

The Java Learning Kit: Chapter 7 Graphical User Interfaces

Lesson 7.1 – Graphical User Interface Programming

Lesson 7.2 – Graphical User Interface Programming in Java

Lab 7A – Programming Example: Swing Components in a GridLayout

Lab 7B – Programming Example: FlowLayout GUI

Lab 7C – Programming Example: BorderLayout GUI

Lesson 7.3 – Setting Java GUI Component Properties

Lesson 7.4 – Using Image Files in a GUI

Lab 7D – Programming Example: Pick a Penguin

Lesson 7.5 – Using Panels to Organize a GUI

The Java Learning Kit: Chapter 7 – Graphical User Interfaces

Copyright 2015 by C. Herbert, all rights reserved.

Last edited January, 2015 by C. Herbert

This document is a chapter from a draft of the Java Learning Kit, written by Charles Herbert, with editorial input from Craig Nelson, Christopher Quinones, Matthew Staley, and Daphne Herbert. It is available free of charge for students in Computer Science courses at Community College of Philadelphia during the Spring 2015 semester.

This material is protected by United States and international copyright law and may not be reproduced, distributed, transmitted, displayed, published or broadcast without the prior written permission of the copyright holder. You may not alter or remove any trademark, copyright or other notice from copies of the material.

Contents

Chapter 7 Learning Outcomes	3
Lesson 7.1 Graphical User Interface Programming	4
Event-Driven Software	4
Inheritance	4
How Object Inheritance Works in Java	6
CheckPoint 7.1	6
Lesson 7.2 Graphical User Interface Programming in Java	7
Swing GUI Components	8
AWT GUI Containers	8
AWT Layout Managers	9
Creating a GUI with AWT and Swing Objects	10
CheckPoint 7.2	11
Lab 7A – Programming Example: Swing Components in a GridLayout	12
Lab 7B – Programming Example: FlowLayout GUI	14
Lab 7C – Programming Example: BorderLayout GUI	16
Lesson 7.3 Setting Java GUI Component Properties	18
Setting Java GUI Component Colors	18
Fonts and Typefaces	20
Setting Java GUI Component Fonts	21
Setting Java GUI Component Borders	22
CheckPoint 7.3	25
Lesson 7.4 Using Image Files in a GUI	26
Images on JLabels	26
Images on JButtons	26
Lab 7D – Programming Example: Pick a Penguin	27
Adding Tool Tips to Swing Components	28
CheckPoint 7.4	28
Lesson 7.5 Using Panels to Organize a GUI	28
Chapter Questions	32
Chapter Exercises	33

The Java Learning Kit: Chapter 7

Graphical User Interfaces

This chapter introduces the creation of graphical user interfaces in Java. It covers creation and display of a GUI, but not activating a GUI, which is covered in subsequent chapters. Lessons 1 describes GUI programming in general and two concepts important in GUI programming: event-driven programming and object inheritance.

Lesson 2 discusses programming in Java to create a GUI using *AWT* and *Swing* objects.

Lessons 3 and 4 show how to affect the appearance of a GUI by manipulating component properties and using image files with components.

Lesson 5 presents the use of *JPanels* to better organize GUI components.

Chapter 7 Learning Outcomes

Upon completion of this chapter students should be able to:

- describe the concept of a Graphical User Interface (GUI) and how it is related to event-driven programming.
- describe the concept of object inheritance and why it is important in object-oriented programming.
- describe what the Abstract Window Toolkit (AWT) and Swing APIs are and how they are used for GUI programming.
- list and describe several containers and layout managers used for GUI programming and how containers and layout managers are related to one another.
- list and describe the function of several commonly used components used for GUI programming.
- describe how to manipulate GUI components by changing component color, font, and border.
- describe how to use images with certain GUI components.
- describe how to use *JPanels* to better organize components in a GUI.
- build GUIs in Java using AWT and Swing objects that demonstrate the use of containers, layout managers, and components.

Lesson 7.1 Graphical User Interface Programming

A **graphical user interface (GUI)** is an event-driven combination of hardware and software that allows a person to communicate with a computer using a pointing device to interact with icons, buttons, and other graphical elements on a computer screen. The user interface for the *Microsoft Windows* operating system is an example of a GUI. Traditional GUIs use a mouse to control the screen pointer, while many newer systems use touch screen technology, in which a person's hand or a stylus can be used to point to objects on the screen by physically touching them. Most smart phones use a touch screen GUI.

A graphical user interface has a coordinated combination of graphics software to display interface components and event-driven software to respond when a user interacts with those components.

Event-Driven Software

Recall from Chapter 2 that **event-driven software** “listens” for and responds to system events, such as someone clicking a button on the screen, or pressing a certain key on a keyboard. Event driven software involves an event listener, an event trigger, and an event handler. An **event listener** is combination of software and hardware sensors that can tell when a specific condition, called an **event trigger**, occurs. When an event listener detects the occurrence of an event trigger, it transfers control of the system to an **event handler**, which is a program that tells the computer what to do when the event trigger occurs.

The *JOptionPane* pop-up dialog windows described in Chapter 2 are examples of GUIs. In this chapter, we will begin to create more elaborate graphical user interfaces using GUI objects from Java's *Abstract Window Toolkit (AWT)* and *Swing API*; including windows, frames, textboxes, buttons, checkboxes, and so on. The Abstract Window Toolkit and the Swing API are part of the **Java Foundation Classes (JFC)**, a set of classes for GUI programming with objects to build portable graphical user interfaces that will work across multiple platforms.

In this chapter, we will focus on drawing GUI interfaces, then in the next few chapters we learn more about classes and event-driven programming so we can create our own reusable, active graphical user interfaces.

Inheritance is also an important object-oriented programming technique used in GUI programming, so we will start with a short discussion of inheritance.

Inheritance

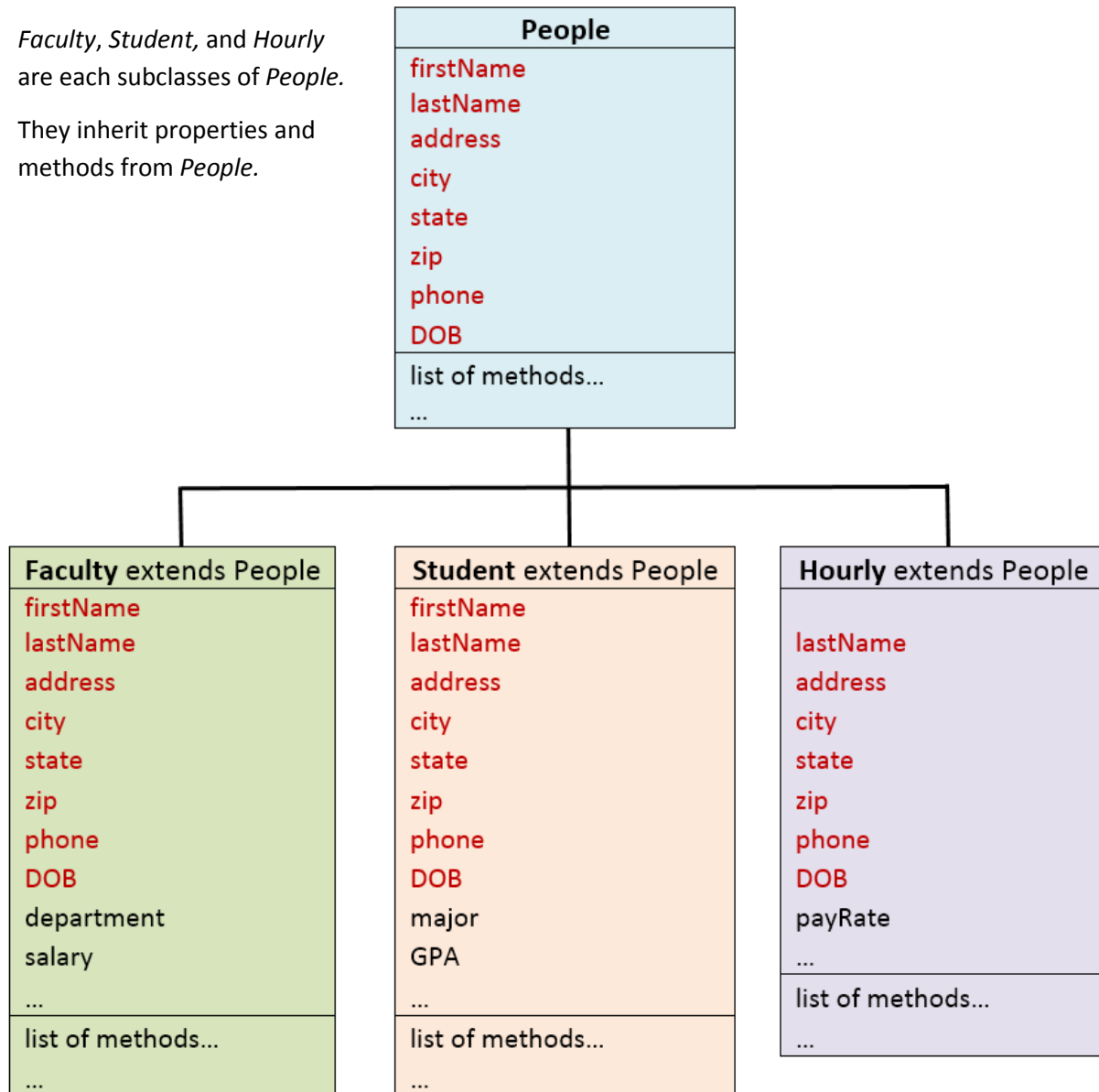
Java is an object-oriented programming language that supports object inheritance. Remember, an object is a collection of properties along with methods to manipulate those properties. **Object inheritance**, also referred to simply as **inheritance**, is the ability in a programming language to define a **parent class** of objects with certain properties and methods, then declare a **child class** of the object, which automatically has the properties and methods of the parent class, along with any newly declared properties and methods of its own. The parent class is often called a **superclass**, and the child class is often called a **subclass**.

