Programming Fundamentals I
Java Fundamentals

Nguyen Ngoc Minh

Saigon Institute of Technology
Readings

• Readings for these lecture notes
  • Textbook:
    • Tony Gaddis & Godfrey, Starting Out with Java From Control Structures through Data Structures
  • Reference:
    • Cay S. Horstmann and Gary Cornell, Core Java™

• These lecture notes contain material © Sun, Tony Gaddis, and Godfrey Muganda, 2007
Chapter Objectives

Upon completion of this chapter, you should be able to:

• Use Abstract Windowing Toolkit (AWT)
• Use Java Foundation Classes (JFC)
• Understand Layout Managers
Outline

- Introduction
- Creating Windows
- Inheritance
- Equipping GUI Classes with a main method
- Adding components
- Interface
- Event Driven Programming
- Foreground and Background Colors
Introduction

• Nowadays, *graphical user interface* or *GUI* is often used in several applications.
• A GUI is a graphical window or windows.
• A GUI help build beautiful/pro apperance for the program and make the interaction with the user easier.
• GUI’s accept input from keyboard, mouse.
• A window in a GUI consists of *components* that:
  • present data to the user
  • allow interaction with the application.
Introduction

• Some common GUI components are:
  • buttons, labels, text fields, check boxes, radio buttons, combo boxes, and sliders.
Introduction
JFC, AWT, Swing

- Java programmers use the *Java Foundation Classes (JFC)* to create GUI applications.
- The two sets of JFC classes that we focus on are AWT and Swing classes.
- *Abstract Windowing Toolkit (AWT)* consists of classes for creating graphical user interfaces.
- Using AWT, programmers can create GUI components for applications.
Introduction
JFC, AWT, Swing

• The AWT does not actually draw user interface components on the screen.
• The AWT communicates with a layer of software, peer classes.
• Each version of Java for a particular operating system has its own set of peer classes.
Introduction
JFC, AWT, Swing

- AWT can offer only components that are common to all the operating systems that support Java.
- The behavior of components across various operating systems can differ.
- Programmers cannot easily extend the AWT components.
- AWT components are commonly called *heavyweight components*.
- In an application that uses an AWT class, it is necessary to use the following statement.
  
  ```java
  import java.awt.*;
  ```
Introduction
JFC, AWT, Swing

- *Swing* is a library of classes that provide an improved alternative for creating GUI applications and applets.
- Very few Swing classes rely on peer classes, so they are referred to called *lightweight components*.
- Swing draws most of its own components.
- Swing components have a consistent look and predictable behavior on any operating system.
- Swing components can be easily extended.
- To use Swing classes, must import following package:
  
  ```java
  import javax.swing.*;
  ```
Outline

• Introduction
• **Creating Windows**
• Inheritance
• Equipping GUI Classes with a main method
• Adding components
• Interface
• Event Driven Programming
• Foreground and Background Colors
• Layout Managers
Creating windows

- Often, applications need one or more windows with various components.
- A window is a *container*, which is simply a component that holds other components.
- A container that can be displayed as a window is a *frame*.
- In a Swing application, you create a frame from the JFrame class.
Creating windows

- A frame is a basic window that has:
  - a border around it,
  - a title bar, and
  - a set of buttons for:
    - minimizing,
    - maximizing, and
    - closing the window.

- These standard features are sometimes referred to as window *decorations*. 
Creating windows

- Example: ShowWindow.java
Creating windows

- The following import statement is needed to use the swing components:
  
  ```java
  import javax.swing.*;
  ```
- In the main method, two constants are declared:
  
  ```java
  final int WINDOW_WIDTH = 350,
  WINDOW_HEIGHT = 250;
  ```
- We use these constants later in the program to set the size of the window.
- The window’s size is measured in pixels.
- A pixel (picture element) is one of the small dots that make up a screen display.
Creating windows

• An instance of the JFrame class needs to be created:
  
  JFrame window = new JFrame();

• This statement:
  
  • creates a JFrame object in memory and
  • assigns its address to the window variable.

