into the methods and follow their instructions.

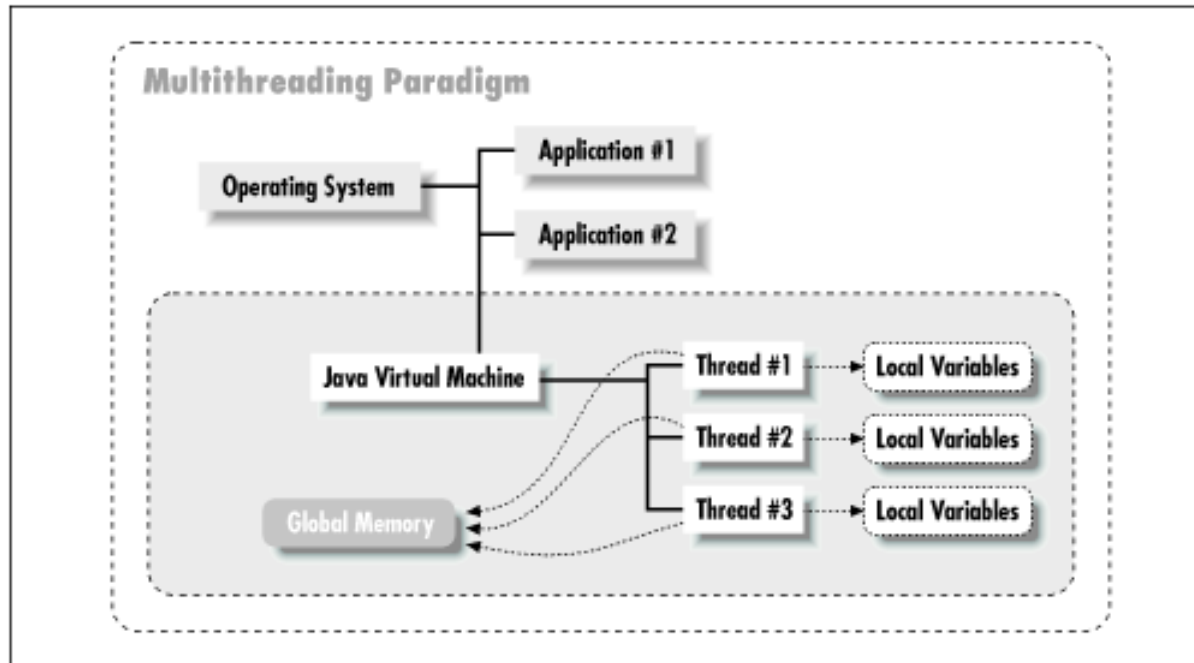
Processes in a Multitasking Environment

- Data within processes is separated; separate area for local variables and data area for objects



Overview of Multithreading

- Analogy : thread \rightarrow process
- Multiple thread running within a single instance of JVM \rightarrow multiple processes within an OS



Initial Remarks about Threads

- Cheaper in computing resources and creation , a thread can only act within a single process.
- Java threads are built into the language
- Threads are "light-weight" processes (unlike UNIX processes), which communicate by a combination of shared memory and message passing.
- BECAUSE threads share data (address) space, you have to be careful about managing access and scheduling execution
- Sharing resources between threads is done with *synchronization*.
- This communication mechanism is employed naturally by Java
- Java threads are based on a *locking* mechanism using *monitors* for synchronization
- Scheduling can be left to automatic (OS) or quasi-preemptive based on priority



Properties of Multiple Threads

- Each thread executes code independently of others in the process/program.
- mechanisms of threads allow cooperation.
- The threads appear to have a certain degree of simultaneous execution.
- Access to various types of data
- Each thread is separate, so that local variables in the methods are separate.
- Objects and their instance variables can be shared between threads in a Java program.
- Static variables are automatically shared .