Inheritance is used when we have several classes of objects that are similar to each other, with many, but not all, properties and methods in common. A college or university information system, for example, needs to keep track of the people associated with the university – it might have a class of

objects for faculty, a class for students, a class for hourly employees, and so on. They are all people that have names, addresses, phone numbers, and so on, but each class is in some way different from the others – for example, faculty belong to a department and have a salary, students have a major and a GPA, hourly employees have an hourly pay rate.

Faculty, Student, and Hourly
are each subclasses of *People*.

They inherit properties and
methods from *People*.



We can establish a parent class named *People*, with the properties common to all people – *name*, *address*, *phone number*, *birth date*, and a few other properties. Then we can establish subclasses of the *People* class with specialized information – such as a *Faculty* class with *department* and *salary*, a *Student* class with *GPA* and *major*, and an *HourlyEmployee* class with a *payRate*. All of the subclasses will inherit *name*, *address*, *phone number*, *birth date*, etc. from the *People* class. The diagram above shows us this.

How Object Inheritance Works in Java

In Java, the keyword **extends** is used to indicate inheritance. The code on this page shows class declarations in Java that match the diagram on the last page. The declarations for the *People*, *Student*, *Faculty*, and *Hourly* classes are shown. The *extends People* phrase means that *Student*, *Faculty*, and *Hourly* each automatically inherit all of the properties and methods from *People*. They also each have properties and methods of their own. The inherited properties are shown in red on the diagram above.

```
class People {  
    String firstName;  
    String lastName;  
    String address;  
    String city;  
    String state;  
    String zip;  
    String phone;  
    ... // and so on, methods are not shown  
} // end class People
```

```
class Faculty extends People {  
    String department;  
    double salary;  
    ... // and so on on, methods are not shown  
} // end class Faculty
```

```
class Student extends People {  
    double GPA;  
    String major;  
    ... // and so on, methods are not shown  
} // end class Student
```

```
class Hourly extends People {  
    double payRate;  
    ... // and so on on, methods are not shown  
} // end class Hourly
```

We will learn more about inheritance after we learn to create classes of our own. For now, we simply need to know what inheritance is, because the objects used for graphical user interfaces are arranged in a hierarchy of classes that inherit properties and methods from parent classes. We will need to use the *extends* clause when creating some elements of a GUI that are sub-classes of exiting GUI object classes.

Checkpoint 7.1

1. What is a Graphical User Interface?
2. Describe how event-driven programming works.
3. What are the Java Foundation Classes?
4. Describe the concept of object inheritance.
5. Describe how subclasses can be established in Java that inherit properties and methods from a superclass named *Animal*.

Lesson 7.2 Graphical User Interface Programming in Java

Graphical user interface programming in Java beyond simple JOptionPane pop-up windows is most often done using the **Abstract Window Toolkit (AWT)** containers and **Swing** components. AWT is a package with objects for creating and organizing user interfaces, as well as for painting graphics and images. Swing is actually a collection of more than a dozen public packages with many classes of objects for GUI components, such as buttons, checkboxes, and so on.

Swing components and the AWT are now part of the *Java Foundation Classes (JFC)*, included as part of the standard *Java with NetBeans* download and installation discussed in Appendix A. If you are using the *Java SE Development Kit 7* (or higher) with *NetBeans* then you have access to the JFC simply by including the necessary import statements in your code.

(Note: Oracle has a detailed tutorial for building graphical interfaces with the JFC classes using Swing components online at: <http://docs.oracle.com/javase/tutorial/uiswing/index.html> , but you may need to finish the next few chapters in this book to be ready to use it.)

As much as possible, AWT and Swing objects are designed to look the same across all user interfaces, although there may be some variation from one platform to another. Keep in mind that the user interfaces we see today on many devices – from screens on the global network of automated teller machines to handheld devices such as tablets, cell phones and the newest digital combination wristwatch/phone/TV/browser – are created using Java. According to Oracle, there are now more than 7 billion devices worldwide with Java user interfaces. Shown here is the incoming call interface on an early model of the Java-enabled Samsung Galaxy Gear watch and Bluetooth smartphone. The watch has an 800 Mhz single core processor.



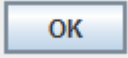
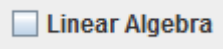
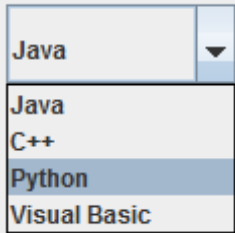
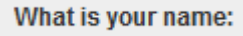
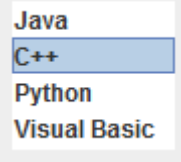
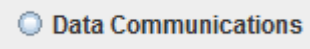

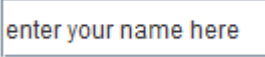
We need three different kinds of objects to create a basic GUI:

- **GUI components** (also known as GUI widgets) such as buttons, text boxes, etc. that provide the functionality of the GUI.
- a **container** to hold the GUI components.
- a **layout manager** to organize the components in the container.

The remainder of this section briefly discusses Swing components that are used to build a GUI, AWT containers to hold the components, and AWT layout managers to organize the components within the container. Several labs follow that show how to use components and layout managers in a JFrame container, followed by sections that discuss manipulating the appearance of components by manipulating their properties, as well as placing images on components from image files. The last section in the chapter shows how to use JPanel containers within JFrames to better organize GUIs.

Swing GUI Components

A GUI **component** is an object with a graphical representation that can be displayed on the screen and methods to interact with the user. We will use basic Swing components to create a GUI. There are dozens of components available for use in Java-based GUIs. We will use just a few:

Swing GUI object	Example	Description
JButton		an implementation of a "push" button.
JCheckBox		an implementation of a check box -- an item that can be selected or deselected, and which displays its state to the user.
JComboBox		a component that combines a button or text field with a selectable drop-down list. The list comes from an array, such as an array of Strings.
JLabel		a display area for a short text string, an image, or both.
JList		a component that displays a list of objects and allows the user to select one or more of them. The list comes from an array, such as an array of Strings.
JRadioButton		an implementation of a radio button – a single item that can be selected or deselected as part of a set, and which displays its state to the user.
JSlider		a component that lets the user graphically select a value by sliding a knob or handle within a bounded interval. It may be horizontal or vertical.
TextField		a component that allows the editing and entry of a single line of text.

AWT GUI Containers

We must have a container of some sort to hold GUI components.

An AWT **Container** is a component that can contain other AWT components. AWT has a *Container* class, with a hierarchy of increasingly specialized subclasses, such as:

- **Window** — a subclass of the Container class. Windows have a window area and a location on the display screen, but no title, border, etc.
- **Frame** — is a subclass of the Window class. Frames have a title and border properties.
- **JFrame** — is a subclass of the Frame class, with properties and methods support for the JFC architecture and Swing components.

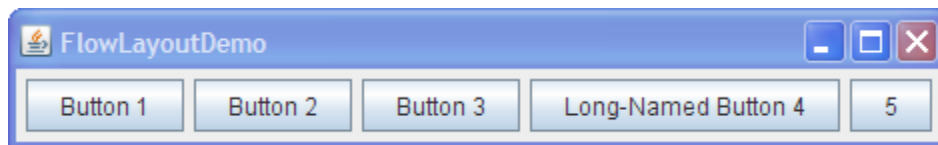
So, a JFrame is a Window with a title and a border, and which supports the use of the JFC Architecture and Swing components. We will use JFrames for our GUI programming. The JFrame windows we create will contain Swing components.

JPanel is another subclass class of Container, which may be used inside a Window, Frame, or JFrame to group components in part of the Window, Frame or JFrame. We will see how to use JPanels inside JFrames to better organize a GUI.

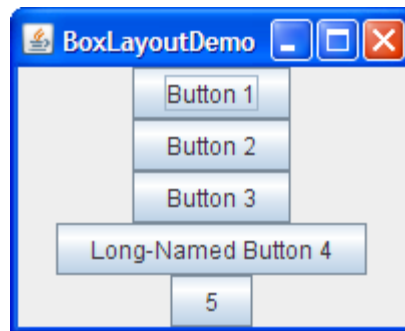
AWT Layout Managers

A **layout manager** organizes components in a GUI. There are many layout managers in the AWT class, but just a few are commonly used for basic GUI programming. The examples included in the list below show examples of JFrame GUIs created using the most popular layout managers:

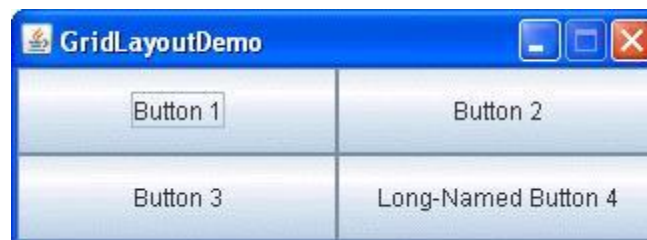
- **FlowLayout manager** is the default layout manager. Items are displayed in a row in order across the container, wrapping around to new rows if the set of components is wider than the container.



