CSE 1325 - Object-Oriented Programming

GUI Programming

Alex Dillhoff

University of Texas at Arlington

GUI Programming in Java

Java's original GUI subsystem is called the **Abstract Window Toolkit (AWT)**.

It included basic controls, windows, dialog, and event handling.

GUI Programming in Java

The visual components in AWT were limited because they are translated into platform-specific components.

Programs written for Windows would use visual elements native to Windows. This ensure that the apps looked familiar to users.

GUI Programming in Java

Since visual elements were dependent on the specific platform, GUI applications suffered from inconsistency.

The **Swing** framework was introduced to alleviate this problem.

GUI Programming in Java

Swing is built on top of AWT, so components from there may still be used.

Swing components are called lightweight. They are written completely in Java and do not depend on any platform-specific implementations.

GUI Programming in Java

With Swing, developers have more control over exactly how their applications should look and feel.

It also comes with a guarantee that applications will behave consistently regardless of the platform.

GUI Programming in Java

Swing supports what is called *pluggable look and feel* (PLAF).

This means developers can separate the look and feel of an object from the logic of *what* the component does.

Components and Containers

Swing GUIs consist of two main visual components: **components** and **containers**.

Containers are actually subclasses of components, but they represent two important concepts.

Components and Containers

Most Swing components derive from the `JComponent` class.

`https://docs.oracle.com/en/java/javase/16/docs/api/java.desktop/javax/swing/JComponent.html`

Components and Containers

A container holds multiple view components such as button, text fields, and labels.

There are two types of containers in Swing:

1. Lightweight containers
2. Top-level containers

Lightweight Containers

Lightweight containers refer to components that manage groups or other components.

These are not top-level containers. A common example is the `JPanel` class.

Top-level Containers

Top-level containers are the only Swing components that do not inherit from `JComponent`.

Instead, they inherit from AWT's `Component` class. They are considered **heavyweight**.

Top-level Containers

Top-level containers in Swing include:

- `JFrame`
- `JApplet`
- `JWindow`
- `JDialog`

Top-level Containers

These containers must be at the very top-level of the visual hierarchy.

It is not recommended to create GUI applications that have top-level containers as child elements of another top-level container.

Top-level Containers

A top-level container has a set of **panes**.

The first is `JRootPane`. This pane manages the other panes as well as an optional menu bar.

Top-level Containers

The other panes are

- Glass pane - Covers other panes, handles mouse events.
- Content pane - Where most visual elements will are placed.
- Layered pane - Controls the depth of visual elements.

Top-level Containers

Example: `BasicFrame`

The Event Thread

In the previous example, the GUI application was launched on an event thread via

```
SwingUtilities.invokeLater(Runnable object)
```

This is necessary since Swing components are *event-driven*.

The Event Thread

Swing components react to events triggered by user interaction as well as other external sources.

These are handled on a separate event thread as to not interfere with the main application thread.

Event Handling

Since Swing components respond to and trigger events, it is important to understand how to handle such events.

The event handlers were originally introduced with AWT. They are still useful with Swing components.

Event Handling

Events are comprised of **sources** and **listeners**.

A source can be something like a visual component or some other part of your program.

A listener is a handler that catches and responds to the corresponding events.

Event Handling

There are many sources available and they all derive from `EventObject`.

`https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/EventObject.html`

Event Handling

These sources can register listeners.

Any registered listener will be sent the corresponding event when it occurs.

These are implemented as interfaces.

Event Handling

Example: `ButtonEventDemo`

Event Handling

The previous example showed that we can define a listener in many different ways.

We will explore different kinds of listeners in future examples.

Layouts

GUI programs can contain only a few components or hundreds of components.

Organizing these components should be done in a way that can easily adapt to new resolutions, resizes, and more.

Layouts

The Java API provides several layouts that can be used to manage visual components.

Every `Container` has a layout associated with it.

You can disable the layout for a container by calling `setLayout(null);`

FlowLayout

`FlowLayout` organizes visual components from top to bottom, left to right.

