Chapter

6

GUI Programming

CHAPTER AUTHORS

Ang Ming You

Ching Sieh Yuan

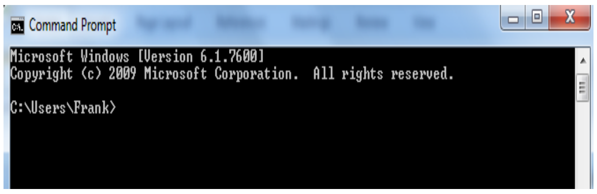
Francis Tam

Pua Xuan Zhan

CONTENTS

1 Introduction

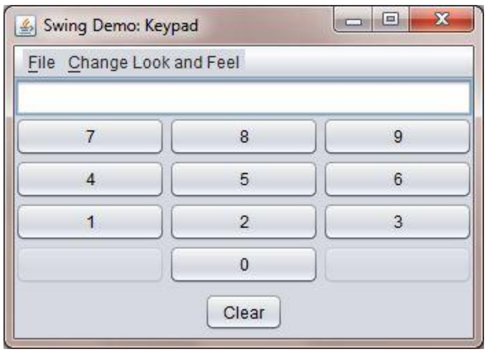
Graphic User Interface (GUI) also pronounced as GOO-ee) is a software interface that the user interacts with using a pointing device, such as a mouse. For example, when you browse Internet you are looking at the GUI of the web browser. The alternative to (and predecessor of) GUIs is the text-based Command Line Interface (CLI) which is much less user-friendly for layman user. Given below are screenshots of a CLI and GUI of a software:



```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Frank>
```

Command line interface: Command Prompt



Simple GUI of a number pad

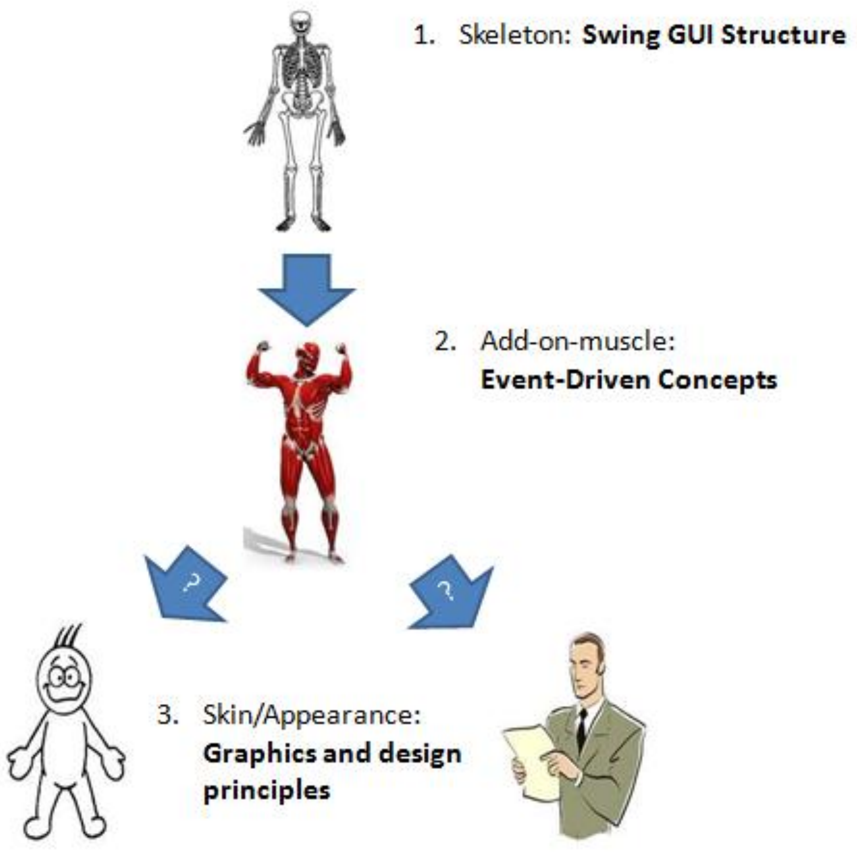
Especially in the market today, a good GUI is essential for marketability of a product; having an efficient and functional GUI is not enough, it has to be tasteful and intuitive for users to interact with. This requires good handling of inputs and outputs with pleasing and comprehensive layout structures for the user

<u>How can this chapter help you?</u>

- Learn basics of Java Swing
- Understand the Swing framework
- Know the components available for GUI interaction
- Understand the basics of handling inputs
- Learn how to use layouts and design appearances of GUI
- Go in-depth to the programming aspects of events-handling
- Get advice on design techniques used today

For teaching the basics of GUI programming, we are going to use Swing, a Java GUI widget[1] toolkit. As most of programmers have experienced coding in Java, programming in Swing would be easy to pick up. Since Java is created by Sun Microsystems, Swing is also stable and reliable. With its simplicity to create basic GUI templates, Swing is easy for beginners to pick up basic GUI programming. Thus, the structure of the content in this chapter is built on the Swing framework. A simple analogy of the content to follow...

[1] Widget is a component of a user interface that operates in a particular way.


1. Skeleton: Swing GUI Structure
2. Add-on-muscle: Event-Driven Concepts
?
?
3. Skin/Appearance: Graphics and design principles

[2]

