Lecture 11
MVC/JTable, Exceptions and Files

Manu Kumar
sneaker@stanford.edu

Recap

• Last Time
  – Thread Interruption
  – Cooperation
  • Wait/notify
  • Swing/GUI Threading
  – SwingThread Demo
  – Threading conclusions
• Assigned Work Reminder
  – HW 3a: ThreadBank
  – HW 3b: LinkTester
  • Both due before midnight on Wednesday, August 6th, 2003
  • Done with HW3a??

Today

• Today:
  – More HW3b intuition…
  – MVC
    • Model View Controller paradigm
    • JTable
  – Exceptions
  – Files and Streams

Homework #3 Part b intuition

• How many of you have not used Napster/Kazaa/Bearshare! ☺
  – The interface HW3 presents for checking links is reminiscent of how P2P filesharing clients download files.

MVC

• MVC paradigm
  – Model
    • Data storage, no presentation elements
  – View
    • No data storage, presentation elements
  – Controller
    • Glue to tie the Model and the view together
• Motivation
  – Provides for a good way to partition work and create a modular design
  – Very flexible paradigm for providing multiple ways to look at the same information
**Model**
- aka Data Model
  - Storage, not presentation
  - Knows data, not pixels
  - Support data model operations
    - Cut/copy/paste, File Saving, undo, networked data
      - All operations on the model
      - work out logic for file save or undo, without worrying about pixels

**View / Controller**
- View
  - Presentation layer
    - Gets all the data from the model and draws or otherwise renders for the user
- Controller
  - The logic that glues things together
  - Manage the relationship between the model and the view
    - Most data changes are initiated by user events. Translated into messages to the model
    - The view needs to hear about changes. This is done in Java using Listeners

**Model Role**
- Respond to getter methods to provide data
- Respond to setters to change data
- Manage a list of listeners
  - When receiving a setData() to change data, notify the listeners of the change
    - fireXXXChanged
    - Change notifications express the different changes possible on the model
    - Iterate through the listeners and notify each about the change

**View/Controller Role**
- Has a pointer to the data model
- Doesn’t store any data
- Send getData() to model to get the data as needed
- User edit operations (clicking, typing) in the UI map to setData() messages sent to the model
- Register as a listener to the model and respond to change notifications
  - On change notification, consider doing a getData() to get the new values to update the presentation/pixels
Tables in Swing

• Tables are one of the more involved UI elements in Swing
  – Basic functionality however it easy
  – Learn by pattern matching!
• Resources:
  – Handout has lots of sample code
    • Source for the code in the handout is available in electronic form on the course website
  – Sun’s Java Tutorial on How to Use Tables
    • http://java.sun.com/docs/books/tutorial/uiswing/components/table.html

Use MVC pattern!
  – Model: TableModel
  – View: JTable
  – Controller: UI elements and listener bindings
• JTable
  – Relies on a TableModel for storage
  – Has lots of features to display tabular data
• TableModel Interface
  – getValueAt(), setValueAt(), getRowCount(), getColumnCount() etc.
• TableModelListener Interface
  – tableChanged(TableModelEvent e)

AbstractTableModel

• Implements common functionality for TableModel Interface
  – But it is abstract, so you must extend it
    • getRowCount(), getColumnCount(), getValueAt()
  – Helper methods for things not directly related to storage
    • addTableModelListener(), fire___Changed()
• DefaultTableModel
  – Extends AbstractModel, but uses a Vector implementation

Provided in the course handout
  – Uses ArrayList implementation
  – getValueAt() to access data
  – setValueAt() to change data
  – Notifies of changes by sending fireTable____() methods
  – Handles listeners
• This is what you should follow!
  – Base your code for HW3b on the BasicTableModel code provided in the handout!

BasicTableModel

• First lets look at the Client side
  – i.e. how we use the BasicTableModel to implement the TableFrame example
  – Code walk through in emacs…
TableFrame Example

- Next let's look at the guts of the BasicTableModel...
  - Code walkthrough in emacs...

Table Tips!

- Put the JTable in a JScrollPane
  - This automatically deals with handling space for the header and does the right things!
- To change column widths
  ```java
  TableColumn column = null;
  for (int i = 0; i < 5; i++) {
    column = table.getColumnModel().getColumn(i);
    if (i == 2) {
      column.setPreferredWidth(100); // second column is bigger
    } else {
      column.setPreferredWidth(50);
    }
  }
  ```

MVC Summary

- MVC is used in Swing in many places
  - Model
  - View
  - Controller
- Advantage of MVC
  - 2 small problems vs. 1 big problem
    - Provides a natural decomposition pattern
    - Can solve GUI problems in the GUI domain, the storage etc. is all separate
      - Example: don't have to worry about file saving when implementing scrolling and vice versa

