Swing, Part 3

Christopher Simpkins
chris.simpkins@gatech.edu

Java Projects

Today's example will involve many classes, so we'll begin by learning how to organize a multi-file (and multi-library) Java project

- the classpath and third-party libraries
- separating source and compiler output
- project directory layout
- packages

The Classpath

Just as your operating system shell looks in the `PATH` environment variable for executable files, JDK tools (such as `javac` and `java`) look in the `CLASSPATH` for Java classes. To specify a classpath:

- set an environment variable named `CLASSPATH`, or
- specify a classpath on a per-application basis by using the `-cp` switch. The classpath set with `-cp` overrides the `CLASSPATH` environment variable.

Pro tip: don't use the `CLASSPATH` environment variable. If it's already set, clear it with (on Windows):

```
C:> set CLASSPATH=
```

or (on Unix):

```
$ unset CLASSPATH
```

Classpath Elements

A classpath specification is a list of places to find `.class` files and other resources. Two kinds of elements in this list:

- directories in which to find `.class` files on the filesystem, or
- `.jar` files that contain archives of directory trees.

A **J**ava **ar**chive, or jar file, is a Zip-formatted archive of a directory tree containing `.class` files and other files.

- To create a JAR file: `jar cf jar-file input-file(s)`
- To view the contents of a JAR file `jar tf jar-file`
- To extract the contents of a JAR file: `jar xf jar-file` or `unzip jar-file`
- To extract specific files from a JAR file: `jar xf jar-file archived-file(s)`
- To run an application packaged as a JAR file (requires the Main-class manifest header): `java -jar app.jar`

See http://docs.oracle.com/javase/tutorial/deployment/jar/index.html for more on jar files.

Specifying a Classpath

To compile and run a program with compiler output (.class files) in the current directory and a library Jar file in the lib directory called util.jar, you'd specify the classpath like this:

```
$ ls -R # -R means recursive (show subdirectory listings)
MyProgram.java     AnotherClass.java

./lib:
util.jar
$ javac -cp .:lib/util.jar *.java # : separates classpath elements
$ java -cp .:lib/util.jar MyProgram # would be ; on Windows
```

Notice that you include the entire classpath in the -cp, which includes the current directory (. means "current directory").

Separating Source and Compiler Output

To reduce clutter, you can compile classes to another directory with `-d` option to `javac`

```
$ mkdir target
$ javac -d target HelloWorld.java
$ ls target/
HelloWorld.class
```

Specify classpath for an application with the `-cp` option to `java`.

```
$ java -cp ./target HelloWorld
Hello, world!
```

If you really want to keep your project's root directory clean (and you do), you can put your source code in another directory too, like `src`.

```
$ mkdir src
$ mv HelloWorld.java src/
$ javac -d ./target src/HelloWorld.java
$ java -cp ./target HelloWorld
Hello, world!
```

Project Directory Layout

Source Directories

- `src/` for Java source files
- `src/resources/` for resources that will go on the classpath, like image files

Third Party Libraries

- `lib/` for jar files

Output Directories

- `target/` for compiled Java .class files and resources copied from `src/resources/`

We'll use this simple layout, but the de-facto standard Java project directory layout for "professional" Java projects introduced by the Maven build tool can be found at http://maven.apache.org/guides/introduction/introduction-to-the-standard-

Organizing your Code in Packages

All professional Java projects organize their code in packages. The standard package naming scheme is to use reverse domain name, followed by project specific packages. For our CompanyGUI application we could use the package name

```
package edu.gatech.cs1331.companygui;
```

and source files would be located in a directory under your `src/` directory as follows

```
src/edu/gatech/cs1331/companygui/
```

And if you tell `javac` to put compiler output in `target/` then the compiled `.class` file would end up in:

```
target/edu/gatech/cs1331/companygui/
```

Aside from organization, packages ensure that names are unique. Using reverse domain names leverages the fact that internet domain names are non-ambiguous.

Swing Applications

- Inner classes
- Built-in dialogs
- Lists, ListModels and the MVC Pattern
- General Structure of Swing Hierarchy
- ScrollPanes and the Decorator Pattern
- Custom Dialog Boxes

We'll cover a lot today. Think of today's lecture as a guided tour through an example Swing application, Company GUI, which provides a GUI for the company classes we wrote a few weeks ago. After the tour you can look at this application for examples to guide your own Swing app development.