Components can be arranged using the following constraints:

- `FlowLayout.LEFT`
- `FlowLayout.CENTER`
- `FlowLayout.RIGHT`
- `FlowLayout.LEADING`
- `FlowLayout.TRAILING`

FlowLayout

Example: `FlowLayoutDemo`

BorderLayout

The default layout for a swing GUI is the `BorderLayout`.

This layout consists of a center region surrounded by 4 border regions.

BorderLayout

The regions can be selected using the following constants:

- `BorderLayout.CENTER`
- `BorderLayout.SOUTH`
- `BorderLayout.EAST`
- `BorderLayout.NORTH`
- `BorderLayout.WEST`

BorderLayout

Example: `BorderLayoutDemo`

GridLayout

`GridLayout` presents components in a 2D grid.

This class has a constructor to specify the number of rows and columns in the grid.

GridLayout

Example: `GridLayoutDemo`

CardLayout

The `CardLayout` may be used when you need to organize multiple layouts.

Each layout can be represented on a *card* with a given index.

CardLayout

The indices are useful for swapping out layouts that contain groups of visual elements.

Example: `CardLayoutDemo`

GridBagLayout

The last layout to choose from is `GridBagLayout`.

This allows for fine-grained control that utilizes relative positioning and sizing.

GridBagLayout

Like `GridLayout`, this layout consists of rows and columns.

The difference is that `GridBagLayout` can have rows with a different number of columns.

GridBagLayout

The sections in this layout are defined by *constraints*.

Understanding how to work with constraints is the key to using this layout properly.

GridBagLayout

Example: `GridBagLayoutDemo`

Additional Components

Not every Swing GUI will use every component available in the Java API.

It is not necessary to have an intimate knowledge of the API either.

Additional Components

Spending time looking through the documentation and implementing individual examples is the best way to learn how to implement each component.

JFileChooser

One particularly useful task is that of locating a file on a system.

Your program may then perform any necessary operations with the file.

JFileChooser

Swing provides `JFileChooser` for navigating the local file system and selecting a particular file.

There are many configuration options available from filtering file types to previewing selected images.

JFileChooser

Example: `JFileChooserDemo`

Creating Modal Dialogs

The Java API provides `JDialog` for creating modal dialog views.

This class is the most general form of a dialog view and can be fully customized.

Creating Modal Dialogs

Interaction between the dialog and the rest of the program is no different than other visual components.

Creating Modal Dialogs

Example: `JDialogDemo`

Working with Images

Working with images is very common for GUI programs.

There are two common approaches to displaying images.

The first involves overriding `paintComponent`. The other simply creates a `JLabel` object with an icon.

Working with Images

The first approach involves creating a custom component, such as `JPanel`, and override `paintComponent`.

This provides access to a `Graphics` object needed to use Java's rendering methods.

Working with Images

Example: `ImageViewDemo`

Working with Images

The second method is as simple as loading an image and assigning it as the icon object for a `JLabel` object.

```
BufferedImage img = ...
JLabel imgLabel = new JLabel(new ImageIcon(img));
```

Model-View-Controller

A common and useful design pattern for GUI development is called **Model-View-Controller (MVC)**.

MVC adheres to 3 core properties of GUI components:

1. How is the component rendered?
2. How does the component react to the user?
3. What state information does the component contain?

Model-View-Controller

MVC maintains a separation of the model (functionality related to the data), the view (rendering and layout), and a controller.

The controller is in charge of facilitating interactions between the interface components and the underlying data models.

Model-View-Controller

The model refers to anything related to data used for the program.

This includes the data class objects as well as any interactions with data retrieval.

In the TableTopRPG project, a `Player` would be a model component.

Model-View-Controller

View components contain code directly related to visual rendering and interaction between the user and the GUI.

These components should never directly depend on model implementations to perform retrieve the data for rendering.

Model-View-Controller

The controller binds the model and views together.

It may respond to user interactions with the GUI, triggering an update to the model component.

It then sends the raw model data to the view components for rendering.

Model-View-Controller

Example: `CharacterCreator`