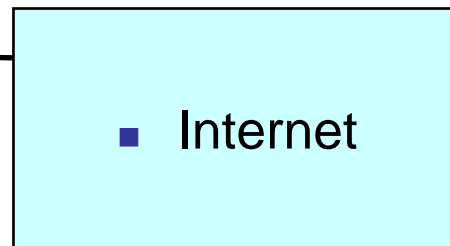


Multithreaded Server: For Serving Multiple Clients Concurrently

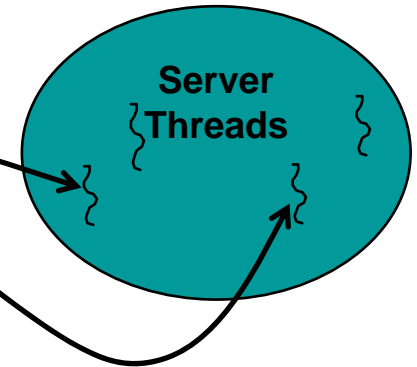
Client 1 Process



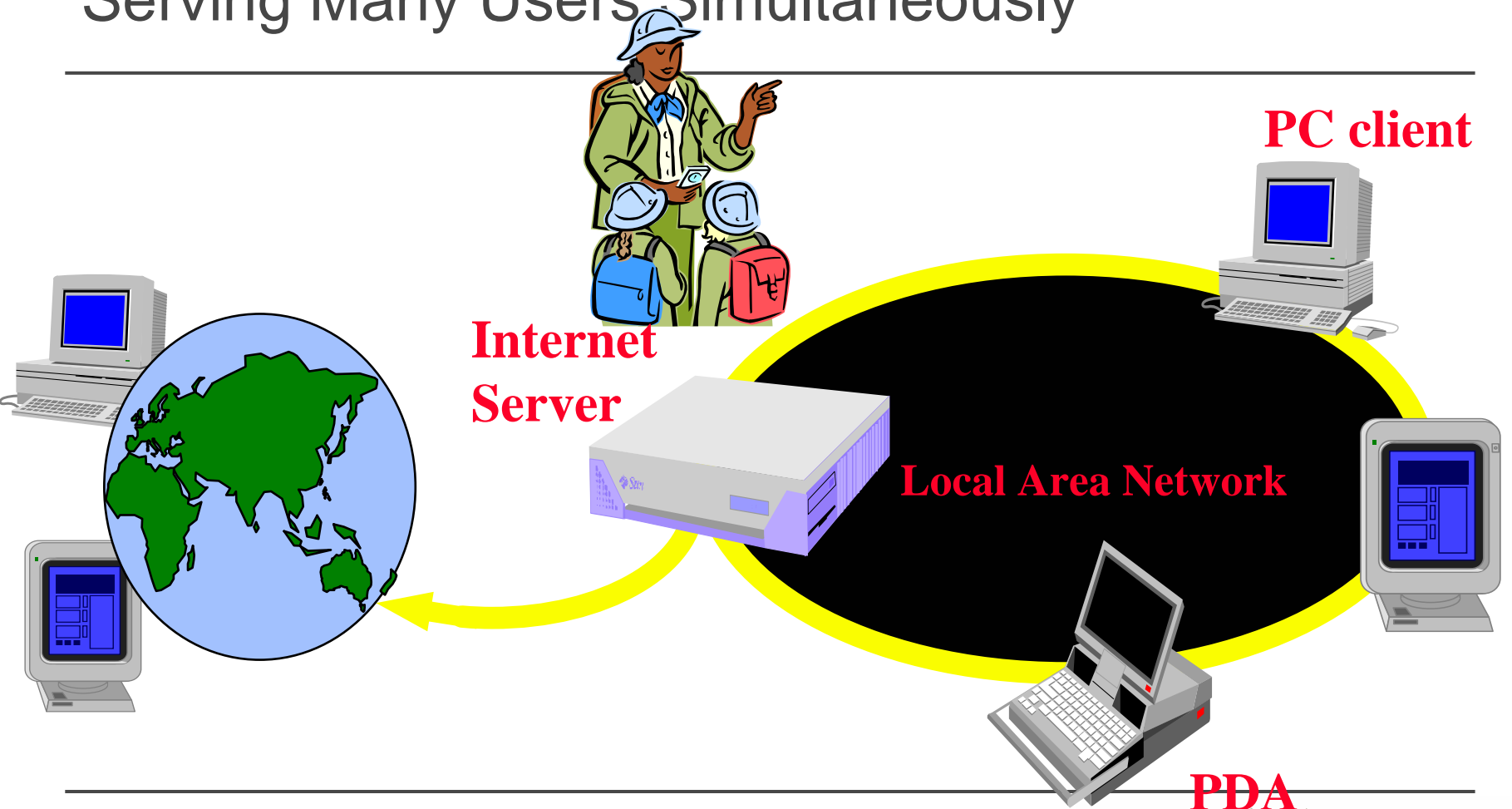
Client 2 Process



Server Process



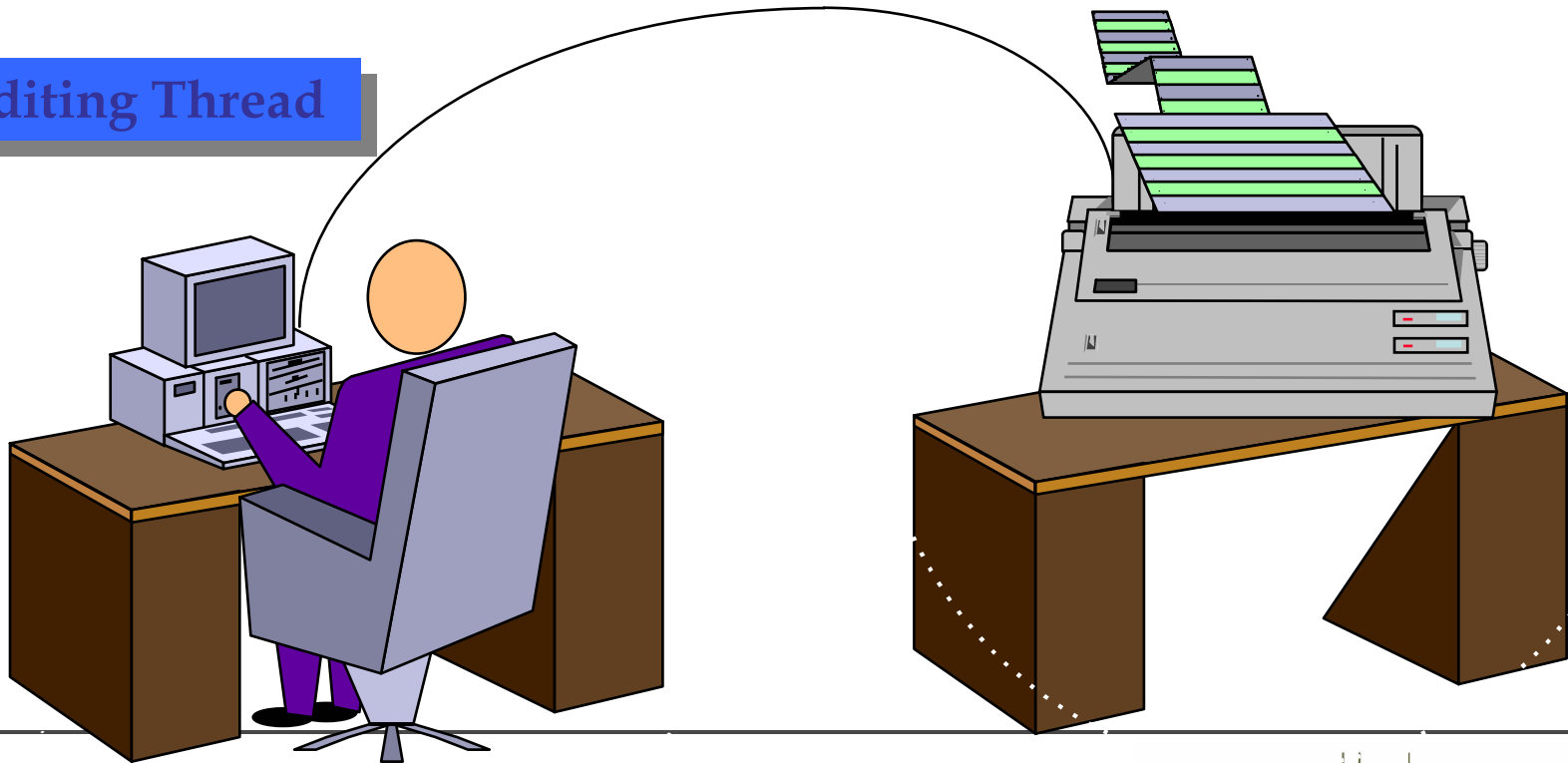
Web/Internet Applications: Serving Many Users Simultaneously



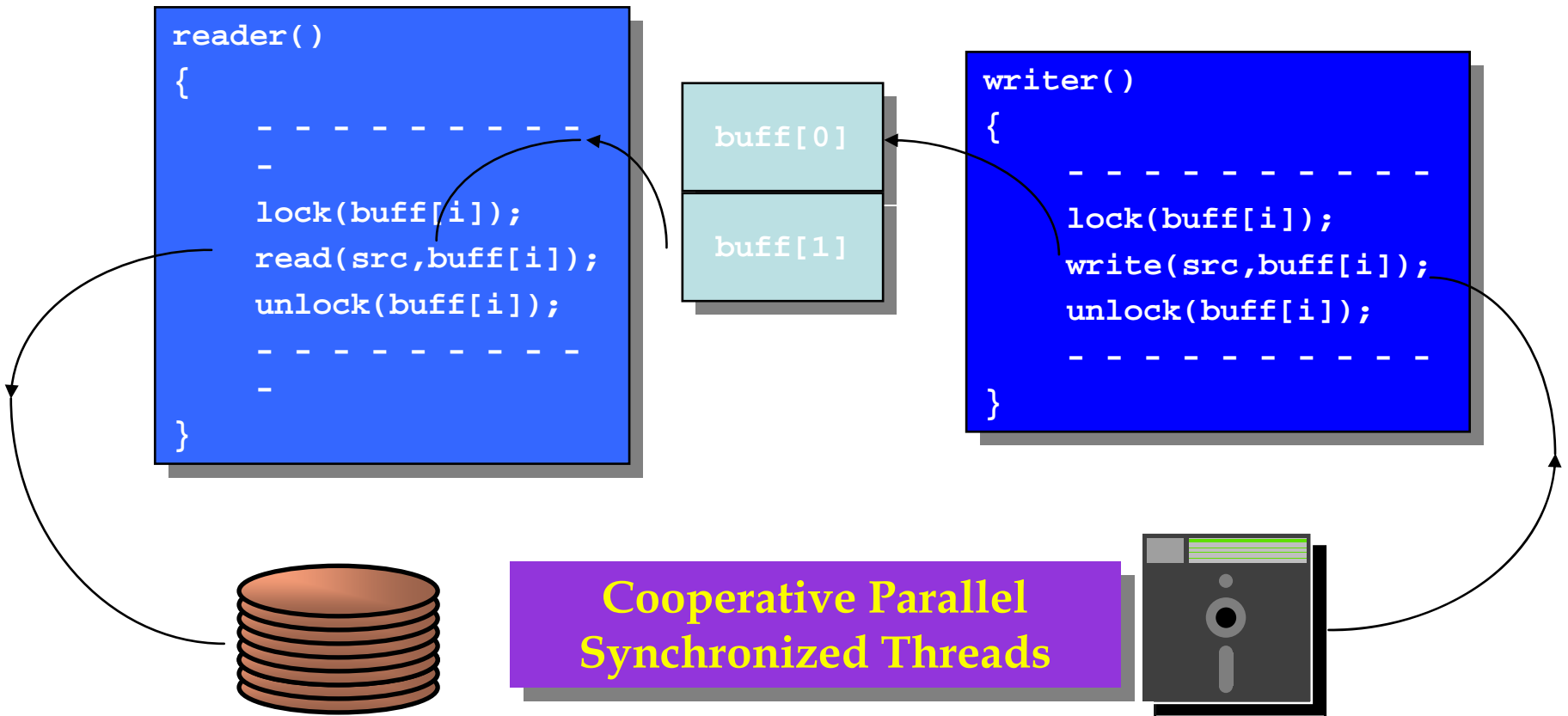
Modern Applications need Threads (ex1):
Editing and Printing documents in background.

Printing Thread

Editing Thread



Multithreaded/Parallel File Copy



Uses for threads

- Long initiations (in applets that take a while to initialize)
 - Repetitive or timed tasks (animations)
 - Asynchronous events (event handling such as a mouse click)
 - Multiple Tasks (To do more than one thing at once)
 - Java applications and applets are naturally threaded.
-
- Can you think of others?
-
- Where do you think you are using threads already?

