Class 21:
Designing Event Listeners
Number Parsing
Multi-Threading &
Java Threads
Designing Event Listeners
Issues for Event Listeners

- How to handle multiple event sources.
- How to handle similarities and differences.
- How does listener know what to do?
Criteria

z What is easiest:
  y To program?
  y To understand?
  y To fix?
  y To update or extend?

z Example: Multiple buttons
Options for Event Listeners

z Approaches
  y Separate listener class and object for each button.
    x Everything hard-coded.
    x Decided at compile time.
  y Same listener class, separate object for each button.
    x Listener remembers its button.
    x Decided at instantiation time.
  y One object for all buttons
    x Queries which button caused event: getSource().
    x Decided at event time.

z Different approaches for different button sets.
Sample Code

- Button10.java
- Button11.java
- Button12.java
  - On the course Web site.
Number Parsing
In class Integer, public static int parseInt(String s) throws NumberFormatException.

- Converts String to integer.
- Throws NumberFormatException if the String is not a valid integer.
- Can also specify radix.

Analogous methods for other wrapper classes.
Parsing FlexNumbers

- Look at the String.
  - If it has '{', it must be FlexComplex.
  - Otherwise, if it has '.' it must be FlexDouble.
  - Otherwise it must be FlexLong.

- Parse the number parts and invoke the right constructor.
Multi-Threading and Java Threads
Multi-Whatever

z What you want:
  y A computer that runs multiple programs at the same time.
  y A program with separate independently-running subtasks.

z What you usually have:
  y A single CPU, which can run one program at a time.
  y Software which makes it appear that multiple programs are running, by giving them each part of the CPU’s time.
Historical Terminology

Variations of this capability have been called:

- Multiprogramming
- Multiprocessing
- Multitasking
- Foreground-background programming
- Partitions
- Multithreading
Current Terminology: Process / Task

A process (or task) is a self-contained instance of a program, **running in its own address space**.

A multi-tasking operating system (e.g. Unix) runs multiple tasks (processes) at a time. Each thinks it has the CPU to itself.

- Preemptive multitasking means the OS can switch tasks at any time.
- Cooperative multitasking means the OS must wait for each task to give up the CPU.
Current Terminology: Threads

z A “thread” is a single sequential flow of control within one process.

y Lighter weight than a separate process.

y Memory is shared among threads.

y Typically tie threads to events, e.g.
   x Button push
   x Timer
   x Menu Choice

z Java supports threads.
Communication Between Processes

- Messages
- Signals
- Semaphores
- Sockets
- Pipes
- Files
- (Possibly) Shared Memory
Communication Between Threads

- In Java, each thread is an object.
- Threads share the same memory space.
- All objects and variables are visible to a method, subject to the standard scope rules.
Java Threads

Each thread is an object, so:

- Define a class which either:
  - inherits from class Thread. --OR--
  - implements the Runnable interface.

- Override the run() method.
Java Threads II

z To create a thread:

y Instantiate an object with new on the class you defined.

y Invoke start() on the thread object.

x This initializes the thread and invokes its run() method.
Java Threads III

1. No need to explicitly switch between threads.
   - Java does it more-or-less at random.

2. OK to have multiple thread objects, instantiated from the same and/or different classes.
The run() Method

- Could exit using the return command.
- Could be an infinite loop.
  - Will run until the entire program stops.
    - Java will occasionally switch to another thread.
  - Should use a sleep() or something to keep it from hogging the CPU.
    - Can still be preempted without sleep(), but the whole system will be less responsive.
Example: SimpleThread.java

- Starts 5 threads.
- Each thread prints out the numbers from 5 down to 1.
A GUI Example

- Remember how buttons work.
  - Each button is an object.
  - Create an ActiveListener object for each button.
  - “add” the ActiveListener object to the button object.
  - When the user presses the button, the actionPerformed() method of the ActiveListener is invoked.
Counter2.java

- `main()` creates the applet and a window for it to run in.
- The applet creates the buttons and a text field.
- In the applet is a variable called `sp`, which holds the reference to the thread object.
When the user presses the Start button, if `sp == null` (i.e. no thread yet), it starts the new thread.

The thread has a private variable `c2` which holds a reference to the main program, so it can access the text field.

When the user presses the Toggle (onOff) button, it invokes `invertFlag()` on the thread.
When the subtask is instantiated, it runs `start()`. The standard `start()` method invokes the `run()` method. The overriding one, not the standard one.

What `run()` does:

- Sleep 100 msec.
- If `runFlag` is true, increment count and display it.
- Repeat forever.

Hitting the Toggle button flips the `runFlag`. 
An Alternative Approach

- Program Counter3.java combines the thread with the main class.
  - Implements Runnable instead of inheriting from Thread.
- On the course Web site.
A More Complex Example: MyCounter4.java

- Creates multiple threads
  - Set on command line.
  - Default to 5.
- Each thread has a text field and a button.
- Counters start and stop.
- Buttons and fields change color.
- On the course Web site.
Next Times

Tomorrow: More threads

Thursday:
  - Java I/O
  - Java Networking

Friday: The Future of Java

Monday: Review for Exam 2

Tuesday: Exam 2 (Java)