

---

# CS11 – Java

---

Spring 2010-2011

Lecture 5

---

# Today's Topics

- Introduction to Java threads
- Swing and threading
- Lab 5 Hints



---

# Java Threads

- A “thread of execution” is a single, sequential flow of execution through your program
    - Threads have a beginning and an end
    - A thread does only one thing at a time
  - All programs have at least one thread of execution
    - The “main thread” runs your `main()` method
  - Multi-threaded programs have several threads of execution
    - They can do multiple things “at the same time”
-

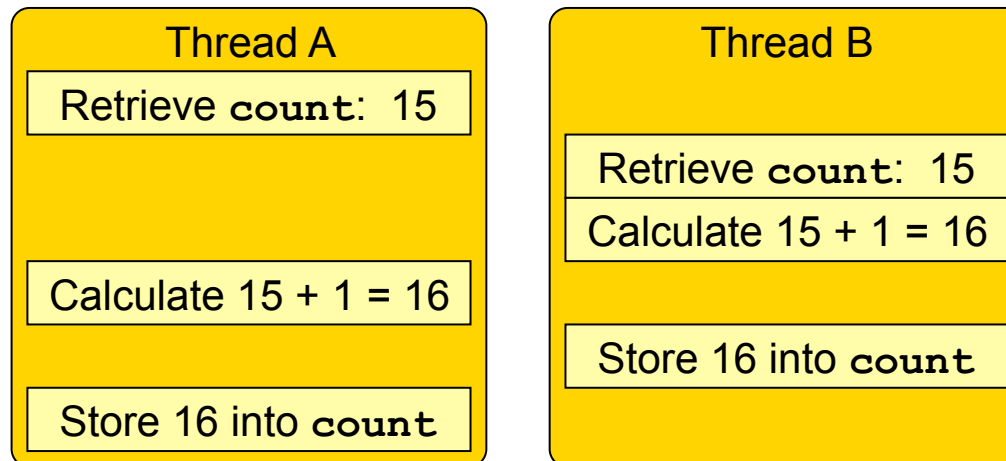
---

# Standard Java Threads

- The Java VM uses multiple threads
    - The main thread runs your program
    - The garbage-collector may use a thread
    - Java AWT/Swing starts its own thread
      - For event-dispatching
    - Some Java library classes use threads internally
  - You can start your own threads too
    - This week's lab doesn't need them (phew!)
-

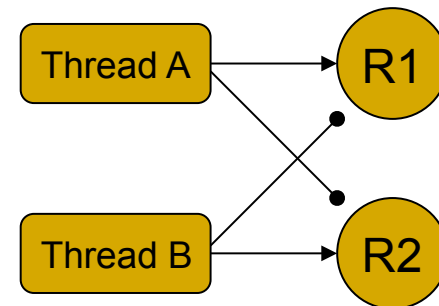
# Threads and Resources

- A thread can have local resources; only used by the thread
- Threads can also share resources between each other
  - This can lead to many problems
- One big problem: interleaved access
  - Example: `count` is a shared variable. Assume `count = 15`.
  - Two threads executing `count = count + 1;`



# Locking Shared Resources

- Shared resources must be manipulated atomically
  - ❑ Only allow one thread to access shared resource at a time
  - ❑ Shared resources can be locked by a thread
- If threads can lock multiple shared resources, deadlock can occur
  - ❑ Thread A locks resource R1
  - ❑ Thread B locks resource R2
  - ❑ Thread A tries to lock resource R2...
  - ❑ Thread B tries to lock resource R1...
  - ❑ Locking order is the issue here.



---

# Swing and Thread-Safety

- Swing has its own thread for event handling
    - the event dispatcher thread
  - ...but, Swing components *aren't* thread-safe!
  - To be thread-safe in Swing:
    - Once a Swing component has been made visible, only interact with it from event dispatcher thread.
  - Initializing a Swing UI from another thread is fine (it hasn't been made visible yet)
    - e.g. usually done from the main thread
-