More Uses for Multithreading

- In general, you'll have some part of your program **tied to a particular event or resource** (and you don't want to hang up the rest of your program because of that).
- So you create a thread associated with that event or resource and let it **run independently** of the main program.
- A good example could be a “quit” button
you don't want to be forced to poll the quit button in every piece of code you write in your program and yet you want the quit button to be responsive, as if you were checking it regularly.
- In fact, one of the most immediately compelling reasons for multithreading is to produce **a responsive user interface.**

Implementing Threads

- One can implement threads in two ways:
 - First, by subclassing the `Thread` class
 - Second, by implementing the `Runnable` interface
- The `Runnable` interface allows you to add threading to a class which cannot conveniently extend `Thread`.
- A class that implements the `Runnable` interface (including the `Thread` class itself) must implement the `run()` method containing the "body" of the thread.
 - `public abstract void run()`

Subclassing the Thread Class

- The simplest way to create a **thread** is to inherit from **class Thread**, which has everything necessary to create and run threads.
- Overview of thread-related methods
 - Constructors
 - Thread(threadName)**
 - Thread()**
Creates an auto numbered **Thread** of format **Thread 1,Thread2..**
 - **run**
"Does work" of thread
Can be overridden in a subclass of **Thread** or in a **Runnable** object

Java.Lang.thread

- Instances of this subclass are instantiated like this:
- `MyThread mt = new MyThread();`
- Thread control:
 - When the thread is created, it does not automatically start running. The class that creates it must call the Thread method **start()**.
 - Other methods may be called: **static method Thread.sleep(), static method Thread.yield(), suspend(), resume(), stop(), interrupt(), join(), interrupted(), isInterrupted(), isAlive(), setName(threadName), getName, toString, currentThread**

A simple thread

```
public class BytePrinter extends
Thread {
public void run()
{
for (int b = -128;
    b < 128; b++)
{ System.out.println(b);
}
}}
```

```
public class ThreadTest
{
public static void main(String args[]) {
System.out.println("Constructing the thread...");
BytePrinter bp = new BytePrinter();
System.out.println("Starting the thread...");
bp.start();
System.out.println("The thread has been started.");
System.out.println("The main() method is
finishing.");
return; } }
```

```
java ThreadTest Constructing the thread... Starting the thread... The thread has been started. The main() method is
finishing. -128 -127 -126 -125 -124 -123 -122 -121 -120 ..
```

Multiple Threads

- The following program launches three BytePrinter threads:

```
public class ThreadsTest {
    public static void main(String args[])
    {
        BytePrinter bp1 = new BytePrinter();
        BytePrinter bp2 = new BytePrinter();
        BytePrinter bp3 = new BytePrinter();
        bp1.start();  bp2.start();  bp3.start();  } }
```

- Output could be as follows;
 - -128,...,127,-128,...,127, -128,...,127
 - In a preemptive system; -128 -127 -126 -125 -124 -123 -128 -127 -126 -125 -128 -127 -126 -125 -124 -123 -122 -121 -120 -119 -124 , -123,..

Multithreading Issues: Thread Scheduling

- Threads are implemented by a scheduler in Java, which asks the local operating system to run threads in the "runnable" state.
- Threads have **priority** from 1 to 10
 - `Thread.MIN_PRIORITY` = 1
 - `Thread.NORM_PRIORITY` = 5 (default)
 - `Thread.MAX_PRIORITY` = 10
- New threads inherit priority of thread that created it
- Timeslicing
 - Each thread gets a quantum of processor time to execute
 - After time is up, processor given to next thread of equal priority (if available)
 - Without timeslicing, each thread of equal priority runs to completion

Thread Execution

- On MacOs :
 - A thread runs to completion or until a higher priority thread becomes ready
 - Preemption occurs (processor is given to the higher priority thread)
- On Win32
 - Threads are timesliced
 - Thread given quantum of time to execute
 - Processor then switches to any threads of equal priority
 - Preemption occurs with higher and equal priority threads

Multithreading Issues: Resources

- **Sharing resources:**

- A single-threaded program may be thought of as **one** lonely entity moving around through the problem space and doing one thing at a time.
- With multithreading two or more threads may coexist and possibly try to use the same resource simultaneously.
- Colliding over a resource must be prevented or else two threads may try to access the same resource at the same time with undesirable results

(e.g. print to the same printer, or adjust the same value, etc.)

Multithreading Issues

- A **fundamental problem** with using threads:
No knowledge about when a thread might be run.
- Neglecting the fact that a resource might be accessed at the same time you're using it.
- Need some way to prevent two threads from accessing the same resource. (at least during critical periods)
- Preventing this kind of collision is simply a matter of putting a **lock on** a resource when one thread is using it.
- The first thread that accesses a resource **locks it**, and then the other threads cannot access that resource **until it is unlocked**.

How Java Shares Resources

- Java has built-in support to prevent collisions:
 - **Methods** can be declared as **synchronized**.
 - Only one thread at a time can call a **synchronized** method of a particular object.
 - Each object contains a single **lock** (also called a monitor). When you call a **synchronized** method of an object, that object is locked and no other **synchronized** method of that object can be called until the first one finishes and releases the lock.
 - Java also has **synchronized blocks of code**. Which object is locked when such code is executed? The current object (**this**).

Synchronization is implemented by Monitors

- An object that can block threads and notify them when it is available is called a **monitor**. A monitor is associated with an instance of the class; it has a **lock** and a **queue**.
- If a class has one or more synchronized methods, each object (instance) of the class has a monitor. The queue holds all threads waiting to execute a synchronized method.
 - A thread enters the queue by calling `wait()` inside the method or when another thread is already executing the method.
 - When a synchronized method returns, or when a method calls `wait()`, another thread may access the object.
 - The scheduler chooses the highest-priority thread among those in the queue.
 - If a thread is put into the queue by calling `wait()`, it can't be scheduled for execution until some other thread calls `notify()`.

Threads and Synchronization

- If a thread must wait for the state of an object to change, it should call `wait()` inside a synchronized method.
 - `void wait()`
 - `void wait(int timeout)`
- These methods cause the thread to wait until notified or until the timeout period expires, respectively.
- The timeout argument is optional. If missing or zero, the thread waits until either `notify()` or `notifyAll()` is called.
 - *`wait()` is called by the thread owning the lock associated with a particular object; `wait()` releases this lock (atomically, i.e., safely)*

Threads and Synchronization

- `void notify()`
- `void notifyAll()`
 - These methods must be called from a synchronized method.
 - These methods notify a waiting thread or threads.
- `notify()` notifies the thread associated with the given synchronization object that has been waiting the longest time
- `notifyAll()` notifies all threads associated with the given object and is therefore safer than `notify()`
- One can mark a variable as "threadsafe" to inform the compiler that only one thread will be modifying this variable

Threads and Synchronization Example

- Suppose that several threads are updating a bank balance (i.e., several threads can access one instance of class Account below). Then a thread that finds insufficient funds to debit an account can wait until another thread adds to the account:

```
public class Account
{ int bankBalance; ...
```

```
public synchronized void
DebitAcct (int amt)
{ while ((bankBalance - amt) <
0) wait();
bankBalance -= amt;
... }
```

```
public synchronized void
CreditAcct (int amt)
{ bankBalance += amt;
notify(); ...
} }
```

Example 2

```
public synchronized void
    addItem(int i){

while (size==length)
    try {
        wait();}
    catch (InterruptedException
        e){}

    end = end⊕1;
    buffer[end] = i;
    size++; notify;
}
```

```
public synchronized int
    removeItem(){
    int value;

while (size == 0)
    try {
        wait();}
    catch (InterruptedException e)
        {}

    value = buffer[front];
    front = front⊕1;
    size --; notify;
    return value;
}
```

Runnable Interface

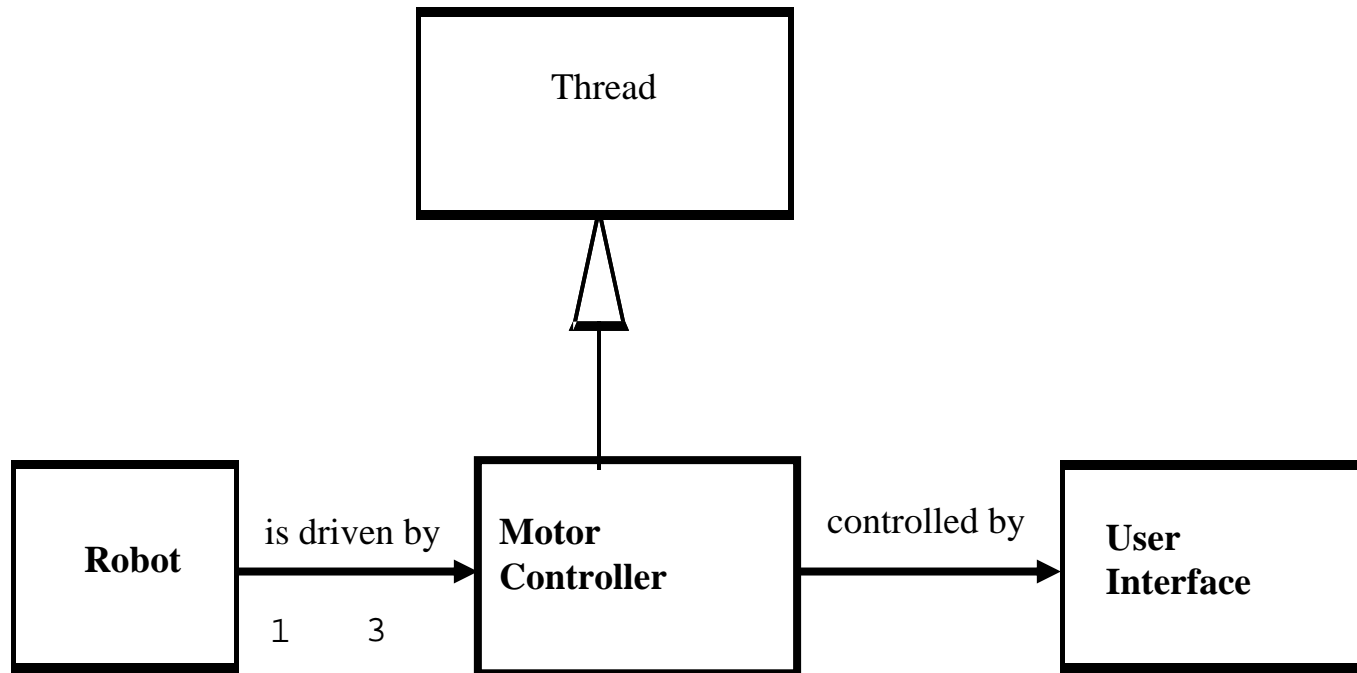
- Java does not support multiple inheritance, instead uses interfaces
- Multithreading for an already derived class
 - Implement interface `Runnable` (java.lang)
 - New class objects "are" Runnable objects
 - Override run method
 - Controls thread, just as deriving from `Thread` class
 - In fact, class `Thread` implements interface `Runnable`
- Create new threads using `Thread` constructors
 - `Thread(runnableObject)`
 - `Thread(runnableObject, threadName)`

