Java Threads

- Sometimes you need **multiple subtasks**.
- Multiprocessors.
- Multithreads can be on one processor, but you don't need to know.

A Thread

- Each thread is like a program on its own.
- CPU(s) are divided between multiple threads.
- Threads are used to make:
  - Responsive user interface
  - Animation (now flash and gif animation are better)
  - Networking i.e. Server ...

Non-Responsive user interface

```
// A non-responsive user interface.
// <applet code=Counter1 width=300 height=100>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Counter1 extends JApplet {
  private int count = 0;
  private JButton
    start = new JButton("Start"),
    onOff = new JButton("Toggle");
  private JTextField t = new JTextField(10);
  private boolean runFlag = true;

  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    start.addActionListener(new StartL());
    cp.add(start);
    onOff.addActionListener(new OnOffL());
    cp.add(onOff);
  }
```

Below we show a method that should have been in a separate thread.

```
  public void go() {
    while (true) {
      try {
        Thread.sleep(100);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
      if (runFlag)
        t.setText(Integer.toString(count++));
    }
  }

  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      go();
    }
  }
  class OnOffL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      runFlag = !runFlag;
    }
  }
  public static void main(String[] args) {
    Console.run(new Counter1(), 300, 100);
  }
}
```

What is the problem?

- **go()** never returns, so no more button press can be handled.
- Need to use a new thread to do the work of **go()**

Thread Creation: (1st method) Inheriting from Thread class

- A thread is an object
- Override **run()**
- **run()** is usually a loop
- So what we do is:
  - Create Thread object
  - Call **start()**, which spawn a new thread and that thread executes **run()**

```
Thread 1(5)
Thread 1(4)
Thread 2(5)
Thread 5(5)
Thread 2(4)
```

Figure 1: Scheduling by new compiler

Simple Program with thread

```
public class SimpleThread extends Thread{
    private int countDown = 5;
    private static int threadCount =0;
    private int threadNumber = ++threadCount;
    public SimpleThread(){
        System.out.println("Making "+ threadNumber);
    }

    public void run(){
        while(true) {
          System.out.println("Thread "+ threadNumber
         + "(" + countDown + ")");
         if (--countDown == 0) return;
        }
    }

    public static void main(String[ ]  args){
     for (int i =0; i<5; i++)
        new SimpleThread().start();
     System.out.println("All threads started ");
    }
}
```

The result is in figure 1

Threads execution are not ordered. But in some compiler, such as my Eclipse 2.1 at home, threads execution can appear like they are in order (figure 2).

A bit better time-slice, with yield()

To prevent a thread from executing too long, a thread can give another thread a chance. We can add a yield() method to the thread code of SimpleThread.

```
public void run() {
    while(true) {
        System.out.println("Thread "+ threadNumber
```

```
Thread 1(5)
Thread 1(4)
Thread 1(3)
Thread 1(2)
Thread 1(1)
Thread 2(5)
Thread 2(4)
Thread 2(3)
```

Figure 2: Scheduling by older compiler

```
Thread 1(5)
Thread 2(5)
Thread 4(5)
Thread 5(5)
Thread 3(5)
Thread 1(4)
Thread 3(4)
Thread 2(4)
```

Figure 3: Scheduling with yield()

```
            + "(" + countDown + ")");
            if (--countDown == 0) return;
            yield();
        }
    }
```

Each thread will have more equal chance of running. One result obtained using yield is as figure 3.

Another basic - Sleep

If you want your thread to sleep for at least 100 milliseconds (may not be exact time, depending on the scheduler), you can use the code below in your run method.

```
try {
    sleep(100);
} catch (InterruptedException e) {
    throw new RuntimeException(e);
}
```

The sleep method must be placed inside a try block because it is possible for sleep( ) to be interrupted before it times out. This happens if someone else has a reference to the thread and they call interrupt( ) on the thread.