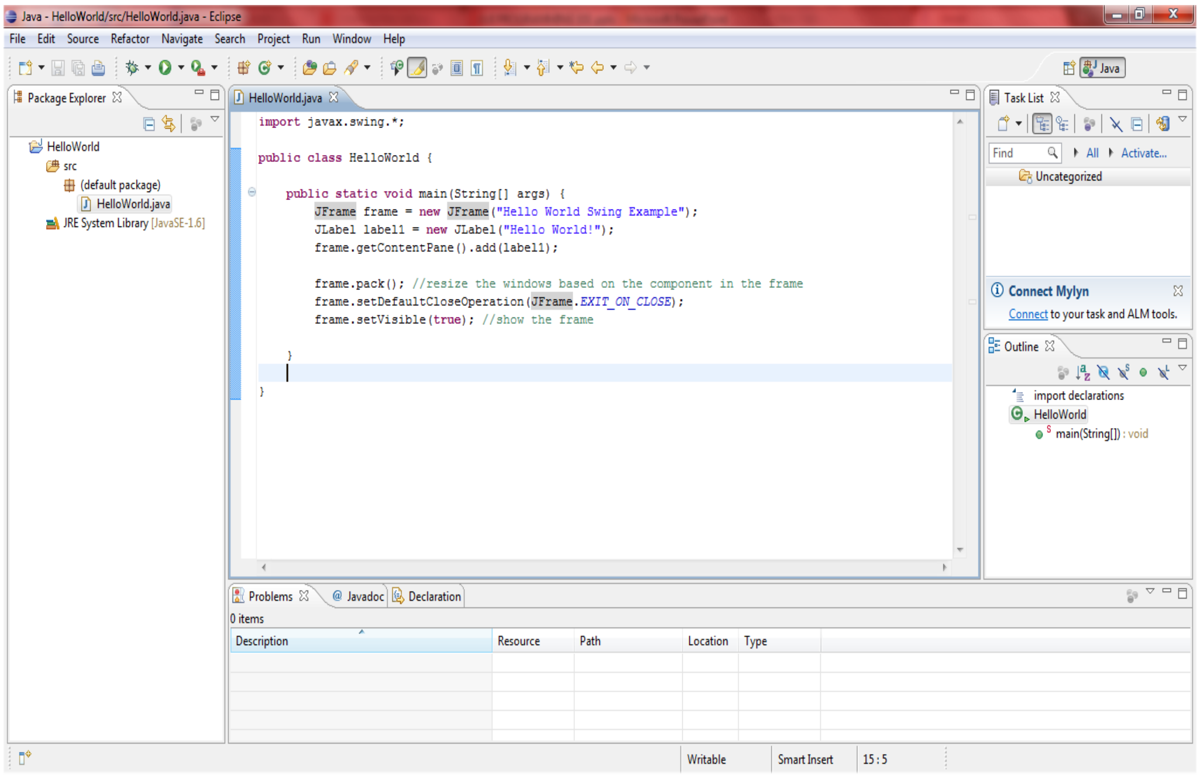
Swing GUI Structure: Similar to the study of the structure of a human skeleton, we study the basic blocks used to build a Swing GUI application. The blocks consist of *containers* and *components*; we will explore the roles of *containers* and *components* and how they are used.

Behaviour: With the skeleton of the GUI done, we need the muscle to make it work or react to input. The muscle here refers to *event*-handling or *event-driven* programming. Imagine *event-handling* as making the GUI react to inputs (i.e. mouse clicks or keyboard inputs).

Appearance: With *event-handling*, a basic GUI application is complete! However, this is not enough to create a good GUI application; we also need to make the GUI presentable for making the software more marketable, just like the reason why we suit up in formal attire in preparation for an interview. Thus, we will be showing some graphical aspects of the Swing framework; layouts, look and feel and some designing concepts.

Here is what to expect for this chapter; *Figures 1a and 1b* below show an example of a basic GUI application written in Java using the Eclipse IDE:

[2] Skeleton: blog.lib.umn.edu. Muscle figure: store.cgsociety.org



```java
import javax.swing.*;

public class HelloWorld {

    public static void main(String[] args) {
        JFrame frame = new JFrame("Hello World Swing Example");
        JLabel label1 = new JLabel("Hello World!");
        frame.getContentPane().add(label1);

        frame.pack(); //resize the windows based on the component in the frame
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true); //show the frame

    }

}
```

Figure 1a: Coding a basic GUI application with Eclipse

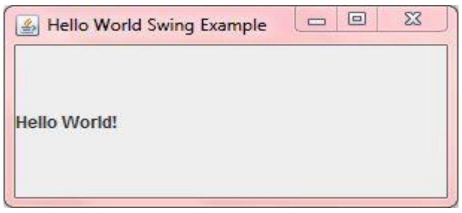
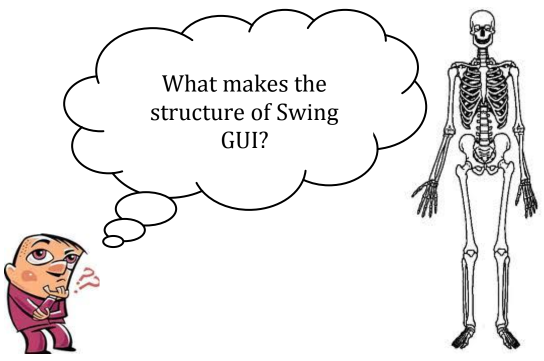


Figure 1b: Basic GUI application (from the code above)

2 GUI STRUCTURE



What makes the structure of a Swing GUI? The structure of the application is made by *containers* and *components*. Containers are on screen windows that contain user interactive components or other sub-containers. **Top-level containers** are the containers existing on the desktop directly and the **non top-level containers** are containers that are shown **within the top-level container**. Components are the user interactive objects of the GUI that could be added into containers.

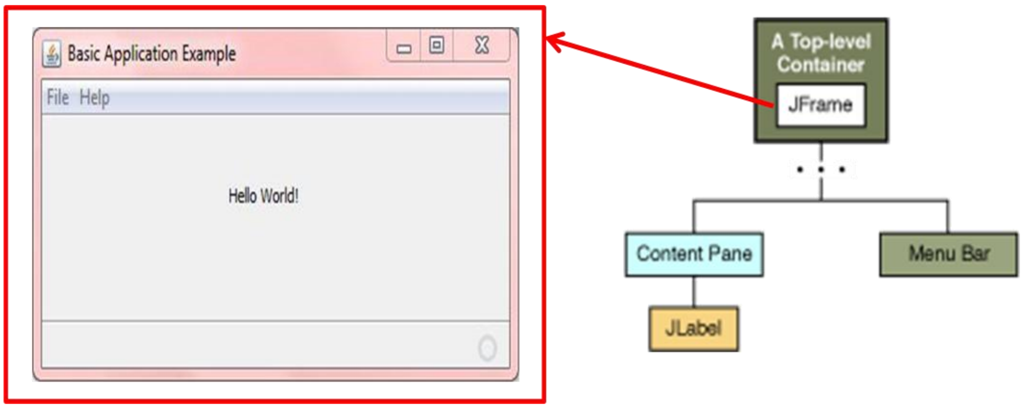


Figure 2.1: JFrame as a top-level container[3]

Figure 2.1 shows an application of the hierarchy in Swing. Each program that uses Swing components has at least one top-level container, which is the root of the containment hierarchy. As a rule, a standalone application with a Swing-based GUI has at least one containment hierarchy with a Jframe as its root.

