Threads

Java Threads

- A "thread" means computer code being executed.
- More than one thread can be executed virtually simultaneously (interleaved).
  - The code for the threads can be the **same**, or different.
  - Each thread has its own state, **sort of**.
  - Threads can **share variables**, and modify the variables they share.
- Programs with > 1 thread are called "concurrent programs".
- With multiple "cores" (processors), threads can run physically simultaneously, in principle.

Multi-Processing

- e.g. Jaguar supercomputer at Oak Ridge National Lab:
  - 1,000,000,000,000,000 (1 quadrillion) floating-point operations per second (= 1 petaflop)
  - 182,000 AMD quad-core Opterons, running at 2.3 gigahertz
  - 362 terabytes of memory (with 578 terabytes per second of memory bandwidth)

Contrasts

	Jaguar SC	Human Brain
Capacity	1.69×10^{14} transistors among 728,000 processors + 362×10^{12} bits memory	100×10^{12} connections over 100×10^{9} neurons
Speed	2.3 GHz	1 kz

Timing of Threads

- Threads don't progress in lock-step fashion.
- One may be started and another stopped in an **unpredictable** fashion by the operating system.
- This behavior is called **asynchronous**.

Similar Idea: Processes

- A process is also code in execution.
- Typically processes don't share variables, although limited types of sharing are possible.
- Using multiple processes is common in, e.g. UNIX:

  generate | filter | test | display (here | means "pipe")
- Processes are "heavy weight", threads are "light weight".
- "Weight" refers to the amount of information that needs to be changed in switching the processor from one context to another.

Why are Threads Useful?

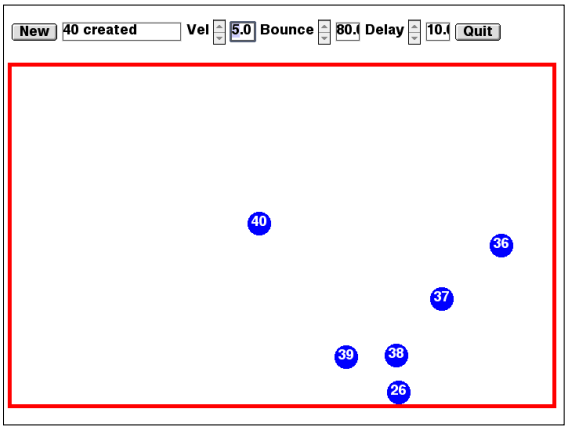
- May wish to have multiple activities going on at once.
- Don't want one activity's waiting (e.g. for an event) to stop the other activities.
- Example: On a user's desktop, there appear to be running simultaneously:
  - Several application programs:
    - A text editor
    - A browser, with several things going on
      - A couple of searches
      - An applet
      - A YouTube video
- This is **only** doable on a 1-processor system with threads (or processes).

Thread Example

- On thread might be a **computational** one, that occasionally needs to wait for input from the outside, say from an input stream of characters.
- Another thread might be a **graphical user interface**, responding to mouse events.
- We don't want **waiting** for input to hold up the graphics, or waiting for a click to hold up the computational thread.
- In fact, the click might tell the computational thread to alter its behavior.

Bouncing Balls Example

- Each ball is run by a separate thread.
- (This is for illustration. It is likely not the way you'd do a video game, because you want more precise control over timing and interactions.)
- Each thread can, in principal, be **interrupted** and re-started independently of the others.
- If a ball is "clicked" in mid-air, it will suspend, and resume if clicked a second time.



Two Ways to Have Threads in Java

- extends Thread
  - Thread is a *base class* with threading capability.
- implements Runnable
  - Runnable is an *interface* that requires method
    - void run()
- **The latter is preferred**, because it does not take away the ability to inherit from another class (multiple inheritance is not allowed in Java).

Using "implements Runnable"

- The class that *implements* Runnable *still* needs to ***contain*** a Thread as an instance variable.
- This Thread is what controls starting and stopping.