Responsive user interface

```
public class Counter2 extends JApplet {
  private class SeparateSubTask extends Thread {
    private int count = 0;
    private boolean runFlag = true;
    SeparateSubTask() { start(); }
    void invertFlag() { runFlag = !runFlag; }
    public void run() {
      while (true) {
       try {
        sleep(100);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
       if(runFlag)
         t.setText(Integer.toString(count++));
      }
    }
  }

  private SeparateSubTask sp = null;
  private JTextField t = new JTextField(10);
  private JButton
    start = new JButton("Start"),
    onOff = new JButton("Toggle");

  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(sp == null)
        sp = new SeparateSubTask();
    }
  }

  class OnOffL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(sp != null)
        sp.invertFlag();
    }
  }

public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    start.addActionListener(new StartL());
```

```
    cp.add(start);
    onOff.addActionListener(new OnOffL());
    cp.add(onOff);
  }

public static void main(String[] args) {
    Console.run(new Counter2 (), 300, 100);
  }
}
```

Creating Many Threads

```
// <applet code=Counter4 width=200 height=600>
// <param name=size value="12"></applet>
import ......

public class Counter4 extends JApplet {
  private JButton start = new JButton("Start");
  private boolean started = false;
  private Ticker[] s;
  private boolean isApplet = true;
  private int size = 12;
  class Ticker extends Thread {
    private JButton b = new JButton("Toggle");
    private JTextField t = new JTextField(10);
    private int count = 0;
    private boolean runFlag = true;
    public Ticker() {
      b.addActionListener(new ToggleL());
      JPanel p = new JPanel();
      p.add(t);
      p.add(b);
      // Calls JApplet.getContentPane().add():
      getContentPane().add(p);
    }
    class ToggleL implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        runFlag = !runFlag;
      }
    }
    public void run() {
      while (true) {
        if (runFlag)
          t.setText(Integer.toString(count++));
        try {
          sleep(100);
```

```
        } catch(InterruptedException e) {
          System.err.println("Interrupted");
        }
      }
    }
  }

  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(!started) {
        started = true;
        for (int i = 0; i < s.length; i++)
          s[i].start();
      }
    }
  }

  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    // Get parameter "size" from Web page:
    if (isApplet) {
      String sz = getParameter("size");
      if(sz != null)
        size = Integer.parseInt(sz);
    }
    s = new Ticker[size];
    for (int i = 0; i < s.length; i++)
      s[i] = new Ticker();
    start.addActionListener(new StartL());
    cp.add(start);
  }
  public static void main(String[] args) {
    Counter4 applet = new Counter4();
    // This isn't an applet, so set the flag and
    // produce the parameter values from args:
    applet.isApplet = false;
    if(args.length != 0)
      applet.size = Integer.parseInt(args[0]);
    Console.run(applet, 200, applet.size * 50);
  }
}
```

Runnable

If you inherit from something else rather than thread, but you want to make it a thread also, use **Runnable**

1. Create a class that implements Runnable interface.
2. Write the run method of that class.
3. Creating the actual thread by createing a separate Thread object and give the runnable object as the thread constructor argument

```
new Thread(the runnable object);
```

and then call start() on it.

Example:

```
public class RunnableThread implements Runnable {
  private int countDown = 5;
  public String toString() {
    return "#" + Thread.currentThread().getName() +
      ": " + countDown;
  }
  public void run() {
    while(true) {
      System.out.println(this);
      if(--countDown == 0) return;
    }
  }
  public static void main(String[] args) {
    for(int i = 1; i <= 5; i++)
      new Thread(new RunnableThread(),""+i).start();
    // Output is like SimpleThread.java
  }
}
```

If you want to do anything else to the Thread object (such as getName( ) in toString( ) you must explicitly get a reference to it by calling

```
Thread.currentThread();
```