Thread Creation

Either:

1. Extend `Thread` class and override the `run` method, or
2. Create an object which implements the `Runnable` interface and pass it to a `Thread` object via the `Thread` constructor
3. `Runnable` is useful when you want to subclass another `Object`

Thread Creation: Robot Example



Classes for Robot

```
public class UserInterface {  
    // Allows the next position of the robot  
    // to be obtained from the operator.  
    public int newSetting (int dim) { ... }  
    ...  
}  
  
public class Robot {  
    // The interface to the Robot itself.  
    public void move(int dim, int pos) { ... }  
    // Other methods, not significant here.  
}
```

Note in Java 1.5, dimension would be an enumeration type


Motor Controller extends Thread I

```
public class MotorController extends Thread
{
    public MotorController(int dimension,
        UserInterface UI, Robot robo) {
        // constructor
        super();
        dim = dimension;
        myInterface = UI;
        myRobot = robo;
    }
}
```

Motor Controller extends Thread II

```
public void run() {
    int position = 0; // initial position
    int setting;
    while(true) {
        // move to position
        myRobot.move(dim, position);
        // get new offset and update position
        setting = myInterface.newSetting(dim);
        position = position + setting;
    }
}

private int dim;
private UserInterface myInterface;
private Robot myRobot;
}
```



run
method
overridden

Motor Controller extends Thread III

```
final int xPlane = 0;  
final int yPlane = 1;  
final int zPlane = 2;
```

```
UserInterface UI = new UserInterface();  
Robot robo= new Robot();
```

```
MotorController MC1 = new MotorController(  
    xPlane, UI, robo);
```

```
MotorController MC2 = new MotorController(  
    yPlane, UI, robo);
```

```
MotorController MC3 = new MotorController(  
    zPlane, UI, robo);
```



threads
created

Motor Controller extends Thread IV

```
MC1.start();  
MC2.start();  
MC3.start();
```

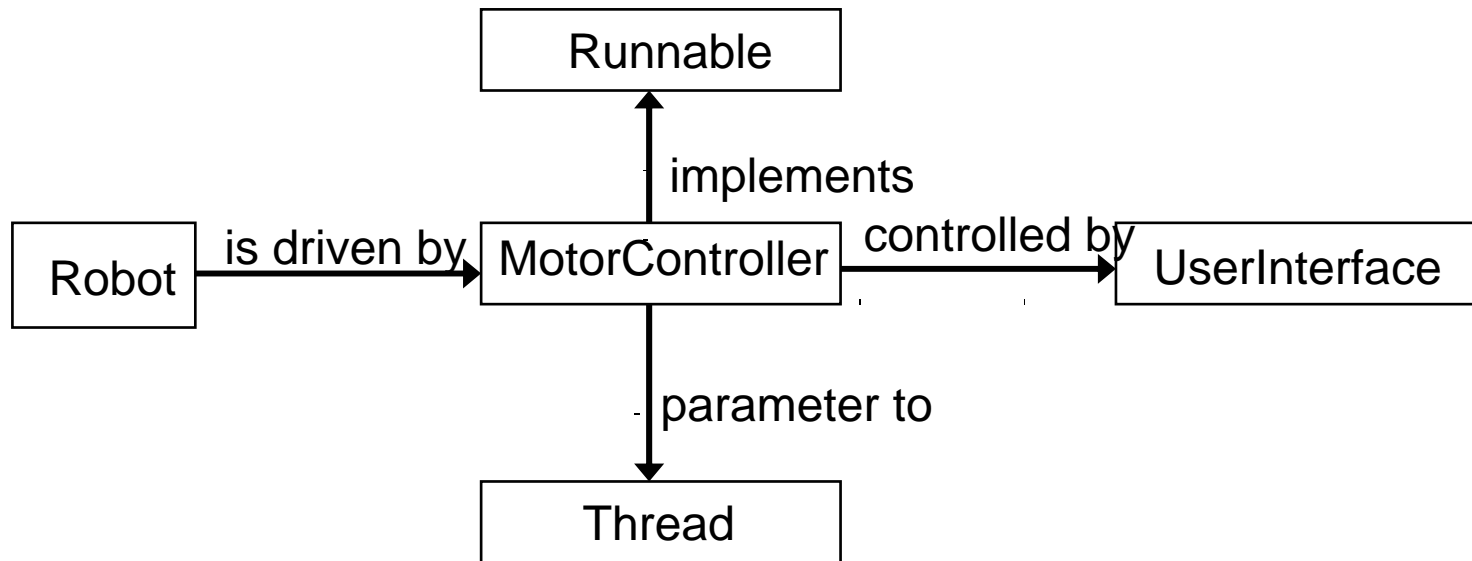
- When a thread is started, its `run` method is called and the thread is now executable
- When the `run` method exits, the thread is no longer executable and it can be considered terminated (Java calls this the `dead` state)
- The thread remains in this state until it is garbage collected
- In this example, the threads do not terminate

Warning

The run method should not be called directly by the application.
The system calls it.

If the run method is called explicitly by the application then the
code is executed sequentially not concurrently

Motor Controller implements Runnable I



Motor Controller implements Runnable II

```
public class MotorController implements Runnable
{
    public MotorController(int Dimension,
        UserInterface UI, Robot robo) {
        // No call to super() needed now,
        // otherwise constructor is the same.
    }

    public void run() {
        // Run method identical.
    }
    // Private part as before.
}
```


Motor Controller implements Runnable III

```
final int xPlane = 0;  
final int yPlane = 1;  
final int zPlane = 2;
```

```
UserInterface UI = new UserInterface();  
Robot robo= new Robot();
```

```
MotorController MC1 = new MotorController(  
    xPlane, UI, robo);  
MotorController MC2 = new MotorController(  
    yPlane, UI, robo);  
MotorController MC3 = new MotorController(  
    zPlane, UI, robo);
```

No
threads
created
yet

Motor Controller implements Runnable IV

```
Thread X = new Thread(MC1);  
Thread Y = new Thread(MC2);  
Thread Z = new Thread(MC2);
```

```
X.start();  
Y.start();  
Z.start();
```

threads started

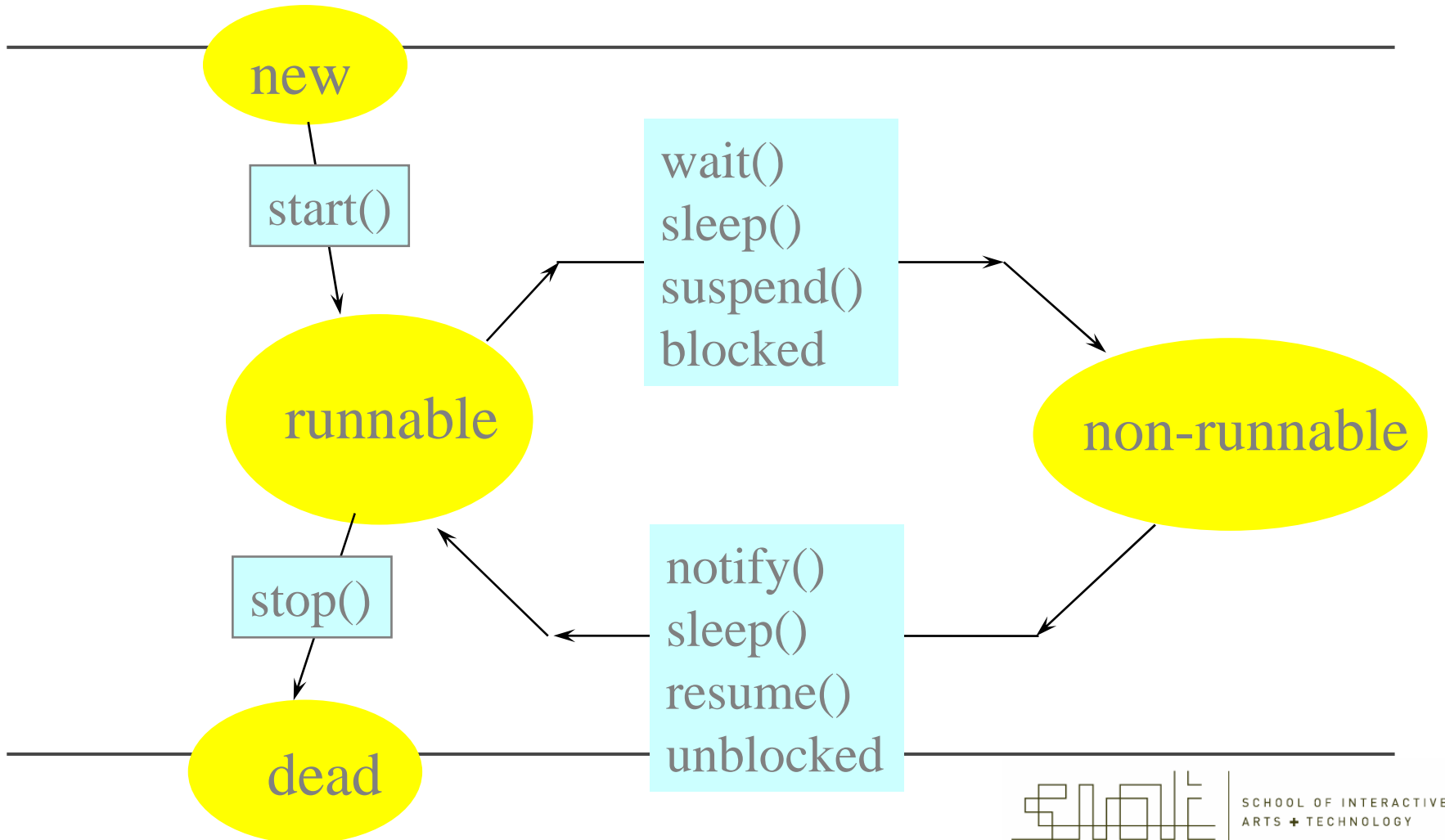
constructors
passed an object
implementing the
Runnable
interface when the
threads are
created