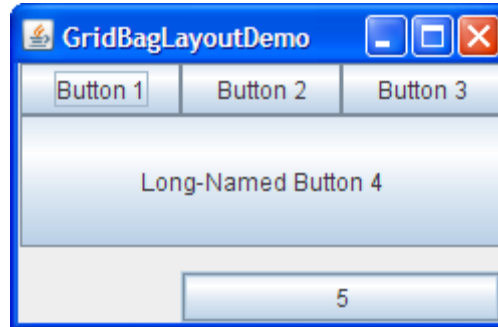
- **BoxLayout manager** puts components in a single row or column. A similar effect can be achieved using FlowLayout manager, but BoxLayout manager forces one row or column.



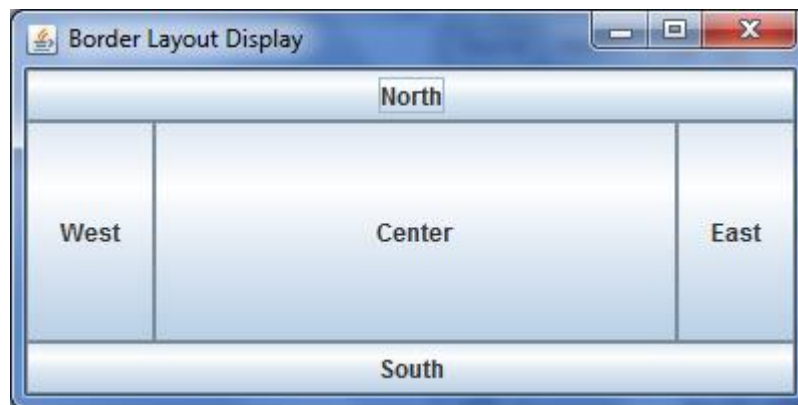
- **GridLayout manager** make components the same size in a uniform grid of rows and columns



- **GridBagLayout manager** is similar to *GridLayout manager*, but cells can be merged, rows can have different heights, and columns can have different widths. It is more flexible than the *GridLayout manager*, but more tedious to use. Panels within a *gridLayout* are more common than using a **GridBagLayout**.



- **BorderLayout manager** places components in five different zones of the container – NORTH, SOUTH, EAST, WEST, and CENTER. Constants to identify each of these zones are defined as part of the class.



JavaFX and *NetBeans* can be used to generate and build more complex GUIs using other layout managers specialized for use with those tools, but for now we will stick with a few simple layout managers and write the code ourselves.

Creating a GUI with AWT and Swing Objects

There are six steps in the process of creating a GUI with AWT and Swing objects in Java:

1. instantiate a container object.
2. instantiate a layout manager for the container and assign it to the container.
3. create any components for the interface.
4. add the components to the container.
5. set the properties for the container, such as size, location and exit behavior.
6. activate the container to make it visible.

We can create a one-time GUI as an instance of an existing container class, or we can create a new class for a reusable GUI by extending an existing container class. In either case, we can customize the GUI to meet our needs, but a one-time GUI will only work within the current software, while a new class for our GUI will allow us to create an instance of the GUI with a few lines of code whenever we need one.

In the long run, it is usually best to define most GUI's by extending JFrame to create a reusable GUI. This will save us from having to redefine the GUI every time we need to use it. This is really how object-oriented programming is intended to be used – to create re-useable code and make large programming projects more manageable.

In this chapter we will create one-time GUI's that are JFrame objects. Our purpose is to learn how GUI components and containers work. After we learn more about writing our own classes, we will create our own GUI classes. At that point, it will be simple to convert our one-time code into reusable objects.

We will start with a simple one-time GUI that shows us several different GUI components. We will create the GUI and show it on the screen. Later, when we learn more about events, we can link the GUI components to event handlers to make the GUI function as desired.

Creating a JFrame GUI

A JFrame is a specialized window with a title and border, and which allows us to use Swing components in the window. It is the most common container for creating GUIs using Java. In general, whenever we need to create a GUI using AWT and Swing components, we can:

1. create a JFrame object or a new object that extends JFrame.
2. create a layout for the JFrame and assign it to the JFrame.
3. create any components we wish to use.
4. add the components to the JFrame.
5. set the properties for the JFrame, such as size, location and exit behavior.
6. activate the JFrame to make it visible.

Checkpoint 7.2

1. List and briefly describe the role of each of the three different kinds of objects used to create a basic GUI.
2. List and briefly describe several commonly used Swing components used for creating a GUI.
3. List and briefly describe four AWT containers used for creating a GUI. Which AWT container is often used inside another container to group components?
4. List and briefly describe five AWT layout managers used to organize a GUI.
5. List and briefly describe the steps in the process of creating a GUI with Swing and AWT objects. What is the difference between creating a one-time GUI and a reusable GUI?

Lab 7A – Programming Example: Swing Components in a GridLayout

In this example, we wish to have a program that creates a JFrame showing us the Swing GUI components described in the last section --JButton, JCheckBox, JComboBox, JLabel, JList, JRadioButton, JSlider, and JTextField.

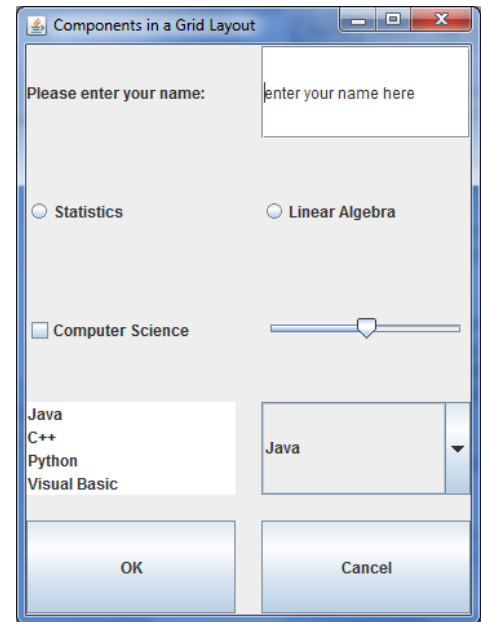
We will use a GridLayout object to organize our JFrame into 5 rows with 2 columns. We can also specify the horizontal and vertical gaps between cells in the grid.

Our example will have

- a JLabel and a JTextField in row one
- two JRadioButtons in row two
- a JCheckBox and an unlabeled JSlider in row three
- a JList and a JComboBox in row four
- and two JButton in row five

Our GUI is shown on the right. Notice that some components, such as the textbox and the buttons, fill the grid cells.

Here is a description of a software to create a sample one-time GUI using a GridLayout.



```
import AWT.* and Swing.*
```

```
create a JFrame object named myJFrame to hold our components
```

```
create a new a GridLayout object named myLayout to hold our components
it should have 5 rows x 2 columns, with horizontal and vertical gaps = 20
```

```
assign mylayout to be the layout for MyJFrame
```

```
create the components for our GUI
```

- a JLabel with text "Please enter your name:"
- a JTextField field with text "enter your name here"
- two JRadioButtons – one for "Statistics" one for "Linear Algebra"
- aJCheckBox – "Computer Science"
- a Jslider – just a default slider, no labels
- a String array with a list of programming languages
- a JList – based on the language array
- a JComboBox – based on the language array
- twoJButtons – one for "OK" and one for "Cancel"

```
add the components to myJFrame in the correct order
```

```
set the properties for myJFrame – title, size, location and exit behavior
```

```
activate the GUI – make it visible
```

Here is the code to create the sample GridLayout GUI described above. This NetBeans Project is in the files for chapter 7.

```
/*
 * ComponentSamples.java
 * CSCI 111 Fall 2013
 * last edited October 18, 2013 by C. Herbert
 * This program shows examples of some Swing components
 * in a JFrame using a GridLayout
 *
 * It is an example of a one-time GUI, not defined as a class
 * No events have been added to the GUI, it only shows components
 */

package componentsamples;
import javax.swing.*.*;
import java.awt.*.*;

public class ComponentSamples {

    public static void main(String[] args) {

        // create a frame to hold our components
        JFrame myJFrame = new JFrame();

        // create a new a grid layout for the frame - 5 rows x 2 cols, gaps=20
        GridLayout myLayout = new GridLayout(5,2);
        myLayout.setHgap(20);
        myLayout.setVgap(20);

        // assign myLayout to be the layout for MyJFrame
        myJFrame.setLayout(myLayout);

        // Create a label with text "Please enter your name:"
        JLabel jlName = new JLabel("Please enter your name:");

        // Create a text field with text "enter your name here"
        JTextField jtfName = new JTextField("enter your name here");

        // Create a radio button for Statistics
        JRadioButton jrbStat = new JRadioButton("Statistics");

        // Create a radio button for Linear Algebra
        JRadioButton jrbLinear = new JRadioButton("Linear Algebra");

        // Create a check box for Computer Science
        JCheckBox jcbCSCI = new JCheckBox("Computer Science");

        // Create a default slider -- it has no labels
        JSlider jlsSample = new JSlider();

        // create a String array for the list box and combo box
        String[] languages = new String[] {"Java", "C++", "Python", "Visual Basic"};

        // Create a list using the language array
        JList jlLanguages = new JList(languages);
    }
}
```