• The string that is passed to the setTitle method will appear in the window’s title bar when it is displayed.
  
  window.setTitle("A Simple Window");

• A JFrame is initially invisible.
Creating windows

- To set the size of the window:
  ```java
  window.setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
  ```
- To specify the action to take place when the user clicks on the close button.
  ```java
  window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  ```
  - The setDefaultCloseOperation method takes an int argument which specifies the action.
  - JFrame.EXIT_ON_CLOSE - causes the window to be closed.
  - JFrame.HIDE_ON_CLOSE - causes the window to be hidden from view, but the application does not end.
  - The default action is JFrame.HIDE_ON_CLOSE.
Creating windows

- The following code displays the window:

```java
window.setVisible(true);
```
- The setVisible method takes a boolean argument.
  - true - display the window.
  - false - hide the window.
Outline

• Introduction
• Creating Windows
• Inheritance
• Equipping GUI Classes with a main method
• Adding components
• Interface
• Event Driven Programming
• Foreground and Background Colors
• Layout Managers
Inheritance

• Different kinds of objects often have a certain amount in common with each other.
  • Ex: Student, Teacher: name, age, address.
  • Dog, Cat: hair color, weight, age
• Object-oriented programming allows classes to *inherit* commonly used state and behavior from other classes.
• each class is allowed to have one direct superclass, and each superclass has 0 or many *subclasses*:
Inheritance
Example

- Classes: Person, Student, Teacher
Inheritance
Extending JFrame

• We usually use inheritance to create a new class that extends the JFrame class.

• When a new class extends an existing class, it inherits many of the existing class’s members just as if they were part of the new class.

• These members act just as if they were written into the new class declaration.

• New fields and methods can be declared in the new class.

Example: SimpleWindow.java, SimpleWindowDemo.java
Outline

• Introduction
• Creating Windows
• Inheritance
• **Equipping GUI Classes with a main method**
• Adding components
• Interface
• Event Driven Programming
• Foreground and Background Colors
• Layout Managers
Equipping GUI Classes with a main method

• Java applications always starts execution with a method named main.
• We have seen applications in 2 separate files, one file for the class that defines the GUI window and one file that contains the main method that creates an object of the GUI window class.
• Applications can also be written with the main method directly written into the GUI class.
Outline

• Introduction
• Creating Windows
• Inheritance
• Equipping GUI Classes with a main method
  • Adding components
• Interface
• Event Driven Programming
• Foreground and Background Colors
• Layout Managers
Adding components

- Swing provides numerous components that can be added to a window.
- Three fundamental components are: JLabel, JTextField, JButton
Adding components

Example

![Diagram of a kilometer converter window](image)
Adding components

```java
import javax.swing.*;

public class KilometerConverter extends JFrame{
    private JLabel messageLabel;
    private JTextField kiloTextField;
    private JButton calcButton;

    public KilometerConverter(){
        messageLabel = new JLabel("Enter a distance in kilometer");
        kiloTextField = new JTextField(10);
        calcButton = new JButton();
    }
}
```
Adding components

- A *content pane* is a container that is part of every JFrame object.
- Every component added to a JFrame must be added to its content pane.
- The content pane is not visible and it does not have a border.
Adding components

```java
public class KilometerConverter extends JFrame {
    private JLabel messageLabel;
    private JTextField kiloTextField;
    private JButton calcButton;

    private Container contentPane;

    public KilometerConverter() {
        messageLabel = new JLabel("Enter a distance in kilometer");
        kiloTextField = new JTextField(10);
        calcButton = new JButton();

        contentPane = getContentPane();
        contentPane.add(messageLabel);
        setVisible(true);
    }
}
```
Adding components

- Components are typically placed on a panel and then the panel is added to the content pane.
- A *panel* is also a container that can hold GUI components.
- Panels are commonly used to hold and organize collections of related components.
- Create panels with the JPanel class.
Adding components

```java
import java.awt.BorderLayout;
import java.awt.Container;
import javax.swing.*;

public class KilometerConverter extends JFrame{

    private final int WINDOW_WIDTH = 310;
    private final int WINDOW_HEIGHT = 100;

    private JLabel messageLabel;
    private JTextField kiloTextField;
    private JButton calcButton;
    private JPanel panel;
    private Container contentPane;