Inner Classes

We’ve defined our listeners as public top-level classes, but if they’re only used in one class they can be defined within the class that uses them:

```java
public class CompanyFrame extends JFrame {

    private class NewEmpListener implements ActionListener {
        public void actionPerformed(ActionEvent e) { ... }
    }
// ...
```

And instantiated where needed:

```java
JButton newEmpButton = new JButton("New Employee...");
newEmpButton.addActionListener(new NewEmpListener());
```

Anonymous Inner Classes

If an inner class is used only once, we can define it and instantiate an instance of it anonymously at the same time:

```java
newEmpButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("New button.  It was clicked.");
    }
});
```

- The syntax `new ActionListener() { ... }` means “instantiate an instance of a class that’s a subclass of `ActionListener`.”
- Required method definitions are given between `{ ... }`.

You’ll see this idiom a lot in Swing code.

Dialog Windows

A dialog window:

- is an independent subwindow that conveys information or gets input form the user.
- can be *modal*, meaning that the dialog window blocks its parent `Frame` while the dialog is visible, or *modeless* meaning it does not block its parent window.

In Swing:

- Every dialog is dependent on a Frame. When a dialog's parent Frame is destroyed, so is the dialog.
- Simple modal dialogs can be created easily with `JOptionPane`.
- Custom dialogs can be created by extending `JDialog`.

Simple Dialogs with `JOptionPane`

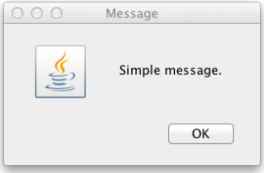
`JOptionPane` has several static methods that instantiate and display simple dialogs. The most common are (from the Java API):

- `showConfirmDialog` asks a confirming question, like yes/no/cancel.
- `showInputDialog` prompts prompts the user for input.
- `showMessageDialog` displays a message to the user.
- `showOptionDialog` is The Grand Unification of the above three.

For example, the line:

```
JOptionPane.showMessageDialog(null, "Simple message.");
```

displays the dialog:



Simple message.

JFileChooser

Swing provides dialog windows for common tasks like choosing colors and choosing files. Here's a general pattern for using `JFileChooser`:

```
// This is how you get the present working directory
File thisDir = new File(System.getProperty("user.dir"));
// Pass thisDir to JFileChooser's constructor to open
// the JFileChooser in the present working directory
JFileChooser chooser = new JFileChooser(thisDir);
// showOpenDialog() blocks until the user clicks ``OK'' or ``Cancel''
int returnVal = chooser.showOpenDialog(null);
// The return value from showOpenDialog() tells you how the user
// dismissed the dialog
if (returnVal == JFileChooser.APPROVE_OPTION) {
    File file = chooser.getSelectedFile();
    // do somethign with file ...
}
```

See CompanyGui.java for an example.

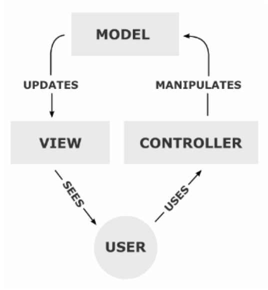
General Structure of Swing Programs

Typically a Swgin GUI program will have

- a “main” class (often in `Main.java`) that is responsible for handling command line arguments (if any), initializing the application configuration (perhaps form a properties file), and opening the application’s main frame
- a “main frame” that represents the application’s main GUI component, it’s “top level” or “home screen.”

In the Company GUI example CompanyGui.java is the main class and CompanyFrame.java is the main frame.

The Model-View-Controller Design Pattern



[1]

- The *model* contains the data that is displayed by the *view*
- The *view* displays the data from the *model* on screen
- The *controller* gets input from the user and manipulates the model

In Swing the view and controller are often combined.

[1] `http://en.wikipedia.org/wiki/File:MVC-Process.png`

JList, ListModel, and MVC

Our main application Frame, CompanyFrame takes a ListModel as an argument:

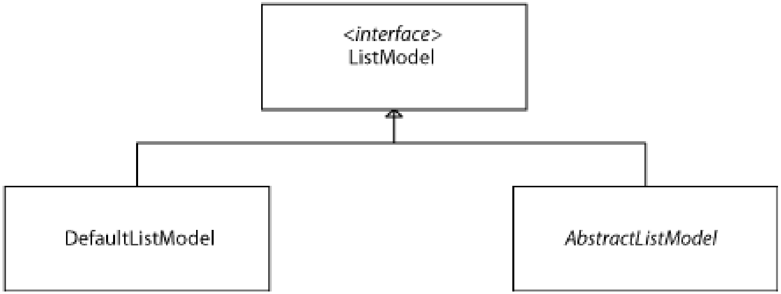
```
public CompanyFrame(ListModel employeeListModel) {
    // ...
    add(new JList(employeeListModel), BorderLayout.CENTER);
    // ...
}
```

And we build the ListModel before we show the main application frame:

```
Company company = new Company(file);
DefaultListModel lm = new DefaultListModel();
for (Employee e: company.getEmployees()) {
    lm.addElement(e);
}
CompanyFrame cf = new CompanyFrame(lm);
```

