CS11 Advanced Java

Winter 2011-2012

Lecture 7

Today's Topics

- Long-running tasks in Swing apps
- A brief overview of Swing dialog windows
- Logging
- Conditional compilation

Boggle Server

- Last week: add the Boggle server!
  - RMI calls from clients to server to start/end games
- Click the "Start Game" button:
  - Client calls the server's **`startGame()`** method…
  - Client freezes for at least 15 seconds!
  - No UI updates at all!
- Problem:
  - RMI call happens on Swing event-dispatch thread!
  - No Swing events can be processed while RMI call is waiting to return

Swing and Long-Running Tasks

- Important rule for Swing UI programming:
  - Don’t perform long-running tasks using the Swing event-dispatcher thread!
  - Blocks the processing of other Swing events, and all UI updates, etc.
- If a Swing app must do long-running tasks:
  - Run the task on a separate worker thread
  - Swing UI code hands the task to the worker thread, then monitors the worker’s progress

`SwingWorker`

- Swing includes a class to handle long-running tasks
  - **`javax.swing.SwingWorker`**
  - Creates and manages a worker thread to execute long-running tasks in a Swing-friendly way
- How to use:
  - Create a subclass of **`SwingWorker`** for your task
  - Implement **`doInBackground()`** method to perform task
    - This method is called from the worker thread automatically
  - Override **`done()`** method to be notified when task is done
    - This method is called on the event-dispatcher thread
    - Can update user-interface from within this method!

Subclassing **`SwingWorker`**

- **`SwingWorker`** uses Java generics
  - **`SwingWorker<T, V>`**
  - **`T`** is the type of the final result produced by task
    - e.g. for call to Boggle server, **`T`** = **`BoggleBoard`**
  - **`V`** is the type of any intermediate results produced
    - Used for tasks that can produce intermediate results
    - If you don’t need it, just specify **`Object`**
- Boggle client example:

```
class StartGameWorker
    extends SwingWorker<BoggleBoard, Object>
```

  - Task returns a Boggle board
  - We don’t care about intermediate results, so specify **`Object`**

Implementing **`doInBackground()`**

- Signature of **`doInBackground()`**:
  - **`protected T doInBackground()`**
- In your implementation:
  - Specify value of **`T`**
  - Can make the method **`public`** as well, but not critical
  - If implementation can throw, let the exceptions propagate!
- Example:

```
@Override
public BoggleBoard doInBackground()
  throws PlayerException, RemoteException {
  return server.startGame(playerName);
}
```

Task Completion

- The **`done()`** method is executed on Swing event-dispatch thread when task is finished
  - **`protected void done()`**
  - Default implementation does nothing
  - Override **`done()`** to perform your own tasks, e.g. updating your Swing UI
  - For Boggle client:
    - Display Boggle board returned from server, start timer, etc.
- Get worker's results by calling **`get()`** method
  - Returns same type as **`doInBackground()`**
  - If **`doInBackground()`** threw, calling **`get()`** will cause an **`ExecutionException`** to be thrown

Using the **`SwingWorker`** Class

- Simple procedure to use Swing workers:
  - Create an instance of the **`SwingWorker`** subclass
  - Call the **`execute()`** method on it
    - Method starts worker thread, then returns immediately
    - Your code can call **`execute()`** from any thread, including event-dispatch thread
- **`SwingWorker`** object can only be used once
  - Cannot reuse a **`SwingWorker`** object!
  - Just create a new one, then call **`execute()`** on it

`SwingWorker` Notes

- Good idea to implement **`SwingWorker`** as a private inner class
  - Can access application's state directly
  - Can manipulate UI objects directly
- The **`get()`** method <u>blocks</u> if the task isn't finished!
  - A good reason to call **`get()`** from inside **`done()`**
  - Can use **`isDone()`** method to check if task is finished
- **`SwingWorker`** has other features too
  - Can cancel in-progress tasks
  - Can produce intermediate results and monitor progress

Dialog Boxes

- Dialog boxes (aka “dialogs”) are pop-up windows
  - Report errors or other important messages to the user
  - Request specific input values from the user
  - Display final results or details of some task to the user
- Two kinds of dialogs:
  - Modal:
    - No other window in the application can receive user input until the dialog window is closed
    - System-modal dialogs block all applications until closed
  - Modeless:
    - Other windows in the application can still receive user input while dialog window is visible
- In Swing, dialog classes derive from **`JDialog`**