    public KilometerConverter(){
        this.setTitle("Kilomete Converter");
        this.setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
    }
}
```
Adding components

messageLabel = new JLabel("Enter a distance in kilometer");
kiloTextField = new JTextField(10);
calcButton = new JButton("Calculate");

panel = new JPanel();

panel.add(messageLabel);
panel.add(kiloTextField);
panel.add(calcButton);

contentPane = getContentPane();
contentPane.add(panel);

this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
this.setLocation(600, 300);
setVisible(true);
Outline

• Introduction
• Creating Windows
• Inheritance
• Equipping GUI Classes with a main method
• Adding components
• Interface
• Event Driven Programming
• Foreground and Background Colors
• Layout Managers
• Radio Buttons
• Check Boxes
• Borders
• Focus on Problem Solving: Extending Classes from JPanel
The outside world interacts with objects through their public methods.
Interface

- The public interace of a class is a contract between the client code and the class that provides the service.
- All methods in an interface are public and abstract.
- All attributes are public, static, and final.
Interface Example

- Interface Flyer declares three operations which are common operations of objects that can fly:

```java
public interface Flyer {
    public void takeOff();
    public void land();
    public void fly();
}
```
Many classes can implement the same Flyer interface.
Interface

• A class which implements an interface must define all methods of that interface.
• To implement an interface, use `implements` keyword.
• Implementing an interface allows a class to become more formal about the behavior it promises to provide.
Interface Example

```java
public class Bird implements Flyer {

    public void fly() {
    }

    public void land() {
    }

    public void takeOff() {
    }

    public void buildNest() {
    }

    public void layEggs() {
    }

}
```
public class temp {
    public static void main(String[] args) {
        Flyer myObject;

        myObject = new Bird();
        myObject.fly();

        myObject = new Airplane();
        myObject.takeOff();

        myObject = new Angel();
        myObject.fly();

        ((Angel)myObject).saveHuman();
    }
}

Interface Example
Outline

• Introduction
• Creating Windows
• Inheritance
• Equipping GUI Classes with a main method
• Adding components
• Interface
• **Event Driven Programming**
• Foreground and Background Colors
• Layout Managers
• Radio Buttons
• Check Boxes
• Borders
• Focus on Problem Solving: Extending Classes from JPanel
Event Driven Programming

• Programs that operate in a GUI environment must be *event-driven*.
• An *event* is an action that takes place within a program, such as the clicking of a button.
• Part of writing a GUI application is creating event listeners.
• An *event listener* is an object that automatically executes one of its methods when a specific event occurs.
Event Driven Programming

- An *event* is an action that takes place within a program, such as the clicking of a button.
- When an event takes place, the component that is responsible for the event creates an event object in memory.
- The *event object* contains information about the event.
- The component that generated the event object is known as the *event source*.
- It is possible that the source component is connected to one or more event listeners.
Event Driven Programming

- An *event listener* is an object that responds to events.
- The source component *fires* an event which is passed to a method in the event listener.
- Event listener classes are specific to each application.
- Event listener classes are commonly written as private *inner classes* in an application.
Event Driven Programming

- All event listener classes must implement an interface.
- There’re lots of listener interfaces build in java.awt.event package.
- Ex: ActionListener, ItemListener, KeyListener, ...


Event Driven Programming
Handle event when pressing a button

- JButton components generate *action events*, which require an *action listener* class.
- Action listener classes must implement the ActionListener interface.

```java
<<interface>>>

ActionListener

+actionPerformed(e: ActionEvent)
```
Event Driven Programming
Handle event when pressing a button

When the button is pressed ...

The JButton component generates an event object and passes it to the action listener object's actionPerformed method.
Event Driven Programming
Registering a Listener

• The process of connecting an event listener object to a component is called *registering* the event listener.
• JButton components have a method named addActionListener.

```java
calcButton.addActionListener(new CalcButtonListener());
```
• When the user clicks on the source button, the action listener object’s actionPerformed method will be executed.
Event Driven Programming
Example

```java
//create a CalcButtonListener object and add it
// to calcButton as a listener for this button
calcButton.addActionListener(new CalcButtonListener());
```

- CalcButtonListener class must be defined somewhere in your program.
- CalcButtonListener class must implement ActionListener interface.
- Usually, listener classes are written as private inner classes.
Event Driven Programming

Example

```java
import java.awt.event.ActionEvent;

public class CalcButtonListener implements ActionListener {

    public void actionPerformed(ActionEvent arg0) {
        JOptionPane.showMessageDialog(null, "You've just clicked Calculate button");
    }
}
```

- CalcButtonListener class is defined as public class.
Event Driven Programming

Example