Ball "extends Thread" Code

```
/**
 * Ball class represents ball's state information
 */

class Ball extends Thread // vs. Thread implements Runnable
  {
  double x, y;                          // this ball's coordinates
  double deltaX, deltaY;                // this ball' velocities
  String myNumber;                      // ball's number as a string

  public Ball(...) // constructor {}

  /**
   * over-ride run() method in parent class (Thread)
   */

  public void run()
    {
    while( true )
      {
      move();                           // move the ball
      sleep(app.delay);                 // sleep (defined in Thread)
      }
    }
```

Ball "implements Runnable" Code

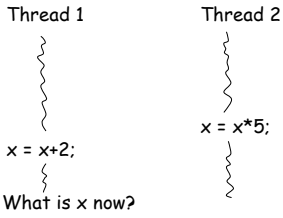
```
class Ball implements Runnable
{
Thread myThread;            // this ball's thread
double x, y;                // this ball's coordinates
double deltaX, deltaY;      // this ball' velocities
String myNumber;            // ball's number as a string

Ball( . . .)                // constructor
  {
  ...
  myThread = new Thread(this);     // make thread for Ball
  }

public void run()                  // run method for this Runnable
    {
    while( true )
      {
      move();                      // move the ball
      myThread.sleep(app.delay);   // sleep
      }
    }
...
}
```

Cautions about Threads

- Reasoning about concurrent programs is inherently more difficult than reasoning about sequential ones.
- They can exhibit **non-deterministic** behavior, when variables are shared among threads.

Non-Determinism

Suppose x == 1 initially.



Language Aspects

- Prior to Java, many languages did not have threads as part of the language.
- Those that did were mostly research vehicles.
- Some had add-on libraries for threads (such as pthreads or Posix-threads).
- Java is the most widely-used example where threads are integral to the language.
- The JVM (Java Virtual Machine) is the interpreter for Java's byte-code. It runs the threads.

Interesting Methods of Thread

`public void` **`start`**`()`

Causes this thread to begin execution;
The JVM calls the `run` method of this thread.

The result is that *two threads are running concurrently*:

the initiating thread (which returns from the call to the `start` method) and

the initiated thread (which executes its `run` method).

Throws:
`IllegalThreadStateException` - if the thread was already started.

Methods of Thread

```
public static Thread currentThread()
```

Returns a reference to the currently executing thread object.

Note: "executing" is more specific than "running":

"executing" means "has the processor"

"running" means "able to execute",
but not necessarily executing

Methods of Thread

```
public static void yield()
```

Causes the currently executing thread object to pause temporarily and allow other threads to execute.

Methods of Thread

```
public static void sleep(long millisec)
                  throws InterruptedException
```

Causes the currently executing thread to sleep
(temporarily stop execution)
for the specified number of milliseconds.

Methods of Thread

```
public void interrupt()
```

Interrupts this thread.

Called by another thread having a reference to this one.

First the checkAccess method of this thread is invoked, which may cause a SecurityException to be thrown.

Methods of Thread

```
public final void setPriority(int newPriority)
```

Changes the priority of this thread.

First the checkAccess method of this thread is called with no arguments. This may result in throwing a SecurityException.

Otherwise, the priority of this thread is set to the smaller of the specified newPriority and the maximum permitted priority of the thread's thread group.

Methods of Thread

```
public final void join(long millisrv)
                throws InterruptedException
```

Waits at most millisec milliseconds for this thread to die.
A timeout of 0 means to wait forever.

Runnable

java.lang

Interface Runnable

Known Implementing Classes:
Thread, TimerTask

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called run.

This interface is designed to provide a common protocol for objects that wish to execute code while they are active. For example, Runnable is implemented by class Thread.
Being active simply means that a thread has been started and has not yet been stopped.

In addition, Runnable provides the means for a class to be active while not subclassing Thread.
A class that implements Runnable can run without subclassing Thread by instantiating a Thread instance and passing itself in as the target.

In most cases, the Runnable interface should be used if you are only planning to override the run() method and no other Thread methods.
This is important because classes should not be subclassed unless the programmer intends on modifying or enhancing the fundamental behavior of the class.

re. Applets

- As you have seen, applets implement Runnable.
- This is in part so the applet can carry out two activities **concurrently**:
  - The main activity or activities of the applet
  - The event-listening activities that deal with user events such as pushing a button, etc.
  - The latter call user-supplied methods, known as **listeners** or **call-backs**, enabling communication with the main activity through common variables.