`JOptionPane` Dialogs

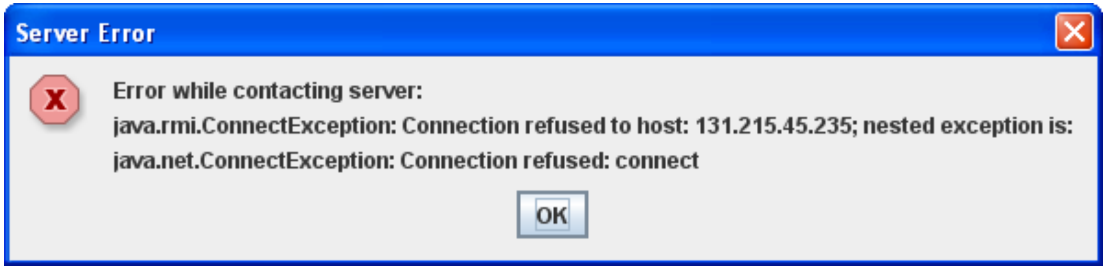
- Swing provides **`JOptionPane`** for most common dialog needs
  - Simple informational or error dialogs
  - Getting a single field of input from user
  - Requesting yes/no-type confirmation from user
- **`JOptionPane`** doesn’t derive from **`JDialog`**!
  - Can’t create a **`JOptionPane`** and show it directly
  - Have to embed a **`JOptionPane`** in a dialog object
- **`JOptionPane`** creates modal dialogs

JOptionPane Dialogs (2)

- **JOptionPane** provides static methods to handle most common dialog tasks
- Example:

```
JOptionPane.showMessageDialog(frame,
  "Error while contacting server:\n" + e.getCause(),
  "Server Error", JOptionPane.ERROR_MESSAGE);
```

  - Shows an error message to the user
  - Specify parent frame so that dialog is centered in the frame
  - Result:



Error while contacting server:
java.rmi.ConnectException: Connection refused to host: 131.215.45.235; nested exception is:
java.net.ConnectException: Connection refused: connect

`JOptionPane` Dialogs (3)

- Static methods:
  - **`showMessageDialog(...)`**
    - Tell the user about something that has happened
  - **`showConfirmDialog(...)`**
    - Asks a question requiring confirmation, e.g. yes/no, ok/cancel, etc.
  - **`showInputDialog(...)`**
    - Prompt the user for a single field of input
  - **`showOptionDialog(...)`**
    - The general-purpose version that exposes all of the above capabilities

Hiding and Disposing Dialogs

- Dialogs are shown by calling:

  ```
  dialog.setVisible(true);
  ```

  - If dialog is modal, this call doesn't return until dialog is closed
- Can hide a dialog (or any window) by calling:

  ```
  dialog.setVisible(false);
  ```

  - The window's UI resources are still held!
- To release a window's UI resources:

  ```
  dialog.dispose();
  ```

  - Will also hide the dialog if it is currently showing

Logging

- For large programs and servers, logging is essential
  - Give developers and users the ability to see, “What in the world is going on?!”
  - In error scenarios, good logs make debugging the system *much* easier
  - In normal operating scenarios, logs can be analyzed to understand usage patterns
- Logging services are provided in all major OSes and widely-used server apps

Information Management (1)

- Log messages typically divided into levels (priorities)
  - **Fatal** – the system cannot continue operating
  - **Error** – the system can handle the situation, but may have reduced capabilities
  - **Warning** – a potentially serious condition was encountered, but its full impact is unclear
  - **Info** – normal details that users may need to know
  - **Trace/Debug** – details that only developers, maintainers, or support personnel would need
- Logging systems usually provide filtering capabilities based on what the user wants to know

Information Management (2)

- Log messages are also grouped by system or component that reports the log message
  - Again, users can filter based on what components or systems they wish to monitor
- Logs can be recorded to several places
  - Most common: the file system
    - Disk usage must be monitored! Log files are typically "rotated" – N most recent are kept; any older are deleted
  - Database storage is sometimes used
  - Other destinations too: SNMP messages, e-mail

Java Logging Frameworks

- Several Java logging frameworks to use!
- Many have sophisticated config options
  - Log storage and management
  - Integration into OS-level logging mechanisms
  - Log formatting – what is in each log message
  - Log filtering based on priority, component, etc.
- Only *slightly* more complicated to use than System.out.println(), and much more powerful
  - Little reason *not* to incorporate such capabilities!

Apache Log4j

- One of the most widely used, powerful, flexible logging frameworks
  - Very mature (released ~1998)
  - Good license! Can be used in commercial projects, etc.
- Highly configurable via several mechanisms
  - Properties file, XML file, etc.
- Loggers form a hierarchy of categories
  - Can configure groups of loggers, or individual loggers
- Log messages have different levels/priorities

Using Log4j

- Retrieve a logger for a specific category

```
Logger logger = Logger.getLogger("server.networking");
```

  - Can easily specify broad categories for application
- Can also specify logger category with **`Class`** object

```
Logger logger = Logger.getLogger(NetHandler.class);
```

  - Class’ package-name and class-name is used for the logging category
- Each category has *exactly one* logger
  - **`Logger.getLogger()`** always returns the same **`Logger`** object for a particular category name
    - (loggers are created the first time they are requested)