continued on next page

continued from previous page

```
// Create a combo box using the language array
JComboBox jcombLanguages = new JComboBox(languages);

// Create a button with text OK
JButton jbtOK = new JButton("OK");

// Create a button with text Cancel
JButton jbtCancel = new JButton("Cancel");

// add components to MyJFrame (in order)

myJFrame.add(jlName); // Add the name label
myJFrame.add(jtfName); // Add the text field to get the name

myJFrame.add(jrbStat); // Add the Statistics radio button
myJFrame.add(jrbLinear); // Add the Linear Algebra radio button

myJFrame.add(jcbCSCI); // Add the Computer Science check box
myJFrame.add(jlsSample); // Add the sample default slider

myJFrame.add(jlLanguages); // Add the languages list
myJFrame.add(jcombLanguages); // Add the languages combobox

myJFrame.add(jbtOK); // Add the OK button
myJFrame.add(jbtCancel); // Add the Cancel button

// set the title, size, location and exit behavior for the JFrame
myJFrame.setTitle("Grid Layout Demo");
myJFrame.setSize(360, 480);
myJFrame.setLocation(200, 100);
myJFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// make the frame visible (activate the frame)
myJFrame.setVisible(true);

} // end main()
} // end class
```

Lab 7B –Programming Example: FlowLayout GUI

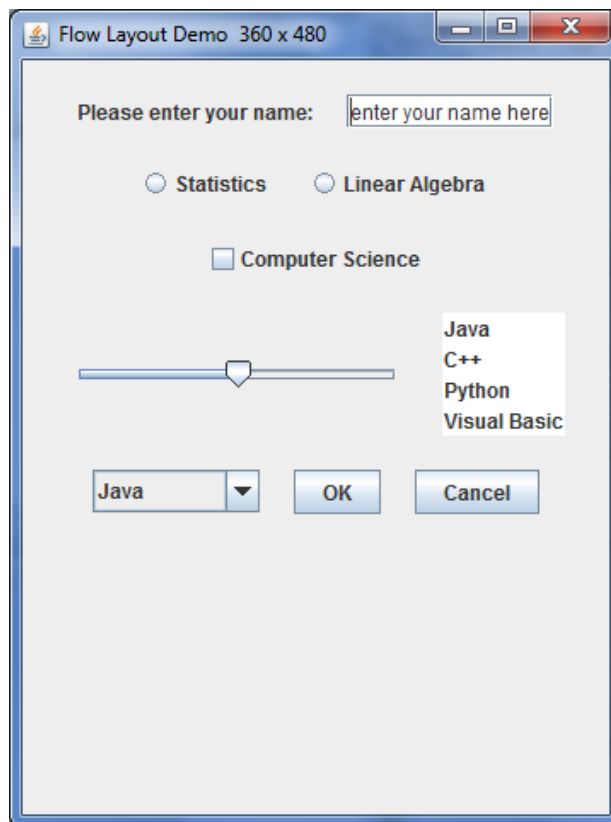
In this example, we will rewrite the code from the last example to use a FlowLayout layout manager. The line that creates MyLayout was changed. Here is the original code:

```
// create a new a grid layout for the frame - 5 rows x 2 cols, gaps=20
GridLayout myLayout = new GridLayout(5,2);
```

Here is the revised code: (The documentation and JFrame title were also changed.)

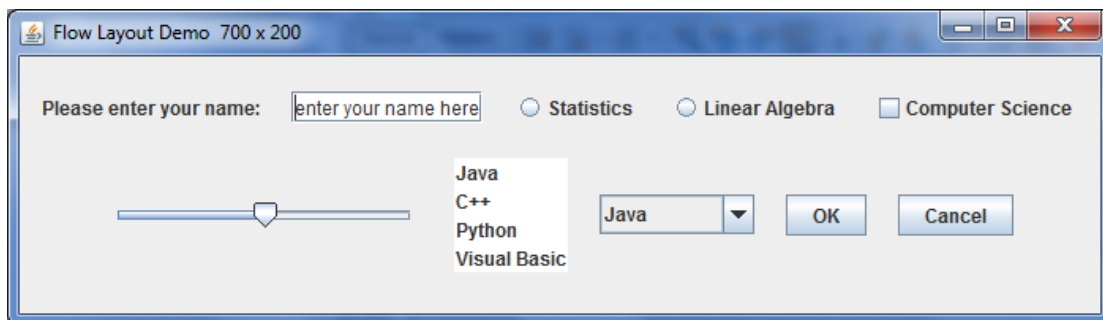
```
// create a new a flowLayout for the frame with gaps set to 20
FlowLayout myLayout = new FlowLayout();
```

This is what the same JFrame from above looks like with a FlowLayout instead of a GridLayout:



The FlowLayout simply puts the components in the JFrame in order, starting at the top left, wrapping around to a new row when there is no more room. It uses the default sizes for components, unlike the GridLayout which changes them to fit the cells in the grid. The NetBeans Project code for this example is in the Chapter 7 files in Canvas as **FlowLayoutDemo**.

Here is the same interface with a flowLayout, but with a different sized JFrame:



In this case, the line:

```
myJFrame.setSize(360, 480);
```

was changed to:

(The JFrame title was also changed.)

```
myJFrame.setSize(700, 200);
```

The components are in the same order, but the JFrame is a different size.

Lab 7C – Programming Example: BorderLayout GUI

In this example, we will place five components in a BorderLayout – one component in each zone: NORTH, SOUTH, EAST, WEST and CENTRAL. The names are capitalized because they are defined in the class as constants identifying each zone in the layout.

START

import AWT.* and Swing.*

create a JFrame object named myJFrame to hold our components

create a new a BorderLayout object named myLayout to hold our components

assign mylayout to be the layout for MyJFrame

create the components for our GUI

- a label with text "Center" for the central zone
- a radio button for the North zone
- a radio button for the South zone
- a String array with a list of programming languages
- a JList - based on the language array
- a JComboBox - based on the language array

add the components to myJFrame , specifying position:

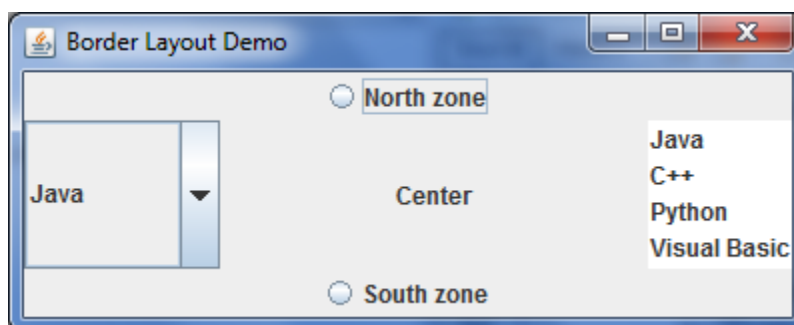
- label in the CENTER zone
- a radio button in the NORTH zone
- a radio button in the SOUTH zone
- JList in the EAST zone
- JComboBox in the WEST zone

set the properties for myJFrame -- title, size, location and exit behavior

activate the GUI – make it visible

STOP

Here is what the JFrame looks like:



Here is the code for the NetBeans Project. It is in the Chapter 7 files in Canvas.


```
/*
 * BorderLayoutDemo.java
 * CSCI 111 Fall 2013
 * last edited October 18, 2013 by C. Herbert
 * This program shows an example of a JFrame with a BorderLayout
 *
 * It is an example of a one-time GUI, not defined as a class
 * No events have been added to the GUI, it only shows components
 */
package borderlayoutdemo;
import javax.swing.*;
import java.awt.*;

public class BorderLayoutDemo {

    public static void main(String[] args)
    {

        // create a frame to hold our components
        JFrame myJFrame = new JFrame();

        // create a new BorderLayout
        BorderLayout myLayout = new BorderLayout();

        // assign mylayout to be the layout for MyJFrame
        myJFrame.setLayout(myLayout);

        // Create a label with text "Center" for the central zone
        JLabel jlCenter = new JLabel("Center");
        jlCenter.setHorizontalAlignment(JLabel.CENTER);

        // Create a radio button for the North zone
        JRadioButton jrbNorth = new JRadioButton("North zone");
        jrbNorth.setHorizontalAlignment(JLabel.CENTER);

        // Create a radio button for the South zone
        JRadioButton jrbSouth = new JRadioButton("South zone");
        jrbSouth.setHorizontalAlignment(JLabel.CENTER);

        // create a String array for the list box and combo box
        String[] languages = new String[] {"Java", "C++", "Python", "Visual Basic"};

        // Create a list using the language array
        JList jlLanguages = new JList(languages);

        // Create a combo box using the language array
        JComboBox jcombLanguages = new JComboBox(languages);

        // add components to BorderLayout )

        myJFrame.add(jlCenter, BorderLayout.CENTER); // Add label to Center zone
        myJFrame.add(jrbNorth, BorderLayout.NORTH); // Add button to North zone
        myJFrame.add(jrbSouth, BorderLayout.SOUTH); // Add button to South zone
        myJFrame.add(jlLanguages, BorderLayout.EAST); // Add list to East zone
        myJFrame.add(jcombLanguages, BorderLayout.WEST); // Add combo box the West zone
    }
}
```

continued on next page

continued from previous page

```

// set the title, size, location and exit behavior for the frame
myJFrame.setTitle("Border Layout Demo");
myJFrame.setSize(400, 160);
myJFrame.setLocation(200, 100);
myJFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// make the frame visible (activate the frame)
myJFrame.setVisible(true);

} // end main()
} // end class

```

Lesson 7.3 Setting Java GUI Component Properties





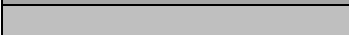








The classes and subclasses of the various Swing components used to create a GUI have a variety of properties that can be manipulated using methods and constants in the classes. In this section we will focus on three properties that can be manipulated in most components:

- color – both foreground color and background color. The foreground color is often text.
- font – the font used for text in some components.
- border – the border around the outside edge of containers and some components.