Example: : Tests threading efficiency... this means...

```
class CBox extends JPanel implements Runnable {
  private Thread t;
  private int pause;
  private static final Color[] colors = {
    Color.black, Color.blue, Color.cyan,
    Color.darkGray, Color.gray, Color.green,
    Color.lightGray, Color.magenta,
    Color.orange, Color.pink, Color.red,
    Color.white, Color.yellow
```

```
  };
  private Color cColor = newColor();
  private static final Color newColor() {
    return colors[
      (int)(Math.random() * colors.length)
    ];
  }
  - - - - - - - - - - - - - - - - - - -
public void paintComponent(Graphics  g) {
    super.paintComponent(g);
    g.setColor(cColor);
    Dimension s = getSize();
    g.fillRect(0, 0, s.width, s.height);
  }
  public CBox(int pause) {
    this.pause = pause;
    t = new Thread(this);
    t.start();
  }
  public void run() {
    while(true) {
      cColor = newColor();
      repaint();
      try {
        t.sleep(pause);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
  }
}
 - - - - - - - - - - - - - - - - - - -
public class ColorBoxes extends JApplet {
  private boolean isApplet = true;
  private int grid = 12;
  private int pause = 50;
  public void init() {
    // Get parameters from Web page:
    if (isApplet) {
      String gsize = getParameter("grid");
      if(gsize != null)
        grid = Integer.parseInt(gsize);
      String pse = getParameter("pause");
      if(pse != null)
        pause = Integer.parseInt(pse);
    }
```

```
    Container cp = getContentPane();
    cp.setLayout(new GridLayout(grid, grid));
    for (int i = 0; i < grid * grid; i++)
      cp.add(new CBox(pause));
  }
  public static void main(String[] args) {
    ColorBoxes applet = new ColorBoxes();
    applet.isApplet = false;
    if(args.length > 0)
      applet.grid = Integer.parseInt(args[0]);
    if(args.length > 1)
      applet.pause = Integer.parseInt(args[1]);
    Console.run(applet, 500, 400);
  }
}
```

Sharing Resources

- Single thread - you own everything, no problem
- Multi - threaded - more than one thread may try to use a resource at the same time:
  - deposit and withdraw from bank account
  - generally... grabbing the same variable
- Java has a lock (or monitor) for each object. First thread that acquires the lock gains control of the object, and the other threads can't call synchronized methods for that object

Improperly Accessing Resources: Example

```
public class Sharing1 extends JApplet {
  private static int accessCount = 0;
  private static JTextField aCount =
    new JTextField("0", 7);
  public static void incrementAccess() {
    accessCount++;
    aCount.setText(Integer.toString(accessCount));
  }
  private JButton
    start = new JButton("Start"),
    watcher = new JButton("Watch");
  private boolean isApplet = true;
  private int numCounters = 12;
  private int numWatchers = 15;
  private TwoCounter[] s;
```

```
- - - - - - - - - - - - - - - - - - -
class TwoCounter extends Thread {
    private boolean started = false;
    private JTextField
      t1 = new JTextField(5),
      t2 = new JTextField(5);
    private JLabel l =
      new JLabel("count1 == count2");
    private int count1 = 0, count2 = 0;
    // Add the display components as a panel:
    public TwoCounter() {
      JPanel p = new JPanel();
      p.add(t1);
      p.add(t2);
      p.add(l);
      getContentPane().add(p);
    }
    public void start() {
      if(!started) {
        started = true;
        super.start();
      }
    }

public void run() {
      while (true) {
        t1.setText(Integer.toString(count1++));
        t2.setText(Integer.toString(count2++));
        try {
          sleep(500);
        } catch(InterruptedException e) {
          System.err.println("Interrupted");
        }
      }
    }
    public void synchTest() {
      incrementAccess();
      if(count1 != count2)
        l.setText("Unsynched");
    }
  }

class Watcher extends Thread {
    public Watcher() { start(); }
    public void run() {
      while(true) {
```

```
        for(int i = 0; i < s.length; i++)
          s[i].synchTest();
        try {
          sleep(500);
        } catch(InterruptedException e) {
          System.err.println("Interrupted");
        }
      }
    }
  }
  class StartL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      for(int i = 0; i < s.length; i++)
        s[i].start();
    }
  }

class WatcherL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      for(int i = 0; i < numWatchers; i++)
        new Watcher();
    }
  }
  public void init() {
    if(isApplet) {
      String counters = getParameter("size");
      if(counters != null)
        numCounters = Integer.parseInt(counters);
      String watchers = getParameter("watchers");
      if(watchers != null)
        numWatchers = Integer.parseInt(watchers);
    }
    s = new TwoCounter[numCounters];
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    for(int i = 0; i < s.length; i++)
      s[i] = new TwoCounter();
    JPanel p = new JPanel();
    start.addActionListener(new StartL());
    p.add(start);
    watcher.addActionListener(new WatcherL());
    p.add(watcher);
    p.add(new JLabel("Access Count"));
    p.add(aCount);
    cp.add(p);
  }
```

```
  public static void main(String[] args) {
    Sharing1 applet = new Sharing1();
    // This isn't an applet, so set the flag and
    // produce the parameter values from args:
    applet.isApplet = false;
    applet.numCounters =
      (args.length == 0 ? 12 :
        Integer.parseInt(args[0]));
    applet.numWatchers =
      (args.length < 2 ? 15 :
        Integer.parseInt(args[1]));
    Console.run(applet, 350,
      applet.numCounters * 50);
  }
}
```