MVC examples

- Networked Multiple Views
  - Model on a central server, different views on clients
- Wrapping Databases
- Web Applications
  - Three-Tier Architecture
    - Application Server
    - Servlets/JSPs

Exceptions

- You've seen these already!
  - So you already have some intuition about these
- Exceptions
  - Are for handling errors
  - Example:
    - ArrayIndexOutOfBoundsException
    - NullPointerException
    - CloneNotSupportedException

Error-Handling

- Programming has two main tasks
  - Do the main computation or task at hand
    - Handle exceptional (rare) failure conditions that may arise
- Bulletproofing
  - Term used to make sure your program can handle all kinds of error conditions
- Warning
  - Since error handling code is not executed very often, it is likely that it will have lots of errors in it!
Traditional Approach to Error Handling

- Main computation and error handling code are mixed together
  
  ```java
  int error = foo(a, &b)
  if (error == 0) { …; }
  ```

- Problems
  - Spaghetti code – less readable
  - Error codes, values have to be manually passed back to calling methods so that the top level caller can do something graceful
  - Compiler does not provide any support for error handling

The Java Way: Exceptions

- Formalize and separate error handling from main code in a structured way
  - Compiler is aware of these “exceptions”
  - Easier to read since it is possible to look at main code, and look at error cases
  - Possible to pass errors gracefully up the calling hierarchy to be handled at the appropriate level

Exception Classes

- Throwable
  - Superclass for all exceptions

- Two main types of exceptions
  - Exception
    - This is something the caller/programmer should know about and handle
    - Must be declared in a throws clause
  - RuntimeException
    - Subclass of exception
    - Does not need to be declared in a throws clause
    - Usually reserved for things which the caller cannot do anything and therefore also usually fatal.

Exception Subclasses

- Exceptions are organized in a hierarchy
  - Subclasses are most specific
  - Higher level exceptions are less specific

- You can create your own subclasses of exceptions which are application specific
  - Rule of thumb: if your client code will need to distinguish a particular error and do something special, create a new exception subclass, otherwise, just use existing classes.

Methods with Exceptions

- Exception throw
  - `throw` can be used to signal an exception at runtime

- Method throws
  - When a method does something that can result in an error, it should declare `throws` in the method declaration
  
  ```java
  public void fileRead(String f) throws IOException {
  ….
  }
  ```

“Handling” Exceptions

- Three possible options
  - Do nothing approach
    - Always a bad idea! Do not use this!!
  - Pass-the-buck-approach
    - Declare the exception in a throws
    - This passes the exception along to the caller to handle
  - Do-Something-approach
    - Use try-catch block to test if an exception can happen and then do something useful

- Which one to use:
  - Depends on the application!
try / catch

- Idea:
  - "try" to do something
  - If it fails "catch" the exception
  - Do something appropriate to deal with the error

- Note:
  - A try may have multiple catches!
    - Depending upon the different types of exceptions that can be thrown by all the statements inside a try block
  - Exceptions are tested in the same order as the catch blocks
    - Important when dealing with exceptions that have a superclass-subclass relationship

printStackTrace() is your friend!

- When dealing with exceptions
- Especially when debugging
- printStackTrace() will:
  - Show you the full calling history
  - With line numbers

- Note:
  - Bad idea to eat an exception silently!
  - Either printStackTrace() or pass it along to be handled at a different level

Exception Patterns: #1

- Inner throws, Outer handler

```java
try {
  if this is the standard way to read a text file...
  FileReader reader = new FileReader(new File(fname));
  BufferedReader in = new BufferedReader(reader);
  String line;
  while ((line = in.readLine()) != null) {
    if readLine() etc. can fail in various ways with
    an IOException
  }
} // Control jumps to the catch clause on an exception
catch (IOException e) {
  // a simple handling strategy -- see below for better strategies
  e.printStackTrace();
}
```

Exception Patterns: #2

- try/catch at every level
  - Usually a bad sign
  - Lower level methods should just identify the problem and pass back the information that it happened
  - More complex try/catch code should be concentrated at one place
    - A try/catch may be used at the lower levels of code if the method deals with the error on it's own and the higher layers don't need to know about it
Exception Patterns: #3

- Multiple catch clauses
  - Possible to have multiple catch clauses for a single try statement
  - Essentially checking for different types of exceptions that may happen
  - Evaluated in the order of the code
    - Bear in mind the Exception hierarchy when writing multiple catch clauses!
    - If you catch Exception first and then IOException, the IOException will never be caught!

Exception Patterns: #3

- Multiple catch clauses example
  ```java
  private void loadXML(File file) {
      try {
          // file opening and XML parsing code
      } catch (SAXException e) {
          System.err.println("XML parse err:" + e.getMessage());
      } catch (IOException e) {
          System.err.println("IO err:" + e.getMessage());
      }
  }
  ```

Exception Patterns: #4

- Clean try/catch
  - Write try/catch statements so that the objects are always left in a consistent state
    - On graceful exit
    - On non-graceful exit
    - Maintain transaction semantics!