Setting Java GUI Component Colors

Foreground and background colors are properties of the Component class, the parent class of all components. *Color* is actually a class itself in AWT, with properties, constants and methods.

Several colors are predefined as constants, with both lower camelCase and all uppercase names:

camelCase name	uppercase name	Color Sample
black	BLACK	
blue	BLUE	
cyan	CYAN	
darkGray	DARK_GRAY	
gray	GRAY	
green	GREEN	
lightGray	LIGHT_GRAY	
magenta	MAGENTA	
orange	ORANGE	
pink	PINK	
red	RED	
white	WHITE	
yellow	YELLOW	

In addition to pre-defined colors, new colors can be specified using the RGB color model.

the **RGB color model**, specifies colors by their red, green, and blue component colors – each in the range of 0 to 255. For more details, see http://www.rapidtables.com/web/color/RGB_Color.htm

We can define a custom color in Java by instantiating a new Color using the RGB constructor of the Color class as follows:

```
new Color(r,g,b);
```

where *r*, *g* and *b* are the red green and blue values in the range 0 to 255. The color constructor can be used any place that a Color is called for, or the new color can be saved using a Color variable:

```
Color myColor = new Color(255,127,127);
```

The **setForeground(Color C)** and **setBackground(Color C)** methods can be applied to almost any components. **C** can be a predefined color, or a new color defined with an RGB color constructor, as shown in the examples below.

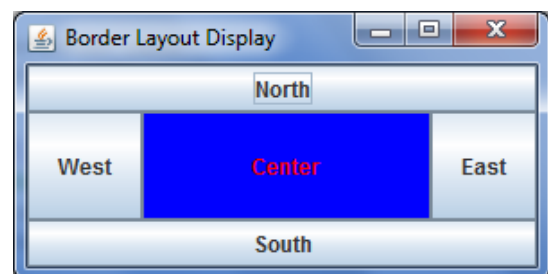
The **setForeground(Color x)** and **setBackground(Color x)** methods are instance methods, which must be used with an instance of a component.

Here are some examples of manipulating a component's colors.

Component Colors Example 1

In this example we will create a blue button with red text using pre-defined colors. We will define the center button in a small JFrame with a BorderLayout.

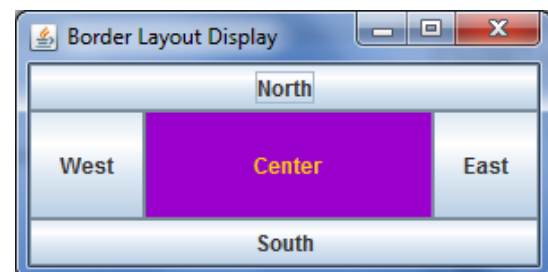
```
JButton jbCenter = new JButton("Center");  
jbCenter.setForeground(Color.red);  
jbCenter.setBackground(Color.blue);
```



Component Colors Example 2

In this example we will create a purple button with gold text using custom RGB-defined colors. We will define the center button in a small JFrame with a BorderLayout.

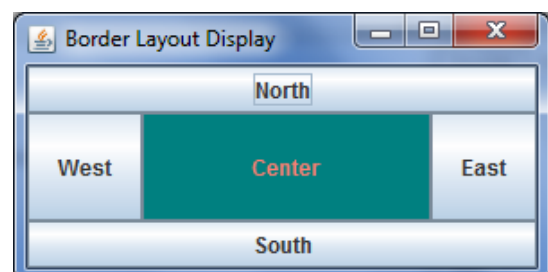
```
JButton jbCenter = new JButton("Center");  
jbCenter.setForeground(new Color(255,204,0));  
jbCenter.setBackground(new Color(153,0,204));
```



Component Colors Example 3

In this example we will define new RGB colors and use them to create a teal button with salmon text for the center button in a small JFrame with a BorderLayout.

```
Color teal = new Color(0,128,128);  
Color salmon = new Color(250,128,114);  
JButton jbCenter = new JButton("Center");  
jbCenter.setForeground(teal);  
jbCenter.setBackground(salmon);
```



Component Colors Example 4

In this example we will create a red label to be displayed next to a textbox with light gray letters. We will define the components in a small JFrame with a FlowLayout.

```
JLabel jlName = new JLabel("Your name:");
jlName.setForeground(Color.RED);

JTextField jtfName = new JTextField(" enter your name here ");
jtfName.setForeground(Color.LIGHT_GRAY);
```



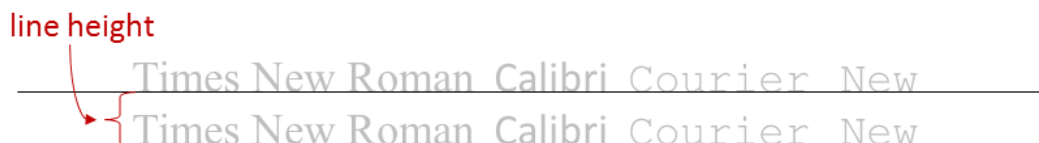
Fonts and Typefaces

We should briefly examine the terminology of typography before looking at fonts in Java.

Each of the symbols that make up a language is a **character** in that language. A **glyph** is a graphical representation of a character. A **typeface** is a set of glyphs for a language that have similar design characteristics. A **font** is a setting for a character or group of characters that includes the *typeface*, *style*, *weight* and *size* of a glyph for each character. *Style* and *weight* are sometimes combined into one style setting, such as ***bold italic***. Color is not considered part of the typeface, but is sometimes specified along with the typeface.

Font is a more specific term than *typeface*. All of the characters in this paragraph, the preceding paragraph, and the heading above are in the *Calibri* typeface, but different fonts are used – the heading is *Calibri 12 point bold*, the body of the text is *Calibri 11 point regular*, with bold and italic used in several places. “*Times New Roman*” is a typeface, while “*Times New Roman 24 point bold*” is a font, but many people – and companies, such as Microsoft – refer to typefaces, such as “*Times New Roman*”, as fonts.

The size of glyphs in a document is normally measured by giving the line height (baseline height) of the font, which is the distance between the baselines of two consecutive lines of text. The term **leading** is synonymous with line height. Original moveable type, dating back to around 1450 in Europe, was composed of carved wooden characters. Printers would space the lines of type by placing strips of soft lead between the blocks of type to achieve the desired spacing. Hence, the term *leading* is synonymous with *line height*. Eventually, cast metal replaced wooden type, before the introduction of modern electronic publishing. It also important to note that the Chinese used movable type as early as 1040 AD, in the Soong Dynasty.



Print is measured in *points and picas*, a unit of printer’s measure, whose exact definition has varied since the first attempts to standardize printing in Europe in the 1700’s. In modern electronic typography a **point** is almost universally accepted as 1/12th of a pica, where a **pica** is 1/6th of an inch. Therefore, a point is 1/72nd of an inch; there are 72 points in an inch. Adobe, Apple, and Microsoft all use this definition of printer’s measure. In some references, the 1/6 of an inch unit is called an *American pica* to distinguish it from the archaic French definition equivalent to roughly 11.33 points.

Setting Java GUI Component Fonts

In the Java language, *Font*, like *Color*, is a class of objects. A Font has three main properties:

- typeface
- style
- size

The **typeface**, also called the font family, can be a physical or logical typeface.

Physical typefaces are the actual typefaces used to display and print characters. They are defined by a font library, which contains a table of glyphs definitions for specific typefaces. All Java Runtime Environments must recognize True Type physical typeface definitions. The True Type standard was developed by Apple and is universally available for free use. True Type typefaces first appeared on the *MacOS 7* and *Windows 3.1* operating systems in 1991. Today they are widely available on almost all systems.

Examples of commonly used physical typefaces are:

- Arial
- Calibri
- Cambria
- Courier New
- Garamond
- Times New Roman

A display of True Type typefaces that are common to all Windows and Mac OS X systems is available online at: <http://www.ampsoft.net/webdesign-l/WindowsMacFonts.html>

Logical typefaces are the five general font families defined in Java: *SERIF*, *SANSERIF*, *MONOSPACED*, *DIALOG*, and *DIALOGINPUT*. (Note that *SANSERIF* is spelled differently than normally.) They are mapped to appropriate physical fonts on specific platforms by the Java Runtime Environment. A **serif** is one of the short stems near the top and bottom of the long parts of some printed letters. A **sanserif** typeface has no serifs. The Calibri typeface used in this paragraph is a sanserif font. **Times New Roman** and **Courier New** are serif typefaces.

