# CS 11 java track: lecture 5

- This week:
  - the final keyword
  - introduction to threads
  - design advice for lab 5

# final

- the many meanings of the final keyword:
  - final classes
  - final methods
  - final fields
  - final local variables
  - final arguments to methods

# final classes (1)

- a final class is a class that cannot be subclassed
- why do this?
  - efficiency: method access is faster
- example of final class:
  - String

# final classes (2)

```
public final class LastVersion {
    // cannot subclass this
}

// or:
public final class LastVersion extends Foo {
    // OK; final class can extend
    // other classes
}
```

# final methods (1)

- final methods cannot be overridden by subclasses
- why use?
  - efficiency again
  - want to force certain behaviors

# final methods (2)

```
public class Foo {
    // really lame example:
    public final void printme() {
        System.out.println("I am a Foo!");
    }
}
```

# final fields (1)

- final fields cannot be changed once set
  - sort of like a constant value
  - "write-once"
- final fields are used to represent symbolic constants
  - usually also `public` and `static`
  - common to see lots of these at top of some classes

# final fields (2)

```
public class Foo {
    public final int val1;
    public final int val2 = 100;
    public static final int VAL3 = 200;

    public Foo() {
        val1 = 50;
    }
}
```

# final local variables (1)

- almost the same as final fields
- some obscure rules

```
public void someMethod() {
    final int n1 = 100;
    final int n2; // not set yet...
    n2 = 200;
}
```

# final local variables (2)

- one obscure rule:
  - anonymous inner classes can't refer to local variables in the method they're defined in
  - unless those variables are final
- why?
  - no good reason
  - reflects the details of java's implementation
  - get used to it! ☹

# final arguments to methods

- methods can have `final` arguments:

```
public class Foo {
    public int someMethod(final int x, final int y) {
        // cannot change x and y
    }
}
```

- final arguments can't be changed in body of method

# threads (1)

- this week's assignment: the game of Life
- need to do two things concurrently:
  - update the game board
  - allow the user to start, stop, step at any time
- in other words, need to do *multiple things at once*
- that's what threads help you do

# threads (2)

- threads are *by far* the hardest and most confusing part of java
- not java's fault; java makes them about as easy as they can be made
- threads are *inherently* tricky
- you will see brand-new kinds of bugs you didn't know existed
  - *e.g.* deadlock

# threads (3)

- the good news: this week's lab uses threads in a very easy way
- the bad news: week 7's lab uses threads in a trickier way
  - but I'll help you out
- you can take whole courses to learn about threads and how they impact software design

# threads (4)

- what is a thread?
- it's like a separate process that uses the same data as the other processes
- primitive kind of parallel programming
- it's the sharing of data that causes most of the problems
  - synchronizing access to data between threads can be tricky

# threads in java (1)

- look these up in the java API:
  - java.lang.Thread class
  - Runnable interface

```
public interface Runnable {
    public void run();
}
```

# threads in java (2)

- read this:

http://java.sun.com/docs/books/tutorial/essential/threads/index.html

- read sections:
  - What is a Thread?
  - Customizing a Thread's run method
    - Implementing the Runnable interface
  - The life cycle of a Thread

# threads in java (3)

- threads are implemented on a per-object basis
  - any class that implements Runnable can run in its own thread
- multiple threads can run on the same object!
  - like having several processes manipulating the same data, calling methods on the same object, etc.
  - coordinating accesses from multiple threads is tricky, but we'll ignore for now (see lab 7)

# threads in java (4)

- by default, application is run in a single thread
  - also another thread for garbage collector, but we don't usually worry about it
- Thread constructor:
  - **`Thread(Runnable target)`**
  - allocates a new Thread object
  - doesn't start the Thread running

# threads in lab 5 (1)

- In lab 5:
  - one thread is running the game of life
    - updating display
  - another thread is waiting for button clicks
    - start, stop, step

# threads in lab 5 (2)

```
public class LifeGUI implements Runnable {
    Thread t;     // represents this object's thread
    LifeGUI gui;  // represents this object

    public LifeGUI() {
        gui = this;  // why do we need this?
        // other stuff...
    }

    public void run() {
        // run the thread...
    }
}
```

# threads in lab 5 (3)

- threads must be explicitly started using their **`start()`** method
- threads continue until reaching the end of their **`run()`** method
  - then they stop and are destroyed
  - this is the only good way to stop a thread

# threads in lab 5 (4)

- can put a thread to sleep:

```
static void sleep(long milliseconds)
```

  - causes the currently executing thread to sleep (temporarily cease execution) for the specified time

# threads in lab 5 (5)

- creating a new thread:
  - when the "go" button is pushed...
  - create and start a new thread

```
JButton go = new JButton("go");
go.addActionListener(new ActionListener() {
    public void ActionPerformed(ActionEvent e) {
        t = new Thread(gui);  // create the thread
        // NOT: t = new Thread(this); WHY?
        t.start();  // start the thread
    };
```

# threads in lab 5 (6)

- when the "stop" button is pushed, `null` out the thread:

```
t = null;
```

- `run()` looks like this:

```
public void run() {
    // Check if t is null; if not, step and
    // then sleep for some time.
    // Keep going until you're interrupted,
    // then return.
    // (This is an infinite loop).
}
```

# design advice (1)

- top-level class: **`Lab5`**
  - handles command-line arguments (if any)
  - creates the graphical interface
  - starts it up
  - and that's all

# design advice (2)

- class: `Life`
  - instances store the state, perform updates for one life board
  - nothing to do with graphics at all
  - should be usable on its own if desired
  - "separation of concerns"
    - don't mix graphics with computation unnecessarily

# design advice (3)

- class: **`LifeGUI`**
  - instances contain a **`Life`** instance as a field
  - implements **`Runnable`** interface
  - has 2D array of **`LifeButtons`** (next slide)
  - purposes:
    - displays states of buttons
    - allows users to interact with buttons
    - allows users to start/stop/step the game
    - updates the life board

# design advice (4)

- class: **`LifeButton`**
  - instances hold state for a single x-y location on the life board
  - hold reference to a **`Life`** instance so they can change the state of that instance when the button is clicked
  - must have a method that allows **`LifeGUI`** instance to flip state of button

# design advice (5)

- button callbacks:
  - use anonymous inner classes to implement response to clicking buttons:
    - go
    - stop
    - step

# next week

- networking basics
  - `Socket` class
- `Vector` class
- Collection classes and Iterators
- parsing strings