```java
import java.awt.event.ActionEvent;

public class CalcButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent arg0) {
        JOptionPane.showMessageDialog(null, "You've just clicked Calculate button");
    }
}
```

- **CalcButtonListener** class is defined as public class.
Event Driven Programming
Example

public class KilometerConverter {
    Data Fields
    constructors

    private class CalcButtonListener implements ActionListener {
        Data Fields
        methods
    }
}

- CalcButtonListener class is defined as Inner class.
Event Driven Programming

Example

```java
private class CalcButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent arg0) {
        String input;
        double miles;

        input = kiloTextField.getText();
        miles = Double.parseDouble(input) * 0.6214;
        JOptionPane.showMessageDialog(null, input + " kilometers is " + miles + " miles.");
    }
}
```
Event Driven Programming
The ActionEvent Object

- Event objects contain certain information about the event.
- This information can be obtained by calling one of the event object’s methods.
- Two of these methods are:
  - `getSource()` - returns a reference to the object that generated this event.
  - `getActionCommand()` - returns the action command for this event as a String.
Outline

• Introduction
• Creating Windows
• Inheritance
• Equipping GUI Classes with a main method
• Adding components
• Interface
• Event Driven Programming
• Foreground and Background Colors
• Layout Managers
• Radio Buttons
• Check Boxes
• Borders
• Focus on Problem Solving: Extending Classes from JPanel
Foreground and Background Colors

- Many of the Swing component classes have methods named setBackground and setForeground.
  - `setBackground()` is used to change the color of the component itself.
  - `setForeground()` is used to change the color of the text displayed on the component.
- Each method takes a color constant as an argument.
Foreground and Background Colors

- There are predefined constants that you can use for colors.

```
Color.black    Color.blue
Color.cyan     Color.darkGray
Color.gray     Color.green
Color.lightGray Color.magenta
Color.orange   Color.pink
Color.red      Color.white
Color.yellow
```
Foreground and Background Colors
Example

- Version 1: Write 3 ActionListener classes as 3 Inner Classes.
- Version 2: Write only 1 ActionListener class, use `getSource()` to check which button generated event.
- Version 3: Write only 1 ActionListener class, use `getActionCommand()` to check which button generated event.
Outline

- Introduction
- Creating Windows
- Inheritance
- Equipping GUI Classes with a main method
- Adding components
- Interface
- Event Driven Programming
- Foreground and Background Colors
- **Layout Managers**
- Radio Buttons
- Check Boxes
- Borders
- Focus on Problem Solving: Extending Classes from JPanel
Layout Managers

• An important part of designing a GUI application is determining the layout of the components.
• The term *layout* refers to the positioning and sizing of components.
• In Java, you do not normally specify the exact location of a component within a window.
  • A *layout manager* is an object that:
    • controls the positions and sizes of components, and
    • makes adjustments when necessary.
Layout Managers

• The layout manager object and the container work together.
  • Java provides several layout managers:
  •FlowLayout - Arranges components in rows. This is the default for panels.
    • BorderLayout - Arranges components in five regions: North, South, East, West, and Center.
    • This is the default layout manager for a JFrame object’s content pane.
  • GridLayout - Arranges components in a grid with rows and columns.
Layout Managers

- The Container class is one of the base classes that many components are derived from.
- Any component that is derived from the Container class can have a layout manager added to it.
- You add a layout manager to a container by calling the setLayout method.
  ```java
  JPanel panel = new JPanel();
  panel.setLayout(new BorderLayout());
  ```
- In a JFrame constructor you might use:
  ```java
  Container contentPane = getContentPane();
  contentPane.setLayout(new FlowLayout());
  ```
Layout Managers

FlowLayout

- FlowLayout is the default layout manager for JPanel objects.
- Components appear horizontally, from left to right, in the order that they were added. When there is no more room in a row, the next components “flow” to the next row.
public class FlowWindow extends JFrame {
    private final int WINDOW_WIDTH = 200;
    private final int WINDOW_HEIGHT = 105;
    public FlowWindow() {
        this.setTitle("Flow Layout Example");
        this.setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLayout(new FlowLayout());
        JButton button1 = new JButton("Button 1");
        JButton button2 = new JButton("Button 2");
        JButton button3 = new JButton("Button 3");
        this.add(button1);
        this.add(button2);
        this.add(button3);
        this.setVisible(true);
    }
Layout Managers
FlowLayout

- **By default**, FlowLayout manager align **CENTER** components
- You can change this default setting by using an overloaded constructor and pass one of following constant as argument:
  
  FlowLayout.CENTER,
  FlowLayout.LEFT, or
  FlowLayout.RIGHT.

- Example:
  ```java
  contentPane.setLayout(new FlowLayout(FlowLayout.LEFT));
  ```
Layout Managers
FlowLayout

• By default, FlowLayout inserts a gap of five pixels between components, horizontally and vertically.
• An overloaded FlowLayout constructor allows these to be adjusted.
• The constructor has the following format:
  FlowLayout(int alignment, int horizontalGap, int verticalGap)
• Example:
  contentPane.setLayout(new FlowLayout(FlowLayout.LEFT, 10, 7));
Layout Managers

BorderLayout

BorderLayout manages five regions where components can be placed.