More top-level containers:

Besides JFrame, here are 2 more examples: JDialog is used for creating a custom dialog boxes; JApplet compared to Applet (from Abstract Window Toolkit, AWT) gives more support for Swing component architecture[4].

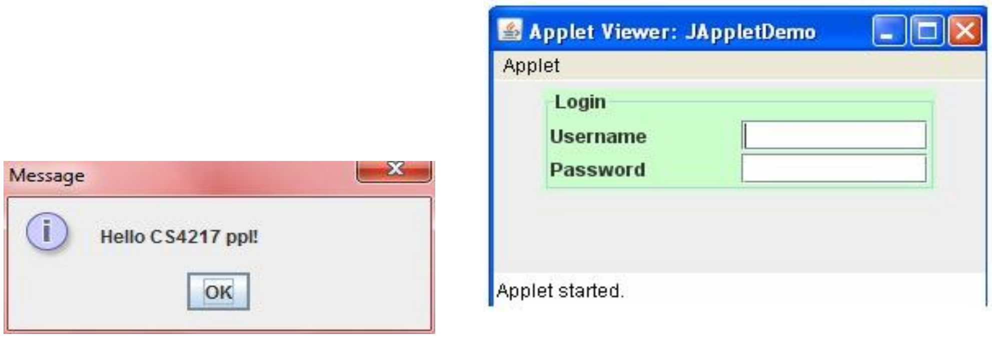


Figure 2.2: JDialog (left) and JApplet (right)

General Purpose containers:

[3] Container hierarchy diagram from http://download.oracle.com/javase/tutorial/uiswing/components/toplevel.html

[4] Swing component architecture (SMA) is an alternative to the Model- View- Controller (MVC) pattern. More information on SMA found here : http://java.sun.com/products/jfc/tsc/articles/architecture/

General purpose containers are also classified as a non top-level or *sub-containers* with examples shown below. (i.e. JScrollPane gives a scrollable view of a large or growable component; JSplitPane displays two components whose relative size can be manipulated by the user.) These can only be added to a top-level container.

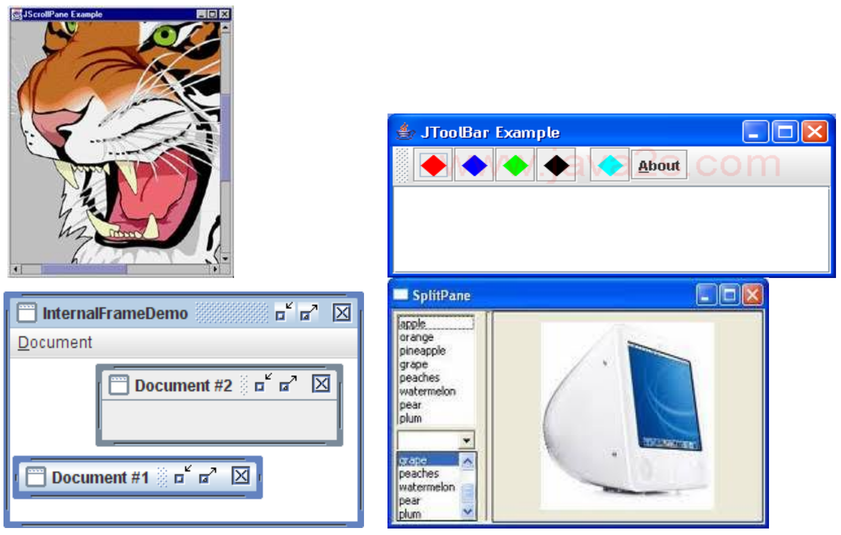


Figure 2.3: JScrollPane (top left, from java.sun.com) and JToolBar (top right, from java2s.com), JInternalFrame (lower left, from codexion.com) and JSplitPane (lower right, from shrike.depaul.edu)

Components and Sub-containers:

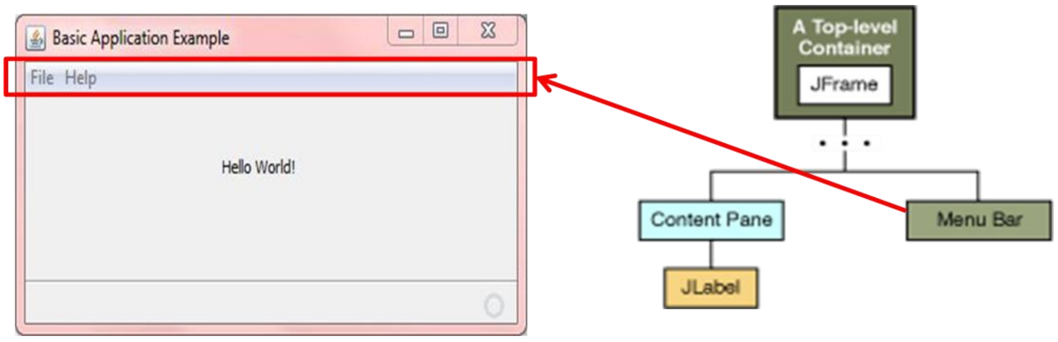


Figure 2.4a: Menu Bar added to JFrame as a component

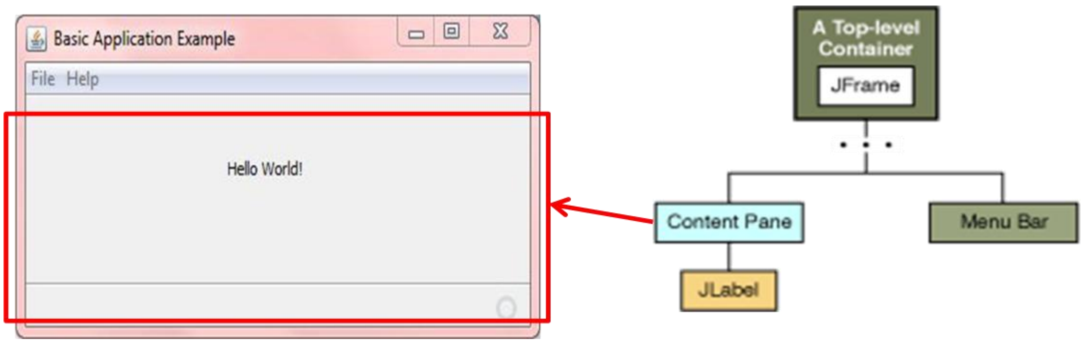


Figure 2.4b: Content Pane of the JFrame is added with a JLabel ("Hello World!)

Components available:

Here is a list of user interactive components the programmer can call upon to add to the content pane of a sub or top-level container. The content pane contains the visible components of the container's GUI.

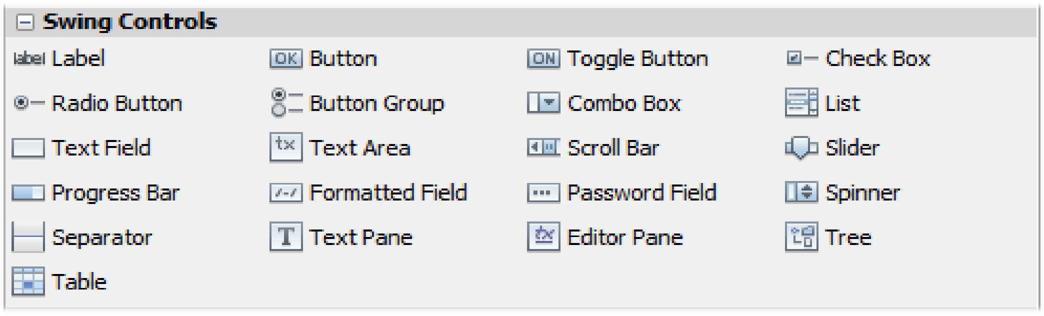
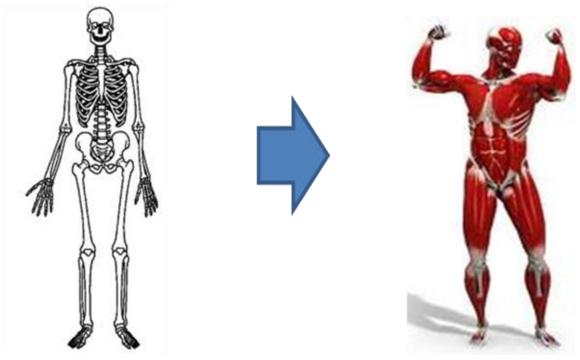


Figure 2.5: JComponent's list of components in Swing

In summary, a Swing GUI structure is made of a (1) **top-level container**, (2) with **sub-containers** and/or (3) **components** added onto the top-level container.

3 GUI Behaviour



How do we get a GUI to respond to interactions? Just like muscle is required to move a skeleton, we now need to handle inputs from the user by programming interactions into the GUI. The programming of interactions to the GUI is called *event-handling* or known as *event-driven programming*. An example is shown in figure 4.5:

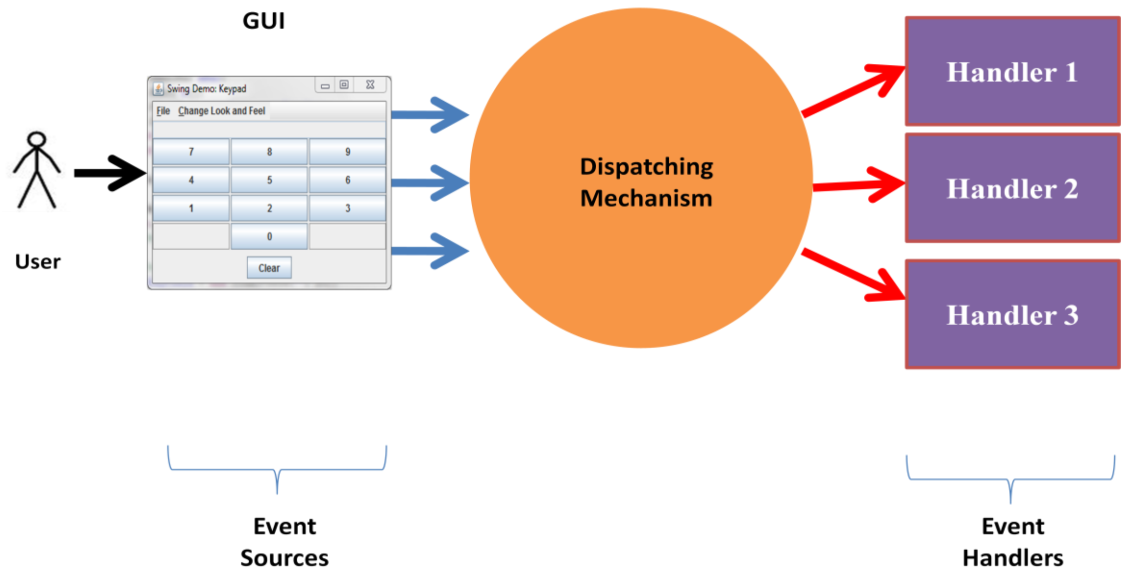


Figure 3.1: Event-handling example

In figure 4.5, we see a few new terms: *event sources*, *event handlers* and *dispatching mechanism*. **Event Sources** creates events whenever it is used (i.e. clicking a button on the GUI), the **dispatching mechanism** queues incoming events while determining its *event type*, event is sent to the **event handler** that deals with events of that type (i.e. events from button "1" to Handler 1, events from button "2" to Hander 2, events from button "3" to Handler 3). Also, more than one event handler can react to an event from an event source; such as having events from buttons "1" and "2" to be handled by Handler 1. In Swing, the equivalent to an event source would be a JButton component, while event handlers are known as listeners.