Note: it is also possible to recommend to the JVM the size of the stack to be used with the thread. However, implementations are allowed to ignore this recommendation.

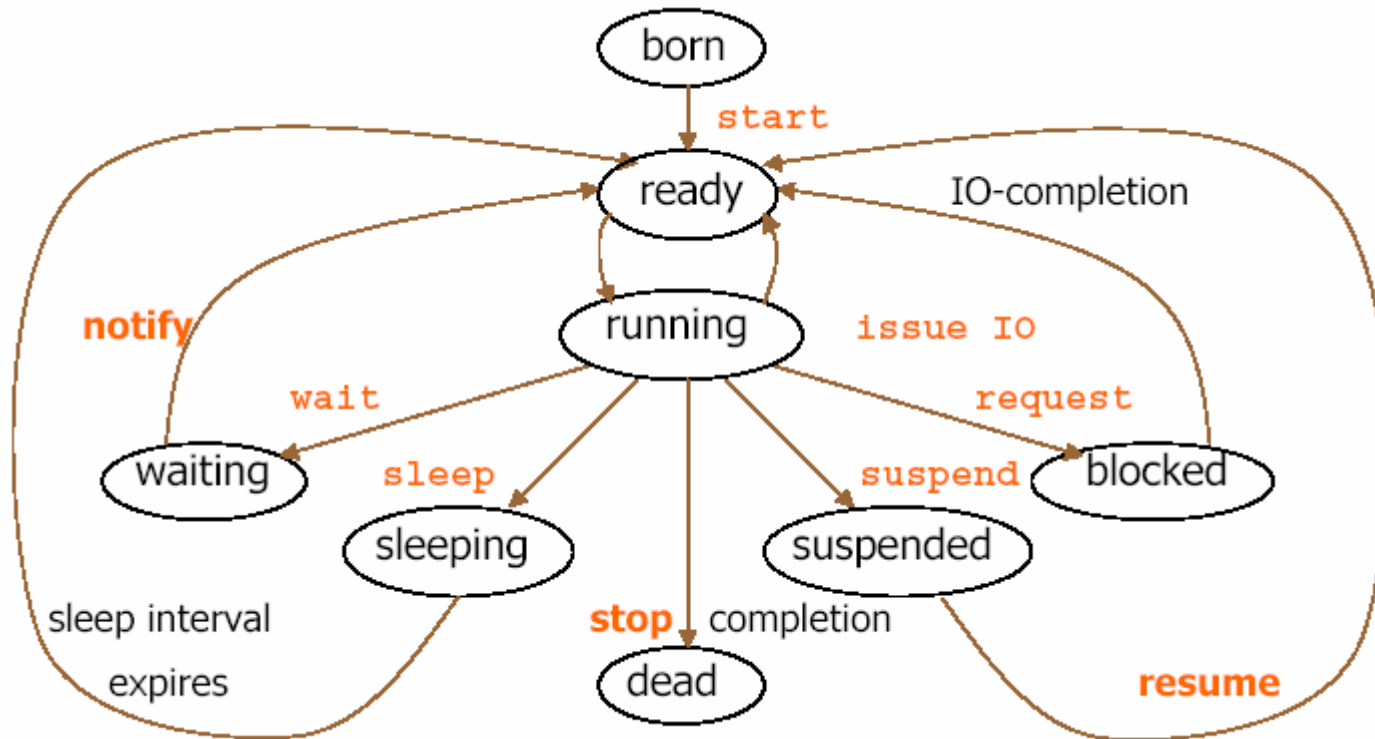
A Thread Terminates:

- when it completes execution of its `run` method either normally or as the result of an unhandled exception
- via a call to its `stop` method — the `run` method is stopped and the thread class cleans up before terminating the thread (releases locks and executes any finally clauses)
 - the thread object is now eligible for garbage collection.
 - `stop` is inherently unsafe as it releases locks on objects and can leave those objects in inconsistent states; the method is now deprecated and should not be used
- by its `destroy` method being called — `destroy` terminates the thread without any cleanup (not provided by many JVMs, now deprecated)

Life Cycle of Thread



Life Cycle of a Thread



So you think you don't use threads

- GUIs such as Swing already rely on threads
- Consider drawing and interaction
- The JPanel object allows drawing
- The typical usage is as follows
 - Create a class that extends JPanel
 - Override the `paintComponent()` method to draw whatever you want
 - Create an instance of your class
 - Add that instance to a JFrame
- **However, Swing is NOT designed to be threadsafe**
- Let's look at an example:
 - A black panel with a red disk in it, with the background flashing blue every second

JPanel Example

- MyPanel class extends JPanel

```
Terminal — vim — 76x24
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class MyPanel extends JPanel {

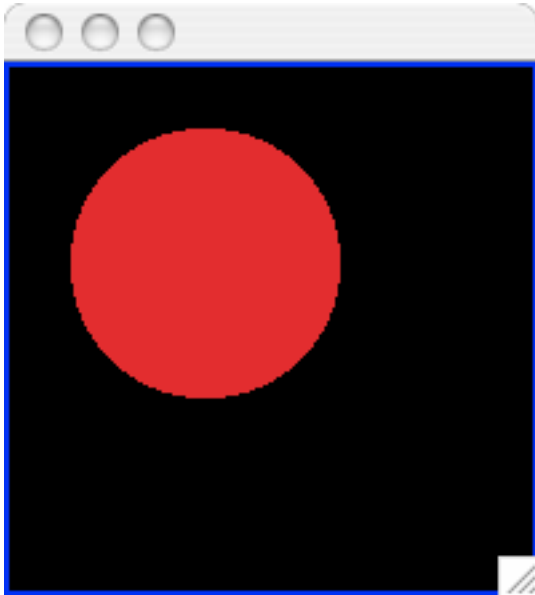
    static final int PANEL_WIDTH=200;
    static final int PANEL_HEIGHT=200;

    public MyPanel() {
        super();
        this.setPreferredSize(new Dimension(PANEL_WIDTH,PANEL_HEIGHT));
        this.setBorder(BorderFactory.createLineBorder(Color.blue,2));
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.setColor(Color.red);
        g.fillOval(PANEL_WIDTH/8,PANEL_HEIGHT/8,PANEL_WIDTH/2,PANEL_HEIGHT/2);
    }
}
}
```

JPanel Example

- What it looks like



- Note that the point (0,0) is in the top-left corner of the screen

JPanel Example

```
Terminal — vim — 67x27
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class DrawingExample {

    public static void main(String args[]) {
        DrawingExample d = new DrawingExample();
    }

    public DrawingExample() {
        JFrame frame = new JFrame();
        JPanel panel = new JPanel();

        frame.add(panel);
        frame.pack();
        frame.setVisible(true);

        while (true) {
            panel.setBackground(Color.blue);
            panel.setBackground(Color.black);
            try {Thread.sleep(1000);} catch (InterruptedException e) {}
        }
    }
}
```

ACTIVE
OGY

How does it work in the JVM?