You need some way to prevent two threads from accessing the same resource, at least during critical periods. You need mutual exclusion.

Object Locks

- One lock per object (also referred to as a monitor)
- One lock per class too
- should consider making all shared data **private**
- If a method is **synchronized**, entering that method acquires the lock
- No other threads may call any **synchronized** method for that object until the lock is released
- If you synchronize only one of the methods, then the other is free to ignore the object lock and can be called, accessing the shared variable immediately.

Synchronizing the Counters

```
public synchronized void run() {
      while (true) {
        t1.setText(Integer.toString(count1++));
        t2.setText(Integer.toString(count2++));
        try {
          sleep(500);
        } catch(InterruptedException e) {
          System.err.println("Interrupted");
        }
      }
```

```
    }
    public synchronized void synchTest() {
      incrementAccess();
      if(count1 != count2)
        l.setText("Unsynched");
    }
  }
```

But this one doesn't work. Why?

Synchronized Clause

```
synchronized(Object){
//only one thread can access
//this part of code at one time
}
```

Allowing Synchtest to look

```
public void run() {
      while (true) {
        synchronized(this){
        t1.setText(Integer.toString(count1++));
        t2.setText(Integer.toString(count2++));
        }
        try {
 ....
```

- One thread may acquire an objects lock multiple times. This happens if one method calls a second method on the same object, which in turn calls another method on the same object, etc.
- The JVM keeps track of the number of times the object has been locked.
  - If the object is unlocked, it has a count of zero.
  - As a thread acquires the lock for the first time, the count goes to one.
  - Each time the thread acquires a lock on the same object, the count is incremented.
  - Naturally, multiple lock acquisition is only allowed for the thread that acquired the lock in the first place.
  - Each time the thread leaves a synchronized method, the count is decremented, until the count goes to zero, releasing the lock entirely for use by other threads.
  - Locking different objects means nothing. Synchronization lock only works if we synchronize on the same object.

  - Theres also a single lock per class (as part of the Class object for the class),
    * So that synchronized static methods can lock each other out from simultaneous access of static data.

Synchronization: speed problem?

- 6x method call overhead, min is 4x... is this really true?
- Older standard Java libraries used **synchronized** a lot, did not provide any alternatives

Thread States

1. New: created but not started yet
2. Runnable: thread can be run when CPU available
3. Dead: returns from **run()** or throw exception
4. Blocked: Scheduler will skip over it and not give it CPU time

Blocked Thread- because??

1. Sleeping: **sleep(milliseconds)**
2. Suspended: **suspend()**, it won't be runnable again until the thread gets the **resume()** message (deprecated in Java 2)
3. Waiting: **wait()**, it won't be runnable again until the thread gets the **notify()** or **notifyAll()** message
4. Waiting for I/O
5. The thread is trying to call a **synchronized** method on another object and that object's lock is not available

Deadlock

- chain of threads waiting for each other
- No language support for preventing it
- Java 2 deprecates **stop()**, **suspend()**, **resume()**, **destroy()** partly to prevent deadlock. **stop()** and **suspend()** do not release locks.

Dijkstra's dining philosophers

- Five philosophers (the example shown here allows any number).
- They sit at a table with a limited number of chopsticks.