Most typefaces are proportionally spaced – a letter like “i” is narrower than a “W”, for example. *Courier* is one of the few monospaced typefaces, in which all letters take up the same width, like they did on old typewriters.

The **styles** for Java fonts are included in a set of constants defined in the Font class. The most commonly used styles in Java are BOLD, ITALIC, and PLAIN.

A font can be defined in a program as needed using a Font constructor:

```
new Font(String typeface, style, size);
```

This is most often used with a *setFont()* method for a GUI component:

```
componentInstance.setFont(new Font(typeface, style, size));
```

A font variable can also be declared and used, just as with Color:

```
Font heading = new Font("Cambria", BOLD, 18);
```

Font Example 1

The following lines of code are used in the NetBeans project *ComponentTypfaceDemo*, which creates the GUI shown:

```
// declare and initialize a Font variable
Font myFont = new Font("Cambria", Font.BOLD, 18);

// Create buttons for each zone, with specified fonts

JButton jbCenter = new JButton("Center");
jbCenter.setFont( new Font("Arial", Font.PLAIN, 24) );

JButton jbWest = new JButton("West");
jbWest.setFont(myFont);

JButton jbEast = new JButton("East");
jbEast.setFont(myFont);

JButton jbNorth = new JButton("North");
jbNorth.setFont(myFont);

JButton jbSouth = new JButton("South");
jbSouth.setFont(myFont);
```



The project file is available in the Chapter 7 project files as *ComponentTypfaceDemo*.

Setting Java GUI Component Borders

A **component border** is a boundary drawn around the outside of a component. Windows, frames, panes, buttons, and text fields are examples of components that have borders.

A *Border* is an object in Java, just like *Color* and *Font*. There is a *BorderFactory* class in Java, whose methods are used to create borders for Java Swing components.

We can use a *setBorder()* method to set borders for components. Just as with *Color* and *Font*, we can declare and use a variable of type *Border*. However, the *Border* class has no constructor methods, so we use other methods to define a *Border* variable – usually *BorderFactory* methods.

Here is an example of how a *BorderFactory* method can be used to create a border with *setBorder()*:

```
jbuttonHello.setBorder( BorderFactory.createLineBorder(Color.black) );
```

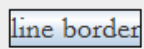


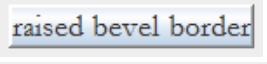
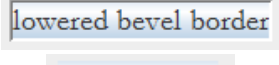
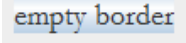
A *Border* variable can be declared and initialized in a similar manner:

```
Border blackLine = BorderFactory.createLineBorder(Color.black);
```

Then the variable can be used whenever we need a component with this type of border:

```
jbuttonHello.setBorder(blackLine);
```

There are six types of borders commonly used with Swing components:

Border Type	Example	BorderFactory code to create this border
Line Border		<code>BorderFactory.createLineBorder(Color.black)</code>
Raised Etched Border		<code>BorderFactory.createEtchedBorder(EtchedBorder.RAISED)</code>
Lowered Etched Border		<code>BorderFactory.createEtchedBorder(EtchedBorder.LOWERED)</code>
Raised Bevel Border		<code>BorderFactory.createRaisedBevelBorder()</code>
Lowered Bevel Border		<code>BorderFactory.createLoweredBevelBorder()</code>
Empty Border		<code>BorderFactory.createEmptyBorder()</code>

Notice that the `CreateLineBorder()` method in the `BorderFactory` class has a color component. There are actually three variations of the `CreateLineBorder` method, which allows you to set the thickness of the border and whether or not it has rounded corners:

```
LineBorder(Color color)
```

Creates a line border with the specified color and a thickness = 1.

```
LineBorder(Color color, int thickness)
```

Creates a line border with the specified color and thickness in points.

```
LineBorder(Color color, int thickness, boolean roundedCorners)
```

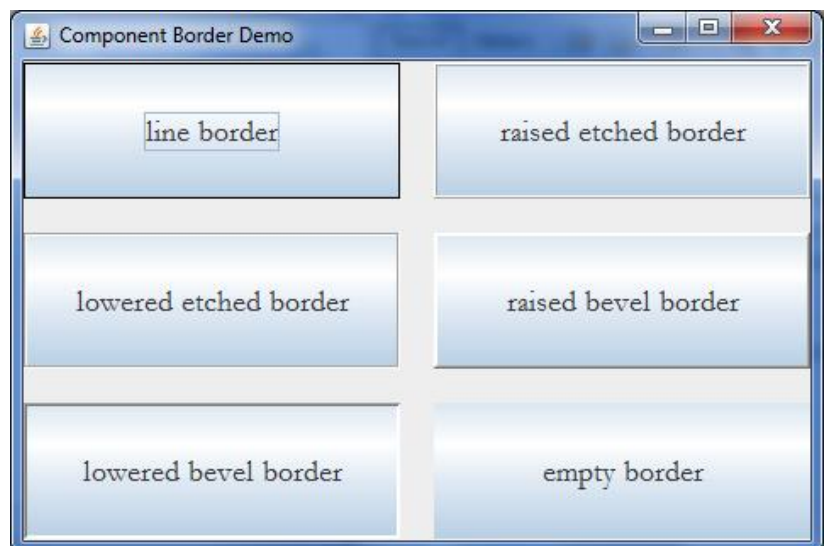
Creates a line border with the specified color, thickness, and corner shape.

Border Example 1

The code that starts on the next page creates the JFrame shown, demonstrating six border types, each on a different button.

The line border is black, with a width set to 2. The typeface is Garamond.

Notice the inner border around the title on the line border button. This indicates the focus of the GUI pointer. At the time the image was captured, the mouse pointer was on the line border button. The inner border will appear around the text on the current button.



```
/* ComponentBorderDemo.java
 * CSCI 111 Fall 2013
 * last edited October 18, 2013 by C. Herbert
 * This program shows how to manipulate GUI component borders
 */
package componentborderdemo;
import javax.swing.*.*;
import java.awt.*.*;
import javax.swing.border.Border;           // All of these import statements or the
import javax.swing.BorderFactory;           // equivalent, are needed to create borders.
import javax.swing.border.TitledBorder;
import javax.swing.border.EtchedBorder;

public class ComponentBorderDemo {
    public static void main(String[] args) {

        // create a frame to hold our components
        JFrame myJFrame = new JFrame();

        // create a new a grid layout for the frame - 3 rows x 2 columns, gaps=20
        GridLayout myLayout = new GridLayout(3,2);
        myLayout.setHgap(20);
        myLayout.setVgap(20);

        // assign mylayout to be the layout for MyJFrame
        myJFrame.setLayout(myLayout);

        // declare and initialize Border variables
        Border blackLine = BorderFactory.createLineBorder(Color.black, 2) ;
        Border raisedEtched = BorderFactory.createEtchedBorder(EtchedBorder.RAISED);
        Border loweredEtched = BorderFactory.createEtchedBorder(EtchedBorder.LOWERED);
        Border raisedBevel = BorderFactory.createRaisedBevelBorder();
        Border loweredBevel = BorderFactory.createLoweredBevelBorder();
        Border empty = BorderFactory.createEmptyBorder();

        // declare and initialize a font for the buttons
        Font buttonFont = new Font("Garamond", Font.PLAIN, 18);

        // Create the buttons
        JButton jb1 = new JButton("line border");
        jb1.setBorder(blackLine);
        jb1.setFont(buttonFont);

        JButton jb2 = new JButton("raised etched border");
        jb2.setBorder(raisedEtched);
        jb2.setFont(buttonFont);

        JButton jb3 = new JButton("lowered etched border");
        jb3.setBorder(loweredEtched);
        jb3.setFont(buttonFont);

        JButton jb4 = new JButton("raised bevel border");
        jb4.setBorder(raisedBevel);
        jb4.setFont(buttonFont);
    }
}
```

continued on next page

continued from previous page

```
        JButton jb5 = new JButton("lowered bevel border");
        jb5.setBorder(loweredBevel);
        jb5.setFont(buttonFont);

        JButton jb6 = new JButton("empty border");
        jb6.setBorder(empty);
        jb6.setFont(buttonFont);

        // add the buttons to the frame
        myJFrame.add(jb1);
        myJFrame.add(jb2);
        myJFrame.add(jb3);
        myJFrame.add(jb4);
        myJFrame.add(jb5);
        myJFrame.add(jb6);

        // set the title, size, location and exit behavior for the frame
        myJFrame.setTitle("Component Border Demo");
        myJFrame.setSize(480, 320);
        myJFrame.setLocation(200, 100);
        myJFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // make the frame visible (activate the frame)
        myJFrame.setVisible(true);

    } // end main()
} // end class
```

Checkpoint 7.3

1. List three properties that can be manipulated in many Swing component classes.
2. List several colors defined as constants in Java's Color class and how they can be used to set the foreground and background colors of Swing components. How can custom colors be defined?
3. How are the terms *font* and *typeface* related to one another? What about the terms *character* and *glyph*? *Physical typeface* and *Logical typeface*? *Serif* and *san serif*?
4. List and briefly describe several constants in the BorderFactory class and how they are used to put borders on certain Swing components, such as JButtons or JLabels.
5. Show how to create two new JButtons and set their properties as follows:
 - a. JButton1 – white foreground, blue background, bold, 18 point *Arial* typeface, line border
 - b. JButton2 – custom background and foreground (you choose the colors), plain text, 24 point *Lucida Sans* typeface, with a raised bevel border.