- The JVM has a thread called the **event dispatch thread**, which does two things
 - Catches and dispatches swing events
 - e.g., mouse click goes to the right button
 - Execute “paint” operations of swing components
 - e.g., redraw something
- As we said before, swing was not designed to be thread safe
- **Swing’s single-thread rule**: the state of “realized” swing components can only be modified by the event dispatch thread
 - A component is realized when:
 - paint(), setVisible(true), or pack() has been called
 - It is part of a realized component
 - Essentially, a component is realized once it is visible/usable

Safe Swing code

```
public class Stuff implements ActionListener {
    public JFrame frame;
    public JButton button;
    public Stuff() {
        button = new Button();
        frame = new JFrame();
        frame.add(button);
        frame.pack();
        frame.setVisible(true);
        ...
    }
    public void someMethod() {
        ...
    }
    public void actionPerformed(ActionEvent event) {
        ...
    }
}
```

Safe Swing code

```
public class Stuff implements ActionListener {  
-   public JFrame frame;  
   public JButton button;  
   public Stuff() {  
       button = new Button();  
  
       ...  
       frame = new JFrame();  
       frame.add(button);  
  
       ...  
       frame.setVisible(true);  
       ...  
   }  
   public void someMethod() {  
       ...  
   }  
   public void actionPerformed(ActionEvent event) {  
       ...  
   }  
}
```

...
frame = new JFrame();
frame.add(button);
...
frame.setVisible(true);
...

**Unsafe to update
state of Swing
components**

**Safe to update
state of Swing
components**



Unsafe Swing Code

- Let's look at what happens when one doesn't write safe code
- We're going to use a contrived example that causes a problem almost instantly
 - In a real-world situation the problem may happen very rarely, which makes it extremely difficult to observe it and therefore to fix it!
- We implement a JFrame that has a scrollable list in which a thread rapidly adds/remove elements
 - Yes, it's useless
 - But it has invalid code in the “red boxes” in the previous slide

Example:

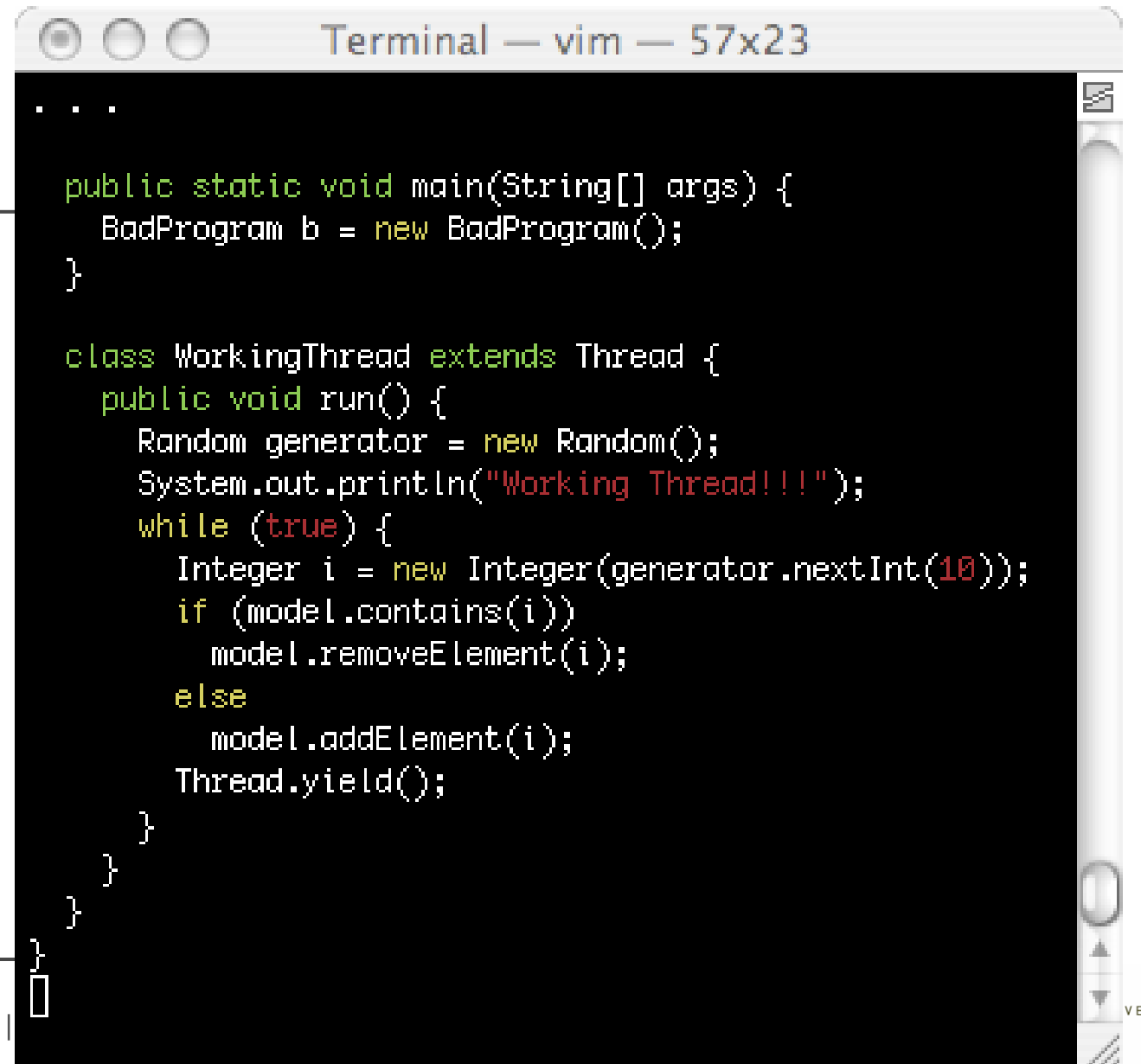
```
Terminal — vim — 63x31
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class BadProgram {

    JFrame frame;
    DefaultListModel model;

    public BadProgram() {
        model = new DefaultListModel();
        frame = new JFrame();
        JList list = new JList(model);
        JScrollPane scrollpane = new JScrollPane(list);
        JPanel p = new JPanel();
        p.add(scrollpane);
        frame.getContentPane().add(p, "Center");
        JButton b = new JButton("Fill List");
        p = new JPanel();
        p.add(b);
        frame.getContentPane().add(p, "North");
        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                WorkingThread t = new WorkingThread(); t.start();
            }
        });
        frame.pack();
        frame.setVisible(true);
    }
}
```

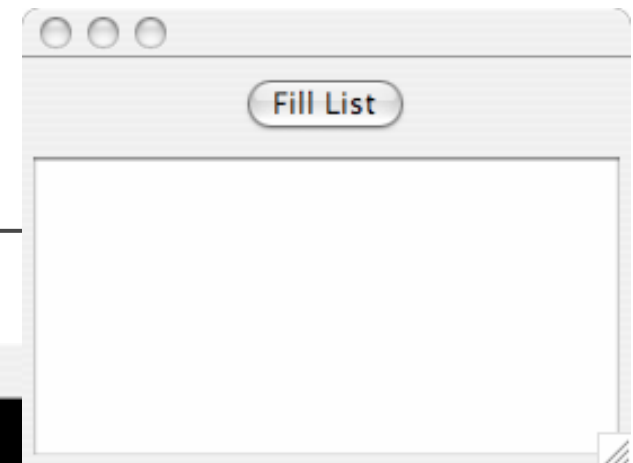
Example:



The image shows a terminal window titled "Terminal — vim — 57x23". The code is as follows:

```
...  
  
public static void main(String[] args) {  
    BadProgram b = new BadProgram();  
}  
  
class WorkingThread extends Thread {  
    public void run() {  
        Random generator = new Random();  
        System.out.println("Working Thread!!!");  
        while (true) {  
            Integer i = new Integer(generator.nextInt(10));  
            if (model.contains(i))  
                model.removeElement(i);  
            else  
                model.addElement(i);  
            Thread.yield();  
        }  
    }  
}
```

What happens?



```
Terminal — vim — 110x24
~/Java/Listing1% java BadProgram
Working Thread!!!
Exception in thread "AWT-EventQueue-0" java.lang.ArrayIndexOutOfBoundsException: 3 >= 3
    at java.util.Vector.elementAt(Vector.java:432)
    at javax.swing.DefaultListModel.elementAt(DefaultListModel.java:70)
    at javax.swing.plaf.basic.BasicListUI.updateLayoutState(BasicListUI.java:1154)
    at javax.swing.plaf.basic.BasicListUI.maybeUpdateLayoutState(BasicListUI.java:1105)
    at javax.swing.plaf.basic.BasicListUI.paint(BasicListUI.java:237)
    at javax.swing.plaf.ComponentUI.update(ComponentUI.java:154)
    at javax.swing.JComponent.paintComponent(JComponent.java:742)
    at javax.swing.JComponent.paint(JComponent.java:1005)
    at javax.swing.JComponent._paintImmediately(JComponent.java:4881)
    at javax.swing.JComponent.paintImmediately(JComponent.java:4667)
    at javax.swing.RepaintManager.paintDirtyRegions(RepaintManager.java:477)
    at javax.swing.SystemEventQueueUtilities$ComponentWorkRequest.run(SystemEventQueueUtilities.java:114)
    at java.awt.event.InvocationEvent.dispatch(InvocationEvent.java:209)
    at java.awt.EventQueue.dispatchEvent(EventQueue.java:461)
    at java.awt.EventDispatchThread.pumpOneEventForHierarchy(EventDispatchThread.java:269)
    at java.awt.EventDispatchThread.pumpEventsForHierarchy(EventDispatchThread.java:190)
    at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:184)
    at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:176)
    at java.awt.EventDispatchThread.run(EventDispatchThread.java:110)
```


Lesson

- What we saw is a typical multi-threading bug:
 - The code looks fine
 - We're not even using index into arrays but just use some built in class to add/remove elements to some data structure via its methods
 - And yet, we saw some "out of bound" exception
 - Therefore:
 - The data structure's methods are not thread-safe
 - More than one thread calls them
 - The two threads are:
 - Our own WorkerThread
 - The JVM's event dispatcher thread, which we didn't even create!
 - We wrote code that modified the state of a swing component inside one of the "red boxes"

How do we fix it?