- Two chopsticks are required to get spaghetti in the middle of the table.
- Only five chopsticks, spaced around the table between the philosophers.
- When a philosopher wants to eat, he or she must get the chopstick to the left and the one to the right.
- If the philosopher on either side is using the desired chopstick, then our philosopher must wait.
- Note that the reason this problem is interesting is because it demonstrates that a program can appear to run correctly but actually be deadlock prone.
- If you have lots of philosophers and/or they spend a lot of time thinking, you may never see the deadlock even though it remains a possibility.
- The obvious case for deadlock is when all philosophers pick a chopstick to their left at the same time. So they all have to wait for their right chopsticks.
- A circular wait has happen.
- The deadlock can be broken by swapping the initialization order in the constructor for the last philosopher, causing that last philosopher to actually get the right chopstick first, then the left.

Example: Deadlock philosophers

The command-line arguments allow you to adjust the number of philosophers and a factor to affect the amount of time each philosopher spends thinking.

```
import java.util.*;

class Chopstick {
  private static int counter = 0;
  private int number = counter++;
  public String toString() {
    return "Chopstick " + number;
  }
}

class Philosopher extends Thread {
  private static Random rand = new Random();
  private static int counter = 0;
  private int number = counter++;
  private Chopstick leftChopstick;
  private Chopstick rightChopstick;
  static int ponder = 0; // Package access
  public Philosopher(Chopstick left,Chopstick right){
    leftChopstick = left;
```

```
    rightChopstick = right;
    start();
  }
  public void think() {
    System.out.println(this + " thinking");
    if(ponder > 0)
      try {
        sleep(rand.nextInt(ponder));
      } catch(InterruptedException e) {
        throw new RuntimeException(e);
      }
  }
  public void eat() {
    synchronized(leftChopstick) {
      System.out.println(this + " has "
        + this.leftChopstick + " Waiting for "
        + this.rightChopstick);
      synchronized(rightChopstick) {
        System.out.println(this + " eating");
      }
    }
  }
  public String toString() {
    return "Philosopher " + number;
  }
  public void run() {
    while(true) {
      think();
      eat();
    }
  }
}

public class DiningPhilosophers {
  public static void main(String[] args) {
   if(args.length < 3) {
    System.err.println("usage:\n"+
    "java DiningPhilosophers numberOfPhilosophers"+
    "ponderFactor deadlock timeout\n"+
    "A nonzero ponderFactor will generate a random"+
    "sleep time during think().\n"+
    "If deadlock is not the string"+
    "'deadlock',the program will not deadlock.\n"+
    "A nonzero timeout will stop the program after"+
    "that number of seconds.");
    System.exit(1);
```

```
    }
     Philosopher[] philosopher =
       new Philosopher[Integer.parseInt(args[0])];
     Philosopher.ponder = Integer.parseInt(args[1]);
     Chopstick
       left = new Chopstick(),
       right = new Chopstick(),
       first = left;
     int i = 0;
     while(i < philosopher.length - 1) {
       philosopher[i++] =
         new Philosopher(left, right);
       left = right;
       right = new Chopstick();
     }
     if(args[2].equals("deadlock"))
       philosopher[i] =new Philosopher(left,first);
     else // Swapping values prevents deadlock:
       philosopher[i] =new Philosopher(first,left);
     // Optionally break out of program:
     if(args.length >= 4) {
       int delay = Integer.parseInt(args[3]);
       if(delay != 0)
         new Timeout(delay * 1000, "Timed out");
     }
   }
}
```

wait and notify

- **suspend()** and **resume()** belong to Thread class
- **wait()** and **notify()** are methods of root class, so have access to locks

Wait

- called in a thread when that thread wants to wait for some condition (that condition is supposed to be set by another thread).
  - Don't idly wait while testing the condition inside your thread. This is called a busy wait and it is a very bad use of CPU cycles.
- The thread that calls wait() must already own the lock.
- If you call it out of synchronization block, the program will compile, but when you run it, you will get an IllegalMonitorStateException with message "current thread not owner".