General Structure of Swing Hierarchy

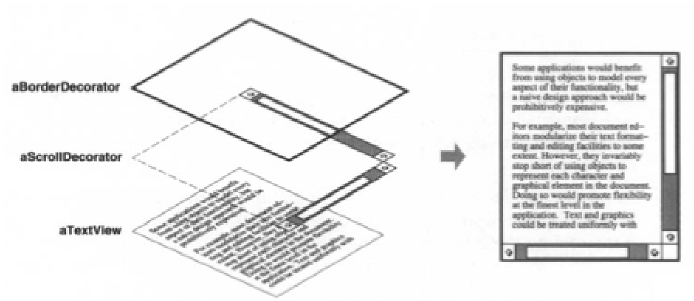
`ListModel` and its descendant classes are typical of the structure of many Swing classes:



- `ListModel` is the general interface
- `DefaultListModel` is a ready-to-use implementation for simple cases.
- `AbstractListModel` is a class that can be built upon for complex needs.

The Decorator Design Pattern

The Decorator Pattern[2]
Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.



Some applications would benefit from using objects to model every aspect of their functionality, but a naive design approach would be prohibitively expensive.

For example, most document editors modularize their text formatting and editing facilities to some extent. However, they invariably stop short of using objects to represent each character and graphical element in the document. Doing so would promote flexibility at the finest level in the application. Text and graphics could be treated uniformly with

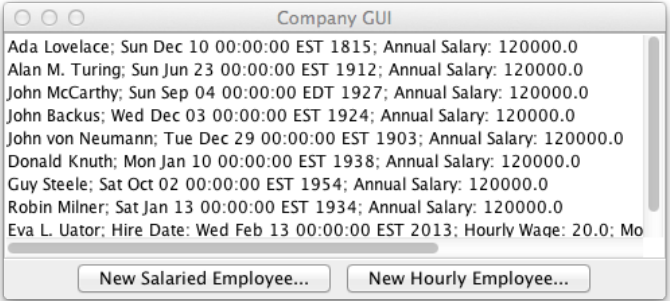
[2]Gamma, Helms, Johns, and Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994

JScrollPane and the Decorator Pattern

The Swing l ibrary provides a scrollbar decorator called `JScrollPane`. Using it is easy:

```
add(new JScrollPane(new JList(employeeListModel)),
    BorderLayout.CENTER);
```

By simply wrapping our `JList` in a `JScrollPane` the list will show horizontal and vertical scroll bars as needed.



```
Ada Lovelace; Sun Dec 10 00:00:00 EST 1815; Annual Salary: 120000.0
Alan M. Turing; Sun Jun 23 00:00:00 EST 1912; Annual Salary: 120000.0
John McCarthy; Sun Sep 04 00:00:00 EDT 1927; Annual Salary: 120000.0
John Backus; Wed Dec 03 00:00:00 EST 1924; Annual Salary: 120000.0
John von Neumann; Tue Dec 29 00:00:00 EST 1903; Annual Salary: 120000.0
Donald Knuth; Mon Jan 10 00:00:00 EST 1938; Annual Salary: 120000.0
Guy Steele; Sat Oct 02 00:00:00 EST 1954; Annual Salary: 120000.0
Robin Milner; Sat Jan 13 00:00:00 EST 1934; Annual Salary: 120000.0
Eva L. Uator; Hire Date: Wed Feb 13 00:00:00 EST 2013; Hourly Wage: 20.0; Mo
```

Custom Dialogs with `JDialog`

To create a custom dialog, extend `JDialog`. Custom dialogs require consideration of

- Modality - should the dialog be modal or modeless?
- Visual design - a dialog box is a visual component just like any other GUI component.
- UI design:
  - Make it clear how the dialog box fits into the overall GUI application (how it is launched, what happens if the user clicks OK or Cancel).
  - Make it clear what the user is expected to do with the dialog.
  - Give the user useful error messages if they do something wrong.

look at some examples in the Company GUI (particularly SalariedEmployeeDialog.java and HourlyEmployeeDialog.java.

Programming Exercise: Word Count GUI

Write a simple GUI for the word count program we wrote a few days ago.

- Write a main class that uses a `JFileChooser` to select the file.
    - Read the API docs and figure out how to restrict the `JFileChooser` to show only text files.
- Update your `WordCount` class so that it's a class that takes a file in the constructor, and has a single method `getWordCounts()` that returns a `Map<String, Integer>` that maps from words to their counts.
    - Bonus: make your `WordCount` class `Iterable`.
- Display the word counts in a `JList` with scroll bars.
    - Even better: use a `JTable` to display the words and associated counts.
- Include a single button at the bottom of your application's main frame that exits the program when clicked.