Lesson 7.4 Using Image Files in a GUI

There are several ways to incorporate images in a Java GUI. In this example, we will see one of the easiest and most convenient, using image files as icons for JButtons and JLabels. Java has an *ImageIcon* class that we can use to do this.

A Java **ImageIcon** is a graphical icon painted from a stored image, which can be used on buttons, labels, etc. The *ImageIcon* is stored in a byte array, created from a properly formatted *GIF*, *JPG*, or *PNG* file.

Here is the method header for the constructor method in the *ImageIcon* class:

```
public ImageIcon(String filename)
```

The filename can take any of the three following forms:

- a full context filename, such as *"C:/data/images/graduation.jpg"*
- a local filename, such as *"graduation.jpg"*.
- a Web URL, such as *"<http://cwherbert.com/cis103/cover.jpg>"*

Most often, a local filename is used, with the file in a local folder, so the filename would include the local folder name, such as *"images/graduation.jpg"*. When using NetBeans, the local file or local folder should be in the folder for the NetBeans project, as in the following example.

Images on JLabels

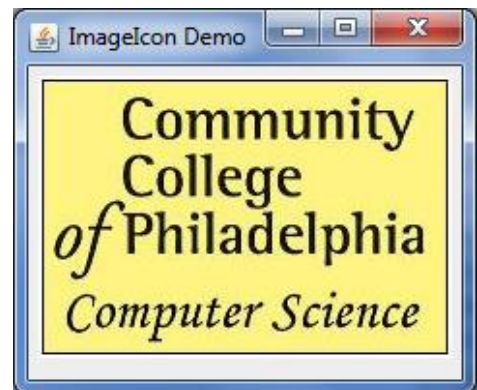
The image on the right shows a JFrame displaying the CCP Computer Science logo. The logo was placed in the JFrame using a JLabel.

The file *CSCI.jpg* is stored in the Netbeans project folder for the project **ImageIconDemo**, which is in the files for Chapter 7 in Canvas.

Here is the line of code that creates the JLabel:

```
JLabel jlCSCI = new JLabel(new ImageIcon("CSCI.jpg"));
```

The program to do this is very simple. Most of it contains the code to set up the JFrame. You should unzip the project and open it in NetBeans to see the complete code and to see how it works.



An *ImageIcon* can also be created and referred to with an *ImageIcon* variable. The line of code above can be replaced with two lines:

```
ImageIcon CSCIpic = new ImageIcon("CSCI.jpg");  
JLabel jlCSCI = new JLabel(CSCIpic);
```

You can try this in the project to see how it works. The only advantage in using an *ImageIcon* variable is that the *ImageIcon* can more easily be used later in the program as necessary.

Images on JButtons

ImageIcons can be displayed on *JButtons* in the same way they can be displayed on *JLabels*:

```
JButton jbGentoo = new JButton(new ImageIcon("penguins/Gentoo.jpg"));
```

In the line above, the image of a Gentoo penguin is stored in the local folder “penguins”. A JButton, named *jbGentoo* is created and the image is linked to the JButton. The JButton will appear on the screen with the image of the penguin on it.

JButtons can have more than one image; in fact they can have several images each linked to a different state of the JButton. When left alone, a JButton is in its normal state, but it has other states, such as “pressed” or selected.

We can link an ImageIcon to the pressed state of a JButton using the `setPressedIcon` method for the JButton class. The *jbGentoo* JButton in the statement above could be linked to a second ImageIcon that appears when the JButton is pressed. Here is an instruction to do this:

```
jbGentoo.setPressedIcon(new ImageIcon("penguins/Gentoo2.jpg"));
```

This instruction, and the one above are used together:

```
JButton jbGentoo = new JButton(new ImageIcon("penguins/Gentoo.jpg"));  
jbGentoo.setPressedIcon(new ImageIcon("penguins/Gentoo2.jpg"));
```

The first instruction defines a JButton named *jbGentoo* with the image *Gentoo.jpg*. The second instruction programs the JButton to display an alternate image, *Gentoo2.jpg*, whenever the JButton is in a pressed state. A JButton is in the pressed state when the user holds down the screen pointer on the JButton. This is different than clicking a JButton.

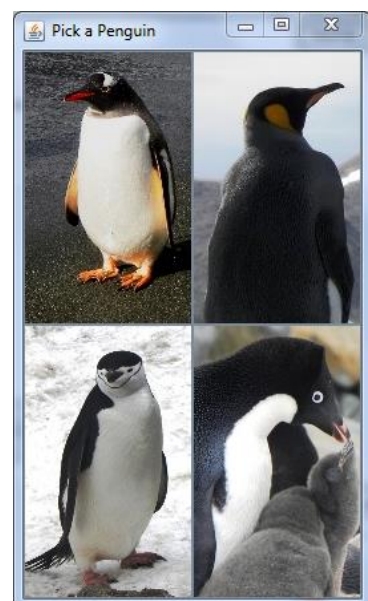
We will see more about clicking buttons and other GUI behavior after we learn more about events in a few weeks. For now, the following example shows us a program with JButtons that show a different image when they are pressed. The event to do this is built into the JButton class. It is an *encapsulated* method whose details we can’t see. It works automatically whenever we define a `PressedIcon` using the `setPressedIcon` method.

Lab 7D – Programming Example: Pick a Penguin

The image on the right shows a JFrame with four pictures of penguins in a two by two grid layout. Whenever one of the buttons is pressed, a different image appears telling the user what kind of penguin is shown in the primary button image.

This example is from the The NetBeans application project *PickaPenguin*, which is included with the files for Chapter 7 in Canvas. You should unzip the project and open it in NetBeans. You can then run the program and try pressing the buttons with the mouse pointer to see the alternate images and learn the names of the different kinds of penguins.

You can also look through the code to see how the JButtons are programmed, using lines similar to the ones shown in the section above.



Adding Tool Tips to Swing Components

The Pick a Penguin program also has an example of tool tips text. **Tool tips** are messages that appear whenever a pointer is paused on a particular JComponent for more than two seconds. A message will appear with information about that component – but only if the tool tips text for that component has been set. We do this with the `setToolTipText()` method, which will work for all JComponents – JButtons, JLabels, etc.

Here is an example of the `setToolTipText()` Method from the Netbeans project:

```
jbGentoo.setToolTipText("Gentoo Penguin, Falkland Islands, by C. Herbert, © 2009");
```

Both *ToolTips* and *ImageIcons* for the pressed state of JButtons can make GUI's seem alive, but everything in this chapter is only about creating and displaying a GUI on the screen. To learn how to truly make a GUI interactive, we will need to see how to associate event listeners and event handlers with Swing components. This will be covered in later chapters, after we learn a bit more about object-oriented programming and defining our own classes. For now,

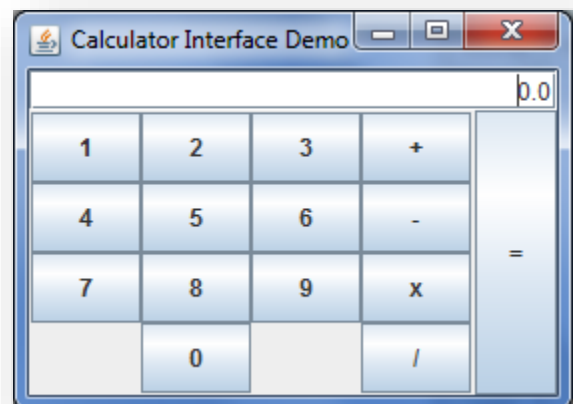
Checkpoint 7.4

1. What is an ImageIcon in Java?
2. What graphics file formats can be used to create an ImageIcon in Java?
3. What are the three different forms that an ImageIcon's file name can take?
4. Show how to display an ImageIcon on a JButton or a JLabel using Java.
5. Show how to set an ImageIcon for the *pressed* state of an existing JButton.

Lesson 7.5 Using Panels to Organize a GUI

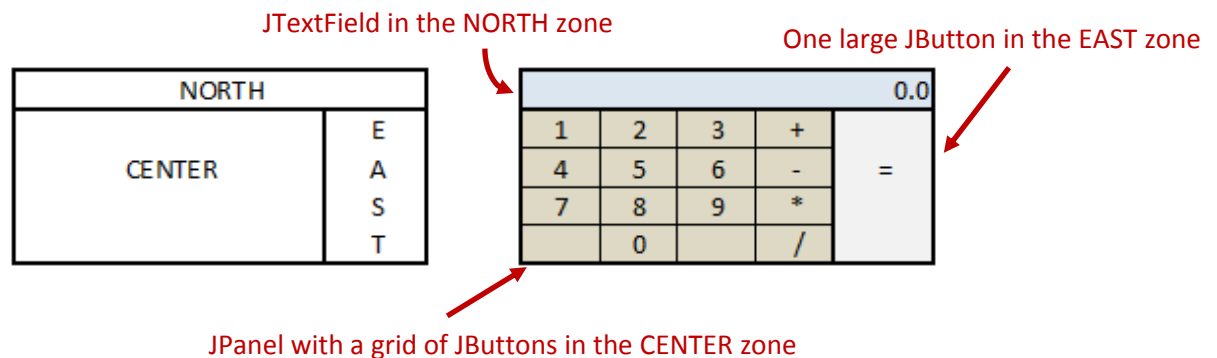
A GUI panel is a container that can serve as a sub-container for components in a GUI Window, Frame or JFrame. The panel can be placed like any other component in a zone or cell created by a layout manager, and can itself contain components organized by a layout manager of the same type or a different type.