- The thread then releases the lock and waits until it is notified with notify() or notifyAll().
- When it is notified, the thread must try to get the lock again.
- When it re-gains the lock, its execution continues from the line after wait(). Usually, it will need to test for the condition that made it wait again. If you have either been notified of something that doesn't concern the condition (as can happen with notifyAll( )), or the condition changes before you get fully out of the wait loop, you are guaranteed to go back into waiting.. This is why we usually see wait() inside a while loop.

Notify

- Like wait, the thread that can call notify() must already own a lock.
  - It is guaranteed that two threads trying to call notify( ) on one object won't step on each other's toes.
- The method notify() chooses a waiting thread from the set of waiting thread (on that particular object).
- The chosen thread then is considered "notified" and it acts according to what is defined for the wait method, just above.
- notifyAll() signals all threads in the set of waiting threads.

Example: As an example, consider a restaurant that has one chef and one waitperson.

- The waitperson must wait for the chef to prepare a meal.
- When the chef has a meal ready, the chef notifies the waitperson, who then gets the meal and goes back to waiting.
- The chef represents the producer, and the waitperson represents the consumer.

```
class Order {
  private static int i = 0;
  private int count = i++;
  public Order() {
    if(count == 10) {
      System.out.println("Out of food, closing");
      System.exit(0);
    }
  }
  public String toString(){
    return "Order " + count;
  }
```

```
}

class WaitPerson extends Thread {
  private Restaurant restaurant;
  public WaitPerson(Restaurant r) {
    restaurant = r;
    start();
  }
  public void run() {
    while(true) {
      while(restaurant.order == null)
        synchronized(this) {
          try {
            wait();
          } catch(InterruptedException e) {
            throw new RuntimeException(e);
          }
        }
      System.out.println(
        "Waitperson got " + restaurant.order);
      restaurant.order = null;
    }
  }
}

class Chef extends Thread {
  private Restaurant restaurant;
  private WaitPerson waitPerson;
  public Chef(Restaurant r, WaitPerson w) {
    restaurant = r;
    waitPerson = w;
    start();
  }
  public void run() {
    while(true) {
      if(restaurant.order == null) {
        restaurant.order = new Order();
        System.out.print("Order up! ");
        synchronized(waitPerson) {
          waitPerson.notify();
        }
      }
      try {
        sleep(100);
      } catch(InterruptedException e) {
        throw new RuntimeException(e);
```

```
      }
    }
  }
}

public class Restaurant {
  Order order; // Package access
  public static void main(String[] args) {
    Restaurant restaurant =new Restaurant();
    WaitPerson waitPerson =
                    new WaitPerson(restaurant);
    Chef chef = new Chef(restaurant, waitPerson);
  }
}
```

The result will be

```
    "Order up! Waitperson got Order 0",
    "Order up! Waitperson got Order 1",
    "Order up! Waitperson got Order 2",
    "Order up! Waitperson got Order 3",
    "Order up! Waitperson got Order 4",
    "Order up! Waitperson got Order 5",
    "Order up! Waitperson got Order 6",
    "Order up! Waitperson got Order 7",
    "Order up! Waitperson got Order 8",
    "Order up! Waitperson got Order 9",
    "Out of food, closing"
```

Simulating Suspend and Resume

```
public class Suspend extends JApplet {
  private JTextField t = new JTextField(10);
  private JButton
    suspend = new JButton("Suspend"),
    resume = new JButton("Resume");
  private Suspendable ss = new Suspendable();

  class Suspendable extends Thread {
    private int count = 0;
    private boolean suspended = false;
    public Suspendable() { start(); }
    public void fauxSuspend() {
      suspended = true;
    }
    public synchronized void fauxResume() {
      suspended = false;
```

```
    notify();
  }
 - - - - - - - - - - - - - - - - - - -
 public void run() {
    while (true) {
      try {
        sleep(100);
        synchronized(this) {
          while(suspended)
            wait();
        }
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
      t.setText(Integer.toString(count++));
    }
  }
}

public void init() {
  Container cp = getContentPane();
  cp.setLayout(new FlowLayout());
  cp.add(t);
  suspend.addActionListener(
    new ActionListener() {
      public
      void actionPerformed(ActionEvent e) {
        ss.fauxSuspend();
      }
    });
  cp.add(suspend);
  resume.addActionListener(
    new ActionListener() {
      public
      void actionPerformed(ActionEvent e) {
        ss.fauxResume(); //called from main
      }
    });
  cp.add(resume);
}
```

Volatile variable

If you define a variable as volatile,

- It tells the compiler not to do any optimizations that would remove reads and writes

- Thus keeping the field in exact synchronization with the local data in the threads.