---

# Shutting Down a GUI Application

- In Java AWT, closing a **Frame** just hides the window
    - If you don't do something special, application keeps running
    - Have to register a **WindowListener** impl to exit application when window closes
  - In Swing, **JFrame** gives you options

```
JFrame f = new JFrame("My App!");  
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

    - Default is **HIDE\_ON\_CLOSE**; like AWT Frame
-

---

# Long-Running Tasks and Swing

- Very common to have UIs performing long-running tasks
    - e.g. web browsers frequently have large files to download when displaying a web page, etc.
  - Problem:
    - If long-running operation is performed on the event-dispatch thread, can't process events!
    - There is only one event-dispatch thread. If it's tied up with work, the UI will *freeze* until work is done.
-

---

# Long-Running Tasks and Swing (2)

- Swing provides a solution to this issue:
    - `javax.swing.SwingWorker`
  - Can dispatch a long-running task on a worker thread, in the background
    - Task won't tie up the event-dispatch thread
    - Can still interact with the user interface while task is being completed
  - When task is finished, `SwingWorker`'s results are made available on event-dispatch thread
    - Can update user interface with results of task
-

---

# SwingWorker Details

- **SwingWorker** is an abstract class
    - Must be subclassed to perform specific tasks
  - Several important methods:
    - **protected Object doInBackground()**
      - Implement this method to perform the long-running task
      - This method is never called on the event-dispatch thread
      - (uses a small thread-pool of worker threads)
    - **protected void done()**
      - This method is always called on event-dispatch thread!
      - Implement this method to update your Swing GUI with results of long-running task
-

---

# SwingWorker<T, V> Details

- **SwingWorker** is also a generic class
    - Can (should) specify type parameters
  - Type **T** specifies what **doInBackground()** returns
    - **protected T doInBackground()**
  - If your **doInBackground()** implementation doesn't return anything:
    - Just set **T** to **Object**, and return **null**
-

---

## SwingWorker<T, V> Details (2)

- Type **V** represents *intermediate state*
    - Some tasks generate intermediate results that need to be represented in the user interface
      - (Many tasks do not, so not every **SwingWorker** subclass uses this functionality)
    - In these cases, task's **doInBackground()** calls:
      - **protected void publish(V[] chunks)**
      - Whenever intermediate state must be published, this can be called
    - Causes this method to be called on the event-dispatcher thread:
      - **protected void process(List<V> chunks)**
-

---

## SwingWorker<T, V> Details (3)

- As before, if your **SwingWorker** task doesn't publish intermediate state:
  - Just set **v** to **Object**, and don't use **publish()** method



---

# Arrays in Java

- In Java, arrays are also objects

- Some different syntax though!

- Example:

```
int[] myInts = new int[10]; // Allocate the array.
for (int i = 0; i < myInts.length; i++) {
    myInts[i] = 100 * i; // Store stuff in it.
}
```

- In Java, *all* arrays are dynamically allocated
  - Elements are accessed with brackets (like C/C++)
  - Arrays expose a **length** field, indicating their size
  - **length** is read-only (of course)
-

---

# Array Variables

- Array-types have brackets after *type*, not after variable name
    - ❑ `String[] names;` vs. `String names[];`
    - ❑ Latter form is supported, but is discouraged.
  - Can declare array-variables without assigning
    - ❑ `boolean[] flags; // Array of boolean values`
    - ❑ `float[] weights; // Array of floats`
  - Must initialize them before using
    - ❑ Can allocate new array with `new type[size];`
      - *size* can be zero! Called an “empty array.”
    - ❑ Can assign an existing array to the variable
      - (Java arrays are basically objects with additional syntax)
    - ❑ Can set to `null` too!
-

---

# More Array Initialization

- Can also assign specific values to arrays

```
String[] colorNames = {  
    "puce", "mauve", "fuchsia", "chartreuse", "umber"  
};  
// colorNames.length == 5
```

- Syntactic sugar for the initialization operations
- Can still reassign and reinitialize such arrays
  - `colorNames` is a reference to an array of `String` objects



---

# Arrays of Objects

- Arrays of objects initially contain `null` values
  - Array initialization does not initialize object-references
  - Must do that in a separate step

- Example:

```
// Allocate an array of 20 point-references
Point2d[] points = new Point2d[20];

// Make a new Point2d object for each elem
for (int i = 0; i < points.length; i++)
    points[i] = new Point2d();
```

---

---

# Arrays of Arrays

- Arrays can contain other arrays

```
int[][] nums2d; // Array of arrays of ints.
```

- First the array-of-arrays is allocated:

```
nums2d = new int[20][];
```

- Each element of `nums2d` is of type `int[]`.

- Next, each inner array is allocated

```
for (int i = 0; i < nums2d.length; i++)  
    nums2d[i] = new int[50];
```

- When size is fixed, Java provides a shortcut

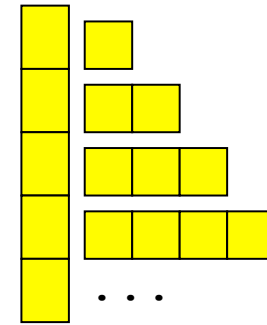
```
int[][] nums2d = new int[20][50]; // Same thing!
```

---

# More Arrays of Arrays

- Inner arrays can be different sizes, if need be

```
int[][] reducedMatrix;  
reducedMatrix = new int[20][];  
for (int i = 1; i <= 20; i++)  
    reducedMatrix[i] = new int[i];
```



- Can't do this with the shortcut syntax

- Can also specify nested initial values

```
double[][] weights = {  
    {3.1, 2.6}, {1.5, 4.4, -3.6}, null, {6.2}  
};
```

---

# Copying Arrays

- Use `System.arraycopy()` to copy one array to another efficiently
  - Can use `clone()` method to duplicate array
    - Result's type is `Object`; must cast to proper type

```
int[] nums = new int[35];  
...  
int[] numsCopy = (int[]) nums.clone();
```
    - Copy is shallow – only top-level array is copied!
      - If array of objects, the objects are not cloned
      - If array of arrays, subarrays are not cloned either
-

---

# Next Week

- Java Sockets API
- **String** processing