JPanel is a class of panels that can serve as sub-containers for JFrames. They are used in the same way that JFrames are used – with a layout manager, added components, and so on, similar to JFrames, except they cannot stand alone; they must be in another container.



The image on the last page shows an interface for a calculator. It was created using a *JFrame* with a *BorderLayout*. The display at the top of the calculator is a text field in the NORTH zone of the *BorderLayout*. The large *equals* button is in the EAST zone of the *BorderLayout*. The set of buttons is in a *JPanel* in the CENTER zone. The *JPanel* is organized into 4 rows and 4 columns with a *GridLayout*. The buttons are components of the *JPanel*; the *JPanel* is a component of the larger *JFrame*.

The WEST and SOUTH zones of the *JFrame*'s *BorderLayout* are not used, so they do not appear in the visible *JFrame*.



JPanel example 1

The code that starts on the next pager creates this calculator interface as a one-time GUI. It is not a functioning calculator, just the visual interface. The buttons work, but they don't do anything yet. Once we learn about events and classes, we can create the calculator as a class that can be re-used, and add events to the buttons to build a functioning calculator. On a touch screen system, a user could use the calculator by touching the buttons on the screen.

The code for this example is in the NetBeans project folder *CalulatorInterface* included with the files for Chapter 7 in Canvas.

```
/* CalculatorInterface.java
 * CSCI 111 Fall 2013
 * last edited October 18, 2013 by C. Herbert
 * This program shows an example of an interface for a calculator.
 *
 * It has a BorderLayout with a JPanel with a GridLayout for buttons in the Center zone
 */
package calculatorinterface;

import java.awt.*;
import javax.swing.*;

public class CalculatorInterface {
    public static void main(String[] args) {

        // create a frame to hold the components
        JFrame myJFrame = new JFrame();

        // create a BorderLayout and assign it to the JFrame
        myJFrame.setLayout(new BorderLayout());

        // Create a panel for the number buttons with a 4 x 4 GridLayout
        JPanel buttonsPanel = new JPanel(new GridLayout(4, 4));

        // Declare an array of 10 JButtons for digits 0-9
        JButton[] numberButton = new JButton[10];

        // Create buttons for arithmetic operations
        JButton addButton = new JButton("+");
        JButton subButton = new JButton("-");
        JButton mulButton = new JButton("x");
        JButton divButton = new JButton("/");
        JButton equButton = new JButton("=");

        // define and add buttons to the first row: 1 2 3 +
        // define and add buttons for 1-3 to the panel
        for (int i = 1; i <= 3 ; i++) {
            numberButton[i] = new JButton(""+i);
            buttonsPanel.add(numberButton[i]);
        } // end for

        // add the plus button
        buttonsPanel.add(addButton);

        // define and add buttons to the second row: 4 5 6 -
        // define and add buttons for 4-6 to the panel
        for (int i = 4; i <= 6 ; i++) {
            numberButton[i] = new JButton(""+i);
            buttonsPanel.add(numberButton[i]);
        } // end for

        // add the subtract button
        buttonsPanel.add(subButton);
```

continued on next page

continued from previous page

```
// define and add buttons to the third row: 7 8 9 X
// define and add buttons for 7-9 to the panel
for (int i = 7; i <= 9 ; i++) {
    numberButton[i] = new JButton(""+i);
    buttonsPanel.add(numberButton[i]);
} // end for

// add the multiplication button
buttonsPanel.add(mulButton);

// define and add buttons to the fourth row: [] 0 [] /
// add the a blank placeholder label
JLabel blankLabel1 = new JLabel("");
buttonsPanel.add(blankLabel1);

// add the zero button
numberButton[0] = new JButton("0");
buttonsPanel.add(numberButton[0]);

// add the a blank placeholder label
JLabel blankLabel2 = new JLabel("");
buttonsPanel.add(blankLabel2);

// add the division button
buttonsPanel.add(divButton);

// add the main button panel to the JFrame
myJFrame.add(buttonsPanel, BorderLayout.CENTER);

// add the large equals button to the JFrame
myJFrame.add(equButton, BorderLayout.EAST);

// Create a right justified text field to display the result
JTextField jtResult = new JTextField("0.0");
jtResult.setHorizontalAlignment(JTextField.RIGHT);

// add the result text field to the JFrame
myJFrame.add(jtResult, BorderLayout.NORTH);

// set the title, size, location and exit behavior for the frame
myJFrame.setTitle("Calculator Interface Demo");
myJFrame.setSize(280, 200);
myJFrame.setLocation(200, 100);
myJFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// make the frame visible (activate the frame)
myJFrame.setVisible(true);

} // end main()
} // end class
```

Chapter Questions

1. How does modern GUI technology differ from traditional GUI technology?
2. How do events work in modern event-driven software?
3. A GUI is a coordinated combination of what two kinds of software?
4. What is the difference between a message dialog box and an input dialog box? What are the five message type constants used for JOptionPane message dialog boxes and what are the icons that go with each of them? What icons are used with which pop-up dialog windows?
5. How can numbers be parsed from Strings in Java? What type of exception can occur when doing this?
6. How does object inheritance work in Java?
7. What two APIs are most often used to create GUIs in Java? How do they differ from one another? What systems use interfaces created with Java?
8. What three things do we need to create a basic GUI in Java?
9. What are some of the most commonly used GUI components and what do they do?
10. What are some of the most commonly used GUI Layout Managers and how do they differ from one another?
11. What are two ways we can create and use a GUI in Java?
12. What are the steps needed to create a JFrame GUI?
13. How are the zones organized in BorderLayout GUI?
14. How does the RGB color model specify colors? How can we change the color of text in a Swing component?
15. What is the difference between a font and a typeface? If a font includes typeface, style, weight, and size, why are only three properties specified for fonts? What is the difference between a physical and a logical font?
16. What are the six commonly used borders for Swing components? What class's methods do we use to create Border objects in Java?
17. How can we put images from picture files in a Java GUI? What file formats can we use? What does the setPressedIcon method do with certain Swing components?
18. What is a tool tip in a GUI? How can we add tool tip text to components in Java?
19. How is a JPanel used to help better organize components? What layout managers can be used with JPanels compared to JFrames?
20. What happens to unused zones in a BorderLayout?

Chapter Exercises

1. The zipped folder cards.zip in the files for the chapter in Canvas contains graphic images of playing cards. The faces of the main playing cards are in image files 1.png to 52.png. Create a program in NetBeans to randomly select five cards and display them next to one another in a GUI.
2. The Windows Snipping Tool or similar tool, can be used to and save an image from a computer screen as file. In this way, text on the screen can be turned into an image file. Design an electronic color display that might be used by a car dealer or in a hardware store to help a customer pick a paint color. Your program should display a dozen different colors on JButtons, then show an alternate image with the name of the color and the RGB codes for that color when the button is pressed. This is similar to what the Pick a Penguin application does. You can use the Windows Snipping Tool, a similar tool, or a program such as Microsoft Paint to create the images.
3. Design an electronic photo album that displays several images on each “page” of the album. You should create a single page from the album with six images and controls for the album, such as next page, previous page, etc. You can use dummy images for your interface. The album could be a family photo album, a police book of mugshots or something similar. You can decide on the design of the page and what titles, controls, etc. should be on each page.
4. Create a GUI that represents a control panel for a spaceship for a science fiction video game. Your control panel will be used by a helmsman to pilot the spaceship. You have a great deal of flexibility here. Be creative and make sure to include a description of your interface in your project report.

The NetBeans project for the Calculator Interface shown in the last section of the chapter is included with the files for the chapter in NetBeans. For any of the following, you should create a GUI interface similar to the Calculator Interface. Your project should include at least one JPanel and examples of changes in font, color and border properties of some components.

5. Revise the design of the chapter’s calculator interface to include additional buttons like those found on most real calculators – such as clear, clear entry, square root, and so on. You can decide what additional buttons to place on the calculator and how to arrange them, but you cannot use the two empty spaces next to the zero key. Add a picture in the WEST zone and a message in the SOUTH zone of your design. Implement your design by revising the code for the existing NetBeans project.
6. Design a universal remote control for an entertainment system (Cable TV, etc). Create the interface as a NetBeans project.
7. Design the telephone interface for a smartphone. Create the interface as a NetBeans project.
8. Design a GUI for a video playback system. You can use dummy images to represent video displays, and should use sliders, buttons, and labels as needed. Create the interface as a NetBeans project.
9. Design an interface for an electronic stopwatch. Create the interface as a NetBeans project.
10. Design a screen for an Automated Teller Machine (ATM) that will only have a touch screen interface and no other physical buttons. Your screen can be any one of the screens for the system (opening menu, deposit screen, withdrawal screen, etc.) and should include a numeric keypad and other buttons, information labels, images, etc. as called for in your design.