However, how should we control the routing of multiple events? (i.e. let an event be handled by handlers 1 & 2) A solution is to use ***observer pattern*** which involves 2 entities; *subjects* and *observers*. **Observers** register to subjects they are interested in and subjects will keep a list of dependent observers with their addresses. If a **subject** has a change of state, all dependent observers to the subject will be notified automatically (refer to figure 3.2). This is also known as the *publish/subscribe pattern*, where the subjects are *publishers* and the observers are *subscribers*. **Publishers** do not need to know **subscribers** to send out notifications and subscribers also do not need to know publishers to subscribe and receive notifications.

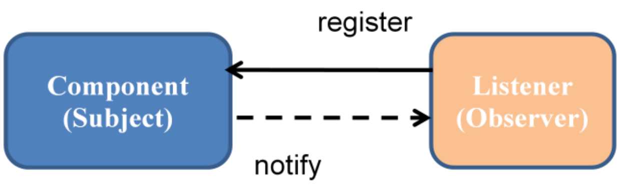
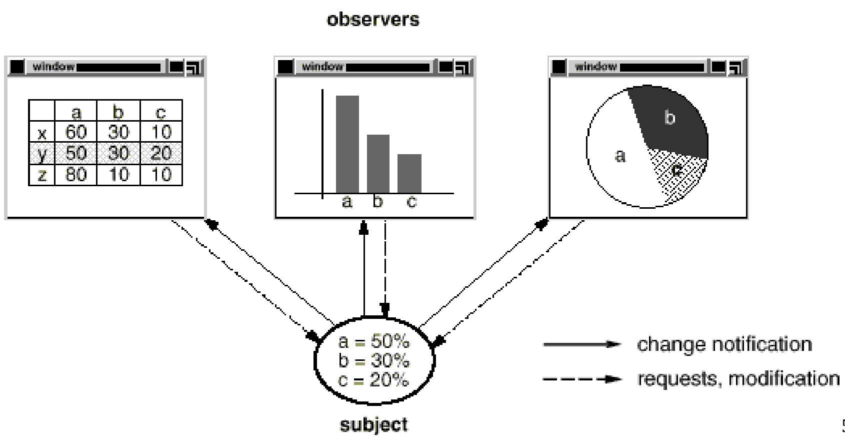


Figure 3.2: Subject-Observer relationship in Swing



[5]

Figure 3.3: Observer pattern example

Figure 4.6 shows an example of using the observer pattern; the windows (observers) are all using the same data (subject). If the values in data change, the 3 windows registered to data would be notified and change accordingly; though how each window will change is unknown to the data.

Using the Observer Pattern results in good decoupling between the subjects and the observers; removing subjects or observers would not affect the overall mechanism of the GUI which makes updating or editing of the code much easier for the programmer.

So how should we implement the Observer Pattern in our GUI? Here is how?

(1) **Create a component**: We create an event source; such as a JButton.
(2) **Setting up the listener**: We also need to prepare tasks for listeners to perform when a particular event type[6] is received.
(3) **Bind a listener to the component:** Add a listener to the component, so that the task defined for the component is performed when the component produces an event.

```
public class Keypad extends JFrame{

        …
        Keypad(){
                …
                button8 = new JButton("8");
                button8.addActionListener(new ButtonListener());
        }

        …
}

public class ButtonListener implements ActionListener{

        public void actionPerformed(ActionEvent event) {
                // Do checking here
                …
        }
}
```

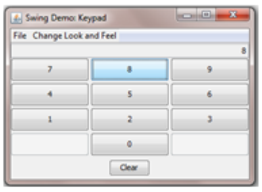


Figure 4.1: Binding a listener to a component

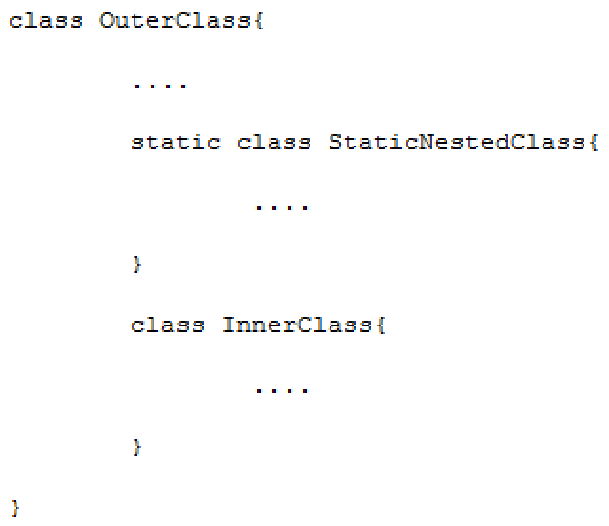
Figure 4.1 gives a simple method of binding an *action listener*[7] to a component (button8) by creating a class for the components (Keypad) and another class that stores the methods to

[5] Observer pattern example from http://www.research.ibm.com/designpatterns/example.htm
[6] *Event type:* please refer to ***Error! Reference source not found.***

perform during event occurrences (ButtonListener). We can do this for the other buttons and listeners too. We can see that the components do not know who the listeners that will handle their events are, the listeners also do not know who the event sources of the event type are until a check is performed. This is how we achieve the **Error! Reference source not found.**; where the subjects (components) and observers (listeners) are oblivious to each other.

There are also other methods of implementing event driven programming, depending on the needs and requirements of the programmer. We will touch on 2 such topics:

- **Nested Class vs. Inner Class**
- **Anonymous Inner Class vs. Inner Class**
- **Nested Class vs. Inner Class:**



```
class OuterClass{

        ....

        static class StaticNestedClass{

                ....

        }

        class InnerClass{

                ....

        }

}
```
[8]

A *nested class* is a static class enveloped by another class, while an *inner class* is a non-static class enveloped by another class. This idea is very similar to static and non-static elements of a class. The nested class has access to all elements of the enveloping class while the inner class requires an object reference to the enveloping class. Since most listeners and components are specific to a certain model (i.e. number keys to a number pad of Keypad GUI), putting the listeners with the modal is logical and uses less packages; making the listener class less prone to data corruption. The inner class can extend to other members (methods and variables) in the enveloping class as long as they are are non-static.