Using Log4j (2)

- Typically store the logger instance as a constant in your class
- Example:

```
public class BoggleServerApp implements BoggleServer {
    /** My Boggle server's logger. **/
    private static final Logger logger =
        Logger.getLogger("boggle.server");
    ...
}
```

- Use simple logging methods in your code:

```
logger.debug("Sent data to " + hostName);
logger.info("Transfer complete.");
logger.warn("Client dropped connection.");
logger.error("File " + fileName + " not found.");
logger.fatal("Couldn't open socket on port " + port);
```

Using Log4j (3)

- Logging methods can also report exceptions

```
try {
  Socket s = ...  // Try to open the socket.
}
catch (Exception e) {
  logger.fatal("Couldn't open socket", e);
}
```

  - Second argument is a **`Throwable`**
  - Log4j sends the full stack-trace to logging output

More Efficient Logging

- Logging config can “turn off” different log levels
  - Typically only warnings or worse are reported
  - Saves time of actually formatting and storing the log entries
- Still uses up CPU cycles for the function calls:

  ```
  logger.debug("Sent data to " + hostName);
  ```

  - String allocation, concatenation, garbage-collection
- Can improve this by guarding debug and info logs

  ```
  if (logger.isDebugEnabled())
      logger.debug("Sent data to " + hostName);
  ```

  - Still has *some* overhead, but it’s very small
  - Can always turn on debug logs when needed
  - Can’t do this for warn/error/fatal (but you wouldn’t want to!)

Java Logging APIs

- Added in Java 1.4 (2002)
- Similar concepts to Log4j in many aspects
- Significantly less capabilities than log4j
  - Was introduced after Log4j had gained popularity
  - Log4j has many community-provided extensions
- Only works with Java 1.4+ projects
  - Log4j works with Java 1.2+
  - Definitely not so much of an issue anymore…
- Basic usage is nearly identical to Log4j
  - Beyond that, the APIs diverge rather quickly

Java Libraries and Logging Frameworks

- You want to provide a library of Java classes to other developers, or an embeddable component…
- If your component *needs* to use logging, then:
  - *Ideally,* your library uses same framework as the code that uses it (Log4j, perhaps?)
  - But, they may have chosen something else! Now they have to configure and support two different logging APIs.
- If you can't guarantee what log framework that other code will use, use a generic wrapper-API
  - Users of your library can incorporate your library's reporting into their logging infrastructure.

Java Libraries, Logging Frameworks (2)

- One common solution:
  - Apache Commons Logging project
    - http://commons.apache.org/logging/
  - Provides a generic API that wraps other logging frameworks (e.g. Log4j and Java logging)
  - A bit too clever for its own good
    - Tries to use classloaders in clever ways
    - Can be *very painful* to use in some circumstances
  - A lot of big projects use commons logging
- Another solution: create your own wrapper!

Logging Framework Websites

- Apache Log4j
  - http://logging.apache.org
- Java Logging APIs
  - http://java.sun.com/j2se/1.5.0/docs/guide/logging
- Apache Commons Logging
  - http://commons.apache.org/logging/
  - http://www.qos.ch/logging/thinkAgain.jsp
    - “Think again before adopting the commons-logging API”

Preprocessors and Java

- Java doesn’t have a preprocessor.
- The good:
  - C/C++-style preprocessors add lots of issues
  - Facilitates binary compatibility of Java classes
- The bad:
  - Lose lots of flexibility to configure project sources
  - Generating Java code requires extra effort, purpose-built tools
- How do you compile out Java source code?

Conditional Compilation

- Can define “flag variables” in Java programs

```
    static final boolean TRACE_ENABLED = true;
    ...
    if (TRACE_ENABLED)
        logger.trace("Move list:  " + genMoveList());
```

  - Set **`TRACE_ENABLED = false`** to disable all trace output in program.
- **`javac`** *compiles out* the code if **`TRACE_ENABLED`** is false
  - **`final`**: variable can be set only once
  - **`static`**: available immediately after the class is loaded
  - A *literal* value is specified (**`true`** or **`false`**)
  - Compiler can easily tell that guarded code will *never* run
  - This is Java’s sole preprocessor-like feature

Conditional Compilation (2)

- What about this?

```
static final boolean isTraceEnabled() {
    return false;
}
...
if (isTraceEnabled())
    logger.trace("Move list:  " + genMoveList());
```

  - Java won’t compile out the trace statement here.
- Conditional compilation is performed in a *very specific* set of circumstances.
  - Should be used very rarely, too.

This Week’s Assignment

- Polish off the client UI behavior
  - Wait for a game to start on a worker thread, so client UI doesn’t lock up
  - Use dialogs to inform user of progress, errors, etc.
- Add logging to the server
  - Use Log4j to log when users start a game, when a game ends, when errors occur, etc.
- Next week:
  - Packaging up the Boggle program into a JAR file!