Wrong -- unclean
```
class HTTPTester {
    private String[] results;
    private int resultCount;
    // Attempts a connection to the given url and adds the result to the array.
    // Suppose that url responds to a connect() message
    public void test(URL url) {
        try {
            url.connect(); // may throw
            resultCount++;
            results[resultCount-1] = url.getData(); // may throw
        } catch (ConnectException e) {
            // log the exception
        }
    }
    ...
}
```

Correct – fail first
```
class HTTPTester {
    private String[] results;
    private int resultCount;
    // Attempts a connection to the given url and adds the result to the array.
    public void test(URL url) {
        int oldCount = resultCount;
        try {
            url.connect(); // may throw
            resultCount++;
            results[resultCount-1] = url.getData(); // may throw
        } catch (ConnectException e) {
            // log the exception
            if (resultCount > oldCount) {
                results[resultCount-1] = null;
                resultCount = oldCount;
            }
        }
    }
    ...
}
```

Correct – clean up
```
class HTTPTester {
    private String[] results;
    private int resultCount;
    // Attempts a connection to the given url and adds the result to the array.
    public void test(URL url) {
        int oldCount = resultCount;
        try {
            url.connect(); // may throw
            resultCount++;
            results[resultCount-1] = url.getData(); // may throw
        } catch (ConnectException e) {
            // log the exception
            if (resultCount > oldCount) {
                results[resultCount-1] = null;
                resultCount = oldCount;
            }
        }
    }
    ...
}```
finally clause

• Try-catch-finally
  – Finally section includes code that is always executed before the block exits
  • Executes in both graceful and ungraceful cases
  – Usually used for
    • Doing cleanup
      – Closing streams and handles
  – A return statement in the try clause will execute the finally clause before returning
  • This is stylistically not good since it is confusing to the reader

Finally example

```java
public void processFile() {
    processing = true;
    try {
        ...
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        processing = false;
    }
}
```

Files and Streams

• File
  – Represents a file or directory
  – Java abstracts away the ugliness of dealing with files quite nicely

• Streams
  – Way to deal with input and output
  – A useful abstraction…

Files and Streams

Streams!??

• Water analogy
  – Think of streams as pipes for water
  – Do you know whether the water that comes out of your tap is coming from a) the ocean b) some river c) a water tank d) a water buffalo?

  • Idea:
    – You abstract away what the stream is connected to and perform all your I/O operations on the stream
    – The stream may be connected to a file on a floppy, a file on a hard disk, a network connection or may even just be in memory!

Hierarchy of Streams

• Java provides a hierarchy of streams
  – Think of this as different “filters” you can add on to your water pipe
    • Some may compress/decompress data
    • Some may provide buffers

  • Common Use Scenario
    – Streams are used by layering them together to form the type of “pipe” we eventually want

Types of Streams

• InputStream / OutputStream
  – Base class streams with few features
    – read() and write()

• FileInputStream / FileOutputStream
  – Specifically for connecting to files

• ByteArrayInputStream / ByteArrayOutputStream
  – Use an in-memory array of bytes for storage!

• BufferedInputStream / BufferedOutputStream
  – Improve performance by adding buffers
    – Should almost always use buffers

• BufferedReader / BufferedWriter
  – Convert bytes to unicode Char and String data
    – Probably most useful for what we need
• When a thread sends a read() to a stream, if the data is not ready, the thread blocks in the call to read(). When the data is there, the thread unblocks and the call to read() returns
• The reading/writing code does not need to do anything special
• Read 10 things at once – create 10 threads!

```
public void readLines(String fname) {
    try {
        // Build a reader on the name (also works with File object)
        BufferedReader in = new BufferedReader(new FileReader(fname));
        String line;
        while ((line = in.readLine()) != null) {
            // do something with 'line'
            System.out.println(line);
        }
        in.close(); // polite
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

```
public static void dumpURL(String urlString) {
    try {
        URL url = new URL(urlString);
        URLConnection conn = url.openConnection();
        InputStream stream = conn.getInputStream();
        BufferedReader in = new BufferedReader(new InputStreamReader(stream));
        String line;
        while ((line = in.readLine()) != null) {
            System.out.println(line);
        }
        in.close(); // polite
    } catch (MalformedURLException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

• Java has build-in and very elegant support for HTTP
• Code on the handout is what you will need for HW #3 Part b!
• URL
  – Uniform Resource Location
    • http://cs193j.stanford.edu
• URLConnection
  – To open a network connection to a URL and be able to get a stream from it to read data!