Anonymous Inner Class vs. Inner Class:

On the other hand, Java does not support multiple inheritance. Using the nested class method is too restrictive if we are considering a GUI with multiple functionalities. We can use *anonymous inner classes*, classes with no names, to define listeners. Why bother since there are inner classes? If the component-to-listener is 1-to-1, anonymous inner class is used to create a no reusable implementation of an abstract class. However, if the components-to-listeners' ratio is many-to-many, there will be a lot of duplicate code. In this case, the inner class would be recommended.

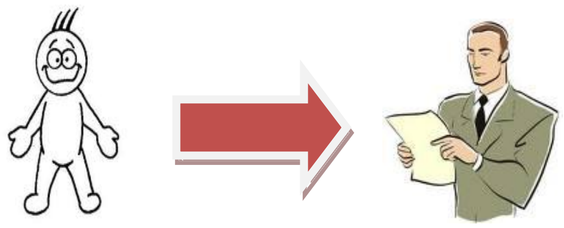
[7] There are other listener interfaces beside *action listeners*, refer to this website for more information: http://download.oracle.com/javase/tutorial/uiswing/events/eventsandcomponents.html
[8] Nested class picture taken from http://users.ecs.soton.ac.uk/dem/teaching/proginjava/s10.html

```
public class Keypad implements ActionListener{
        ...
        Keypad(){
            ...
            button7 = new JButton("7");
            button8 = new Jbutton("8");

            button7.addActionListener(new ActionListener(){
                    public void actionPerformed(ActionEvent event) {
                            textField.setText("7");
                    }
            });
            button8.addActionListener(new ActionListener(){
                    public void actionPerformed(ActionEvent event) {
                            textField.setText("8");
                    }
            });
        }
        ...
}
```

Figure 4.2: Anonymous Inner Class example

4 GUI APPEARANCE



After creating the GUI structure and adding in interactions, how do we make it look good? In this section, we will go through *layouts* and *look-and-feel* to provide more designing options for the GUI. This is a rather big topic to cover, so we will attempt to give a general concept of Swing GUI appearances in 2 sections:

- **Layouts**
- **Look & Feel**

Layouts: These are layout strategies used to position components and sub-containers. These are strategies are controlled by a layout manager object that implements the *LayoutManager* interface. The object determines and has the final say in the size and position of the components in a container. A couple of layouts available are:

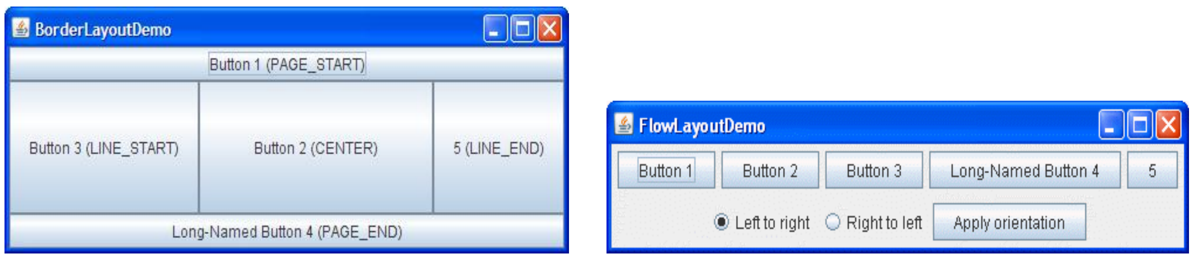
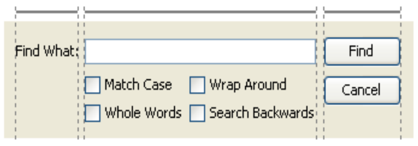


Figure 5.1a: BorderLayout(left) and FlowLayout (right)

Commonly used layouts:

- *BorderLayout* creates a container, resizing its components to fit five regions: north, south, east, west. Each region can only have 1 component.
- *FlowLayout* arranges components in a left-to-right flow, much like lines of text in a paragraph.
- *GroupLayout* allows grouping of components together in a hierarchal manner. In figure 2.6b, this is known as sequential group layout.
- *SpringLayout* lays out the children of its associated container according to a set of constraints.



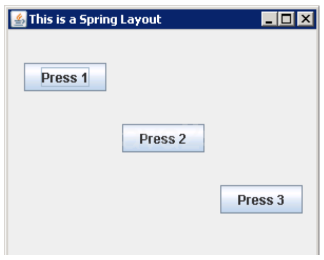


Figure 5.1b: GroupLayout (left) and SpringLayout (right)

Layouts exist to provide more consistency and organization for the GUI application. For beginners, these pre-defined layouts are useful as minimal coding is required. For more information about layouts please check these sites:

http://www.formdev.com/jformdesigner/doc/layouts/

http://download.oracle.com/javase/tutorial/uiswing/layout/using.html

Look & Feel (L &F): It refers to the general appearance of a user interface; it acts like the skin of a GUI. It has the ability to change the whole appearance of an application without programmer having to change each individual component or container. However, this also means that different L&Fs may have different font sizes or sizing schemes for containers and components. Below, figure 2.7 highlights such an issue. L&F is good for providing a GUI with **consistency** and **organization** since all components are similar in appearance and positioning.