- **Solution:** ensure that **only** the event dispatch thread calls the `removeElement()` and the `addElement()` methods, as these methods are not thread-safe in the Swing package
- But how can we do this?
 - The code to remove/add elements cannot go into any of the green boxes of the “Safe Swing code” slide
- Luckily, there is a mechanism to do exactly this
- The `InvokeLater()` method: forces some code to be executed by the event-dispatcher thread
 - The Swing designers put it there exactly so that we can do what we wanted to do in our example

InvokeLater ()

- The class `SwingUtilities` contains a static method called `invokeLater()`
- `invokeLater()` takes a `Runnable` object as parameter
- That `Runnable` object's `run` method should contain the code that should be executed by the event-dispatch thread
- The execution request is put on the event-dispatch thread's "queue", and the thread will get to it later (soon)
 - One more reason why nothing requiring a long execution time should ever be executed by the event-dispatch thread
 - Never put much work in the `actionPerformed()` method
 - Never give much work to do via `invokeLater()`
- Example (using a compact syntax):

```
invokeLater(new Runnable() {  
    void run() {...}  
});
```

How can we fix our code?

- Create two Runnable classes

```
Terminal — vim — 55x13
class ElementRemover implements Runnable {
    int index;
    DefaultListModel model;
    public ElementRemover(DefaultListModel m, int i) {
        index = i;
        model = m;
    }
    public void run() {
        model.removeElement(index);
    }
}

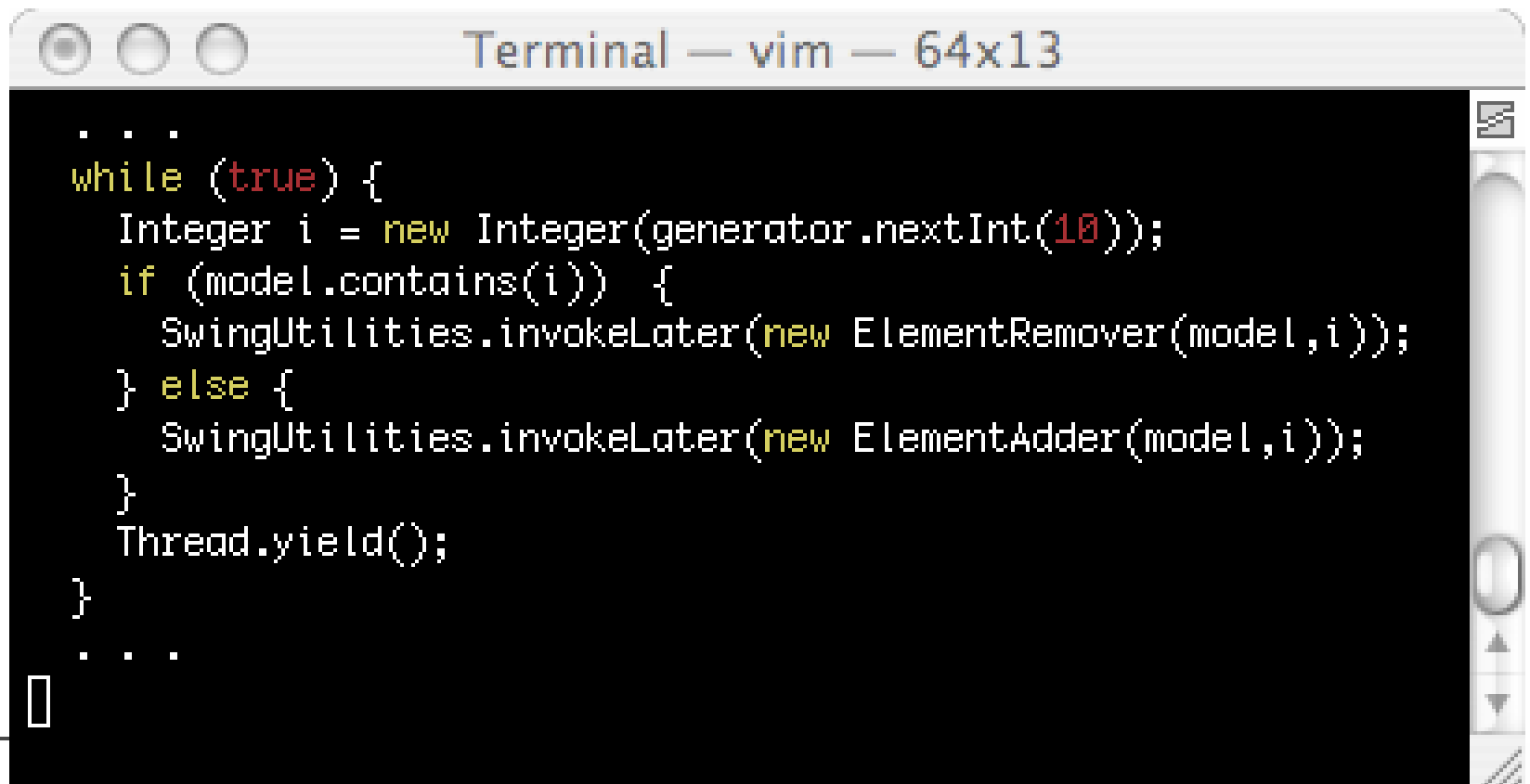
```

```
Terminal — vim — 55x13
class ElementAdder implements Runnable {
    int index;
    DefaultListModel model;
    public ElementAdder(DefaultListModel m, int i) {
        index = i;
        model = m;
    }
    public void run() {
        model.addElement(index);
    }
}

```

How can we fix our code?

- Call `invokeLater()`



```
Terminal — vim — 64x13  
.  
.  
.  
while (true) {  
    Integer i = new Integer(generator.nextInt(10));  
    if (model.contains(i)) {  
        SwingUtilities.invokeLater(new ElementRemover(model, i));  
    } else {  
        SwingUtilities.invokeLater(new ElementAdder(model, i));  
    }  
    Thread.yield();  
}  
.  
.  
.  
█
```



Other unsafe code

- Let's look at our flashing panel program again
- Turns out there are two big problems with it

```
Terminal — vim — 67x27
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class DrawingExample {

    public static void main(String args[]) {
        DrawingExample d = new DrawingExample();
    }

    public DrawingExample() {
        JFrame frame = new JFrame();
        MyPanel panel = new MyPanel();

        frame.add(panel);
        frame.pack();
        frame.setVisible(true);

        while (true) {
            panel.setBackground(Color.blue);
            panel.setBackground(Color.black);
            try {Thread.sleep(1000);} catch (InterruptedException e) {}
        }
    }
}
```

Other unsafe code

- **Problem #1: the event-dispatch thread may “miss” the setBackground calls**
 - Unlikely with the `sleep(1000)` in this program
 - But what if we had many threads and a shorter period?
- **Problem #2: the calls to `setBackground()` are not thread-safe!**
 - May cause a problem with the event-dispatcher thread
 - Very unlikely, but if it ever happens it will be very hard to debug
 - Nothing worse than a user saying “your code fails once a month for no known reason”

```
Terminal — vim — 67x27
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class DrawingExample {

    public static void main(String args[]) {
        DrawingExample d = new DrawingExample();
    }

    public DrawingExample() {
        JFrame frame = new JFrame();
        MyPanel panel = new MyPanel();

        frame.add(panel);
        frame.pack();
        frame.setVisible(true);

        while (true) {
            panel.setBackground(Color.blue);
            panel.setBackground(Color.black);
            try {Thread.sleep(1000);} catch (InterruptedException e) {}
        }
    }
}
```

Fixed Flashing Panel

```
Terminal — vim — 49x16

class backgroundSetter implements Runnable {

    JPanel panel;
    Color color;

    public backgroundSetter(JPanel p, Color c) {
        panel = p;
        color = c;
    }

    void run() {
        p.setBackground(c);
    }
}
```

```
Terminal — vim — 68x6

. . .
SwingUtilities.invokeLater(new backgroundSetter(panel,Color.blue));
SwingUtilities.invokeLater(new backgroundSetter(panel,Color.red));
. . .