For example, we may have a class

```
public class SerialNumberGenerator {
  private static volatile int serialNumber = 0;
  public static int nextSerialNumber() {
    return serialNumber++;
  }
}
```

In this class, a compiler may save serialNumber locally. So serialNumber used by the class will be different from serialNumber used by another class. Declaring it volatile forces read and write to the shared memory, so that the data are synchronized.

Thread Priority

The priority of a thread

- tells the scheduler how important this thread is.
- If there are a number of threads blocked and waiting to be run, the scheduler will lean toward the one with the highest priority first.
- This doesn't mean that threads with lower priority aren't run. Lower priority threads just tend to run less often.
- The priorities are adjusted by using Thread's setPriority( ).
- priority level of thread 1 is at the highest level.
- use setPriority(priority) to set the priority of each thread.

Daemon threads

- It is supposed to provide a general service in the background as long as the program is running
- It is not part of the essence of the program.
- When all of the non-daemon threads complete, the program is terminated.
- If there are any non-daemon threads still running, the program doesnt terminate.

Example:

```
public class SimpleDaemons extends Thread {
  public SimpleDaemons() {
    setDaemon(true); // Must be called before start()
    start();
  }
  public void run() {
    while(true) {
      try {
        sleep(100);
      } catch (InterruptedException e) {
        throw new RuntimeException(e);
      }
      System.out.println(this);
    }
  }
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new SimpleDaemons();
  }
}
```

- Set the thread to be a daemon by calling setDaemon( ) before it is started.
- Once the threads are all started, the program terminates immediately
- Thus, the program terminates without printing any output.
- You can find out if a thread is a daemon by calling isDaemon( ).
- If a thread is a daemon, then any threads it creates will automatically be daemons.

Join

- One thread may call join( ) on another thread to wait for the second thread to complete before proceeding.
- If a thread calls t.join( ) on another thread t,
  - then the calling thread is suspended until the target thread t finishes (when t.isAlive( ) is false).
- You may also call join( ) with a timeout argument (in either milliseconds or milliseconds and nanoseconds) so that if the target thread doesnt finish in that period of time, the call to join( ) returns anyway.
- The call to join( ) may be aborted by calling interrupt( ) on the calling thread, so a try-catch clause is required.

Example: A Joiner is a thread that waits for a Sleeper to wake up by calling join( ) on the Sleeper object.

```
class Sleeper extends Thread {
  private int duration;
  public Sleeper(String name, int sleepTime) {
    super(name);
    duration = sleepTime;
    start();
  }
  public void run() {
    try {
      sleep(duration);
    } catch (InterruptedException e) {
      System.out.println(
         getName()+"was interrupted."+
        "isInterrupted(): " + isInterrupted());
      return;
    }
    System.out.println(getName() + " has awakened");
  }
}

class Joiner extends Thread {
  private Sleeper sleeper;
  public Joiner(String name, Sleeper sleeper) {
    super(name);
    this.sleeper = sleeper;
    start();
  }
  public void run() {
   try {
      sleeper.join();
    } catch (InterruptedException e) {
      throw new RuntimeException(e);
    }
    System.out.println(getName()+"join completed");
  }
}

public class Joining {
  public static void main(String[] args) {
    Sleeper
      sleepy = new Sleeper("Sleepy", 1500),
      grumpy = new Sleeper("Grumpy", 1500);
    Joiner
```

```
      dopey = new Joiner("Dopey", sleepy),
      doc = new Joiner("Doc", grumpy);
    grumpy.interrupt();
  }
}
```

The printed result will be

```
"Grumpy was interrupted. isInterrupted(): false",
"Doc join completed",
"Sleepy has awakened",
"Dopey join completed"
```