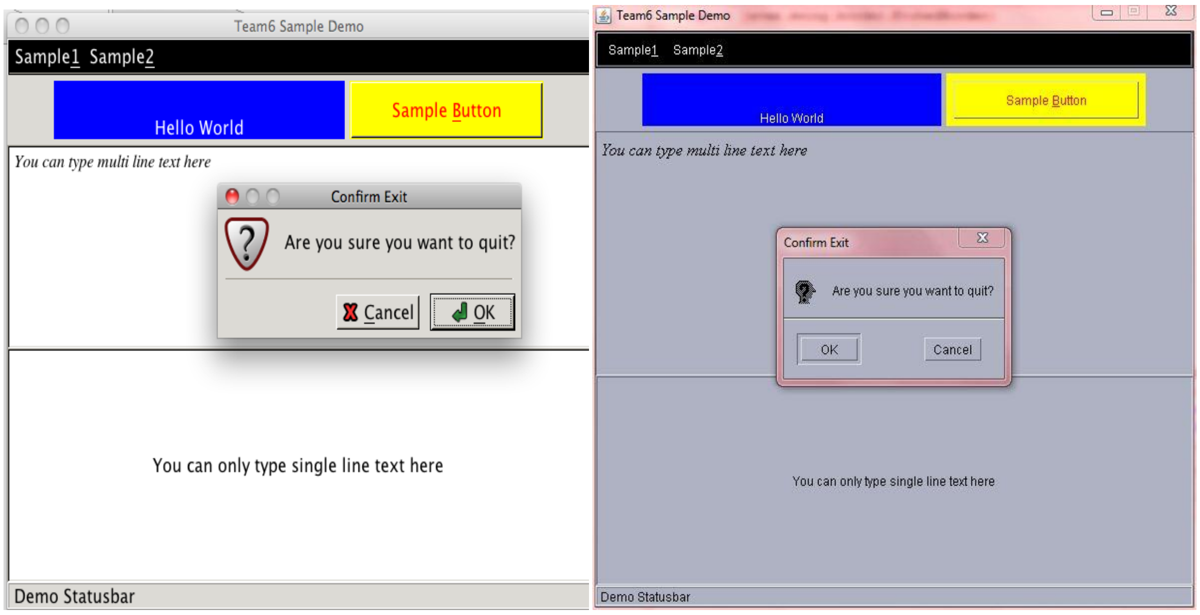

Team6 Sample Demo
Sample1 Sample2
Hello World
Sample Button
You can type multi line text here
Confirm Exit
Are you sure you want to quit?
Cancel
OK
You can only type single line text here
Demo Statusbar
Team6 Sample Demo
Sample1 Sample2
Hello World
Sample Button
You can type multi line text here
Confirm Exit
Are you sure you want to quit?
OK
Cancel
You can only type single line text here
Demo Statusbar

Figure 5.2: Mac L&F versus Win7 L&F

Looking at figure 2.7, the biggest difference between the Mac L&F and the Win7 L&F is the font size; with the Mac L&F having larger default font size. Even the sizes of the pop out windows are different. Thus, a programmer also needs to know the differences when shifting L&F from one operation system to another, so that alignment of buttons or text will not be out of place.

L&F's main purpose is to provide beautification to the application, act as branding for a company and it is ease of use to change the whole outlook of your application. On an interesting note, this tends to be a hot legal issue because some software companies claim that competitors who copy the L&F of their products are infringing on their copyright protection.

More on methods to use L&F in Java can be found here:

http://download.oracle.com/javase/6/docs/api/javax/swing/LookAndFeel.html

5 Concurrency in GUIs

Have you experienced a GUI that stops functioning and becomes unresponsive? This is known as the "freezing" GUI, where it stops taking in inputs from the user and stops working until the task at hand is complete. The task that requires a lot of computation is known as *intensive*. If this is the case, we need a way to allow the GUI to continue working while running the intensive task or tasks. This is done by *multi-threading*.

In multi-threading, we introduce threads to take care of certain tasks. For instance, in Swing we have an *initial thread* that executes the GUI, an *event dispatch thread* that handles light weight tasks which do not require much computation and the intensive tasks are delegated to *worker threads.* Using this method, the GUI would "freeze" less.

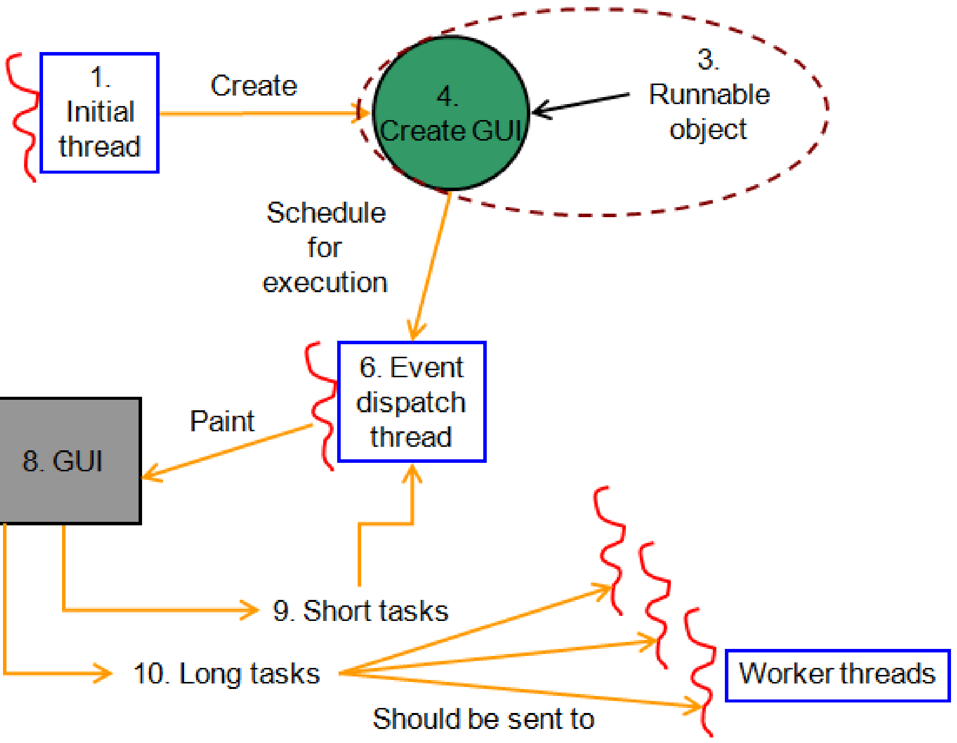


Figure 6.1: Multi-threading in Swing

6 GUI DESIGN GUIDELINES

Before going into the principles of designing a good GUI, we need to take note of the common mistakes made by programmers.