```


Even Pickier Thread-Safety

- There is a third problem with our flashing panel program!
- After the call to `pack()`, the `JFrame` object is *realized*
- Therefore, the call to `setVisible()` is in a “red box” and is not thread-safe!
- It is very unlikely that there could be a problem
- But removing all doubts is easy

```
Terminal — vim — 67x27
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class DrawingExample {

    public static void main(String args[]) {
        DrawingExample d = new DrawingExample();
    }

    public DrawingExample() {
        JFrame frame = new JFrame();
        MyPanel panel = new MyPanel();

        frame.add(panel);
        frame.pack();
        frame.setVisible(true);

        while (true) {
            panel.setBackground(Color.blue);
            panel.setBackground(Color.black);
            try {Thread.sleep(1000);} catch (InterruptedException e) {}
        }
    }
}
```

Absolutely safe code



```
...l — vim — 42x13
class JFrameShower implements Runnable {
    JFrame frame;

    public JFrameShower(JFrame f) {
        frame = f;
    }

    void run() {
        f.setVisible(true);
    }
}

Terminal — vim — 54x6
. . .
frame.pack();
SwingUtilities.invokeLater(new JFrameShower(frame));
. . .
```

invokeAndWait ()

- There is also an `invokeAndWait()` method
 - Returns only once the event-dispatcher thread has executed the code
- Some of the Swing methods are thread-safe
 - As stated explicitly in the documentation
- For these, one doesn't have to call `invokeLater()` or `invokeAndWait()`
- Examples:
 - `repaint()`
 - `revalidate()`, `invalidate()`
 - Use to facilitate adding components to an existing/displayed GUI

Background Threads

- In many cases, one really has the following simple system
 - A **main thread** for the application execution
 - The **Event Dispatch (ED)** thread that deals with all Swing events
- But what if clicking on some buttons can launch a time consuming computation?
- Clearly you don't want the ED to do it, to avoid the frozen syndrome
- Therefore, you use worker threads, often called **background threads**
- When the button is clicked, for instance, a new thread is spawned to do the computation
- This could be done in the `actionPerformed()` method of the `ActionListener` of the button.

Worker Threads in a GUI

- In many GUI applications worker threads don't really need to talk to each other
 - They are independent threads
 - Perhaps they can terminate each other, but that's it
- At this point, we have all the tools to do all the above
 - We can create threads, call `invokeLater()`, etc.
 - We'll see how to terminate threads later in the course
- This is something countless developers do
 - Create a GUI
 - Create worker threads for everything under the sun
- This requires quite a bit of code to be rewritten
- Turns out, there is a very convenient class called `SwingWorker`, recently added to Java SE 6
- I have included some notes but will not talk about it here
 - I'll omit some details as there are great `SwingWorker` tutorials on-line
 - You can do everything in this course without it, but you may find it used in the real world

Summary

- It is important to decide when to use multithreading and when to avoid it.
- The main drawbacks to multithreading are:
 - **Slowdown** while waiting for shared resources
 - **Additional CPU overhead** required to manage threads
 - **Unrewarded complexity**, such as having a separate thread to update each element of an array
- An additional advantage to threads is that they are “**light**” execution context switches
- Care must be taken in resource sharing and (to a lesser extent) scheduling
- You’re dealing with threads even when you don’t know it (GUIs)

Summary II

- Multi-threading issues aplenty in programs that don't even have multiple threads!
- One has to be very **thorough** and saying “my code is thread-safe” is a big claim
- Something that may “always” work could break one day, especially on a different system
 - Some developers don't even know that calling `setVisible()` after `pack()` is technically a thread-safety problem
 - but it's considered dangerous by some
 - Some developers would never do it
 - considered completely paranoid by some
 - but if you've been bitten once by a “horrible multi-threading bug that took you 1 month to figure out” ...
- For simple background tasks, `SwingWorker` is useful
- You **MUST** understand what code runs in the ED thread
 - `SwingUtilities.isEventDispatchThread()` returns a boolean
- Check out the on-line material and documentations

Why SwingWorker?

- If the only reason you have threads in your GUI is so that it is responsive, and if these threads do not need to talk to each other at all, then SwingWorker is convenient
 - It removes the need to create threads explicitly
 - You never have Thread/Runnable objects visible to you
- SwingWorker is an **generic abstract** class that must be extended
- It has important methods
 - `doInBackground()`: do the work without freezing everything
 - `done()`: invoked when the method finishes
 - `publish()`: used to provide intermediate results
 - `process()`: used to deal with published intermediate results
- Only the first one must absolutely be overridden
- It also makes it possible to get some object returned by `doInBackground()`
 - which is why the class is generic

SwingWorker Example

- Say you want to have a GUI in which you have a “Load” button
- When clicked, the button counts from 1 to 100, sleeping .10 seconds in between numbers
- While counting some progress bar on the GUI is updated
- When done, we want to get back a String that says “done”
- Yes, this is USELESS, but it’s simple and quick, and if you know how to do it, you can do useful things
- Let’s look at this in details

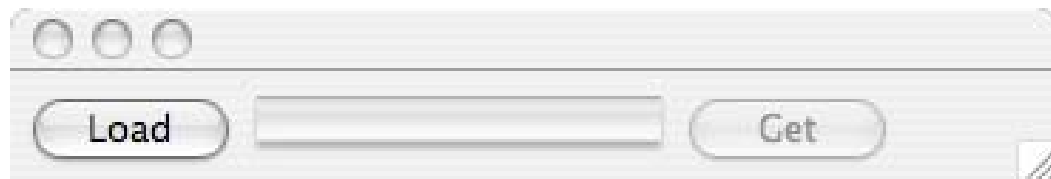
```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import org.jdesktop.swingworker.*;
import java.util.concurrent.*;

public class SwingWorkerExample extends JFrame
    implements ActionListener {

    private JButton start_button;
    private JButton get_button;
    private JProgressBar bar;
    private JLabel label;

    private MySwingWorker worker;

    public SwingWorkerExample() {
        this.setLayout(new FlowLayout());
        start_button = new JButton("Load");
        start_button.addActionListener(this);
        bar = new JProgressBar(0,100);
        get_button = new JButton("Get");
        get_button.addActionListener(this);
        get_button.setEnabled(false);
        label = new JLabel("          ");
        this.add(start_button);
        this.add(bar);
        this.add(get_button);
        this.add(label);
    }
    . . .
}
```



```
Terminal — vim — 55x16  
.  
.  
.  
private static void createAndShowGUI() {  
    SwingWorkerExample swe = new SwingWorkerExample();  
    swe.pack();  
    swe.setVisible(true);  
}  
public static void main(String[] args) {  
    SwingUtilities.invokeLater(new Runnable() {  
        public void run() {  
            createAndShowGUI();  
        }  
    });  
}  
.  
.  
.  
DL OF INTERACTIVE  
+ TECHNOLOGY
```

```
public void actionPerformed(ActionEvent event) {  
    Component c = (Component)event.getSource();  
    if (c == start_button) {  
        start_button.setEnabled(false);  
        get_button.setEnabled(false);  
        worker = new MySwingWorker(start_button,  
                                    bar, get_button);  
        worker.execute();  
    }  
    if (c == get_button) {  
        try { label.setText(worker.get()); }  
        catch (InterruptedException e) {}  
        catch (ExecutionException e) {}  
        get_button.setEnabled(false);  
    }  
}
```

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import org.jdesktop.swingworker.*;

public class MySwingWorker extends SwingWorker<String,Void> {
    private JButton start_button;
    private JProgressBar bar;
    private JButton get_button;

    public MySwingWorker(JButton x, JProgressBar y, JButton z) {
        super();
        start_button = x; bar = y; get_button = z;
    }

    // Executed by the ED thread
    public void done() {
        start_button.setEnabled(true);
        get_button.setEnabled(true);
        bar.setValue(0);
    }
}
```

```
..  
..  
// Executed by a background thread  
public String doInBackground() {  
    for (int i=0; i<100; i++) {  
        SwingUtilities.invokeLater(new ProgressUpdater(bar,i));  
        try { Thread.sleep(50); }  
        catch (InterruptedException e) {}  
    }  
    return new String("done");  
}  
  
private class ProgressUpdater implements Runnable {  
    private JProgressBar bar;  
    private int value;  
  
    public ProgressUpdater(JProgressBar x, int y) {  
        bar = x; value = y;  
    }  
    public void run() { bar.setValue(value); }  
}
```

SwingWorker Conclusion

- SwingWorker provides a higher abstraction above Threads to avoid the hassle of multi-threaded programming in some situations
- There is actually a cooler way to update a program bar using “properties” (see on-line material if curious)
- There are several such higher levels of abstractions (will see something called ThreadPool for instance)
- But in the end, if you don’t know how things work underneath it’s difficult to write good concurrent code
- For assignments you can use SwingWorker if you feel like it
 - There is a link to the .jar file and some documentation on the course’s Web page



Thread language hierarchy