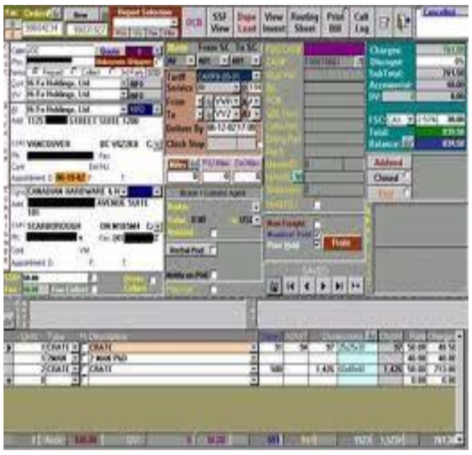
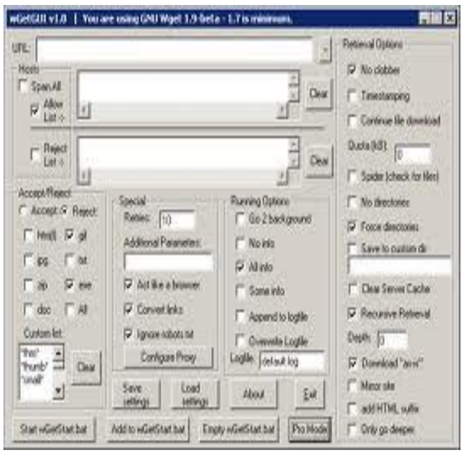
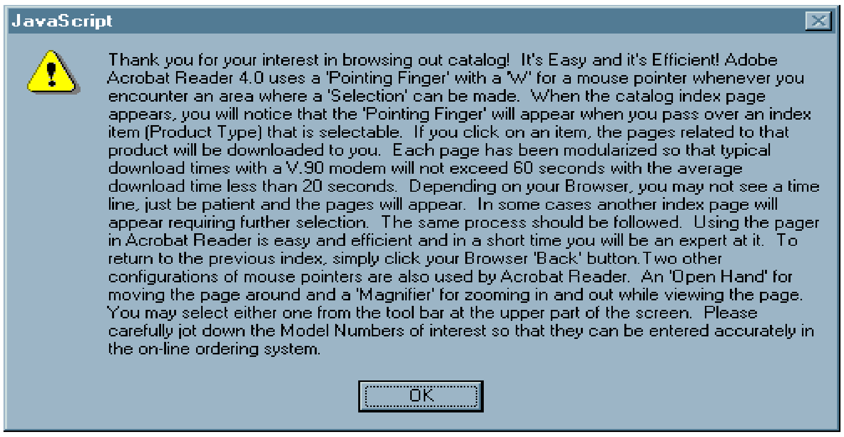



Figure 3.1a: Bad interface (left, from designinginteractive.com and right, from blog.vinova.sg)

Problem: **Too many features**.
Solution: **Design for clarity.**
Having a clear interface GUI design helps the use to focus on the basic functions of the GUI. As we look at the bad interface examples in figure 3.1a, the cluttering of functions make it very difficult to tell what functions we want to use at first glance. Besides the basic functions, advanced or extra functions should be kept hidden until user requests for them



JavaScript

Thank you for your interest in browsing out catalog! It's Easy and it's Efficient! Adobe Acrobat Reader 4.0 uses a 'Pointing Finger' with a 'W' for a mouse pointer whenever you encounter an area where a 'Selection' can be made. When the catalog index page appears, you will notice that the 'Pointing Finger' will appear when you pass over an index item (Product Type) that is selectable. If you click on an item, the pages related to that product will be downloaded to you. Each page has been modularized so that typical download times with a V.90 modem will not exceed 60 seconds with the average download time less than 20 seconds. Depending on your Browser, you may not see a time line, just be patient and the pages will appear. In some cases another index page will appear requiring further selection. The same process should be followed. Using the pager in Acrobat Reader is easy and efficient and in a short time you will be an expert at it. To return to the previous index, simply click your Browser 'Back' button.Two other configurations of mouse pointers are also used by Acrobat Reader. An 'Open Hand' for moving the page around and a 'Magnifier' for zooming in and out while viewing the page. You may select either one from the tool bar at the upper part of the screen. Please carefully jot down the Model Numbers of interest so that they can be entered accurately in the on-line ordering system.

Figure 3.1b: Bad feedback

Problem: **Too much text.**
Solution: **Keep text clear.**
From figure 3.1b, it is difficult for the user to accept first glance on what this window is about. Being concise and keeping feedback to about one line is difficult, but it helps the user to understand the situation better. This is similar to the principle above.

There are many more principles to follow, for more information, please approach this website: http://www.classicsys.com/css06/cfm/article.cfm?articleid=20

7 Conclusion

What have learnt in this topic?

(1) We have gone through the basics of containers and components in the Swing framework.
The adding of the components to a GUI can only be seen as making a basic GUI skeleton, where the components do not react to user interaction (i.e. clicking a button)
(2) To add muscle to skeleton, we use event driven programming to handle interactions to the GUI by introducing listeners that react to events triggered from the event source.
(3) We have gone through some basic design issues and concepts to when designing a GUI.

For more information related to this chapter, or to see tutorials of coding Swing GUIs, please visit our website at: https://sites.google.com/site/cs4217jan2011team6/

We hope that the journey so far has been an eventful one for the readers, especially the beginners. We will like to leave the readers with this quote to figure out:

A picture is worth a thousand words. An interface is worth a thousand pictures.

- ***Ben Shneiderman, 2003***

8 References

Design Principles:

http://www.classicsys.com/css06/cfm/article.cfm?articleid=20

Event Driven Programming:

http://eventdrivenpgm.sourceforge.net/event_driven_programming.pdf

Components and Look-and-Feel:
http://download.oracle.com/javase/tutorial/uiswing/components/index.html

Observer Pattern:

http://www.research.ibm.com/designpatterns/example.htm

http://www.javaworld.com/javaworld/javaqa/2001-05/04-qa-0525-observer.html

Concurrency in Swing:

http://download.oracle.com/javase/tutorial/uiswing/concurrency/index.html