```
+-----------------+-----------------+-----------------+
| North Region    | Center Region   | East Region     |
|-----------------+-----------------+-----------------|
| West Region     |                 |                 |
|                 | South Region    |                 |
|                 |                 |                 |
+-----------------+-----------------+-----------------+
```
Layout Managers
BorderLayout

- A component placed into a container that is managed by a BorderLayout must be placed into one of five regions:
  - BorderLayout.NORTH
  - BorderLayout.SOUTH
  - BorderLayout.EAST
  - BorderLayout.WEST
  - BorderLayout.CENTER
Layout Managers
BorderLayout (Example)
**Layout Managers**

**BorderLayout (Example)**

```java
this.setLayout(new BorderLayout());

JButton northButton = new JButton("North Button");
JButton southButton = new JButton("South Button");
JButton eastButton = new JButton("East Button");
JButton westButton = new JButton("West Button");
JButton centerButton = new JButton("Center Button");

this.add(northButton, BorderLayout.NORTH);
this.add(southButton, BorderLayout.SOUTH);
this.add(eastButton, BorderLayout.EAST);
this.add(westButton, BorderLayout.WEST);
this.add(centerButton, BorderLayout.CENTER);
```
Layout Managers
BorderLayout

- Each region can hold only one component at a time.
- When a component is added to a region, it is stretched so it fills up the entire region.
- BorderLayout is the default manager for JFrame objects.

```java
contentPane.add(button, BorderLayout.NORTH);
```
- If you do not pass a second argument to the add method, the component will be added to the center region.
Layout Managers
BorderLayout

• Normally the size of a button is just large enough to accommodate the text that it displays.
• The buttons displayed in BorderLayout region will not retain their normal size.
• The components are stretched to fill all of the space in their regions.
Layout Managers
BorderLayout

• If the user resizes the window, the sizes of the components will be changed as well.
  • Components in the north or south regions may be resized horizontally so it fills up the entire region,
  • Components placed in the east or west regions may be resized vertically so it fills up the entire region.
  • Components placed in the center region may be resized both horizontally and vertically so it fills up the entire region.
Layout Managers

BorderLayout

• By default there is no gap between the regions.
• An overloaded BorderLayout constructor allows horizontal and vertical gaps to be specified (in pixels).
• The constructor has the following format
  BorderLayout(int horizontalGap, int verticalGap)
• Example:
  contentPane.setLayout(new BorderLayout(5, 10));
Layout Managers
Nesting Components in a Layout

• Adding components to panels and then nesting the panels inside the regions can overcome the single component limitation of layout regions.

• By adding buttons to a JPanel and then adding the JPanel object to a region, sophisticated layouts can be achieved.
Layout Managers
GridLayout

GridLayout creates a grid with rows and columns, much like a spreadsheet. A container that is managed by a GridLayout object is divided into equally sized cells.
Layout Managers
GridLayout

- GridLayout manager follows some simple rules:
- Each cell can hold only one component.
- All of the cells are the size of the largest component placed within the layout.
- A component that is placed in a cell is automatically resized to fill up any extra space.
- You pass the number of rows and columns as arguments to the GridLayout constructor.
Layout Managers
GridLayout

• The general format of the constructor:

  ```java
  GridLayout(int rows, int columns)
  ```

• Example

  ```java
  contentPane.setLayout(new GridLayout(2, 3));
  ```

• A zero (0) can be passed for one of the arguments but not both.

• passing 0 for both arguments will cause an IllegalArgumentException to be thrown.
Layout Managers

GridLayout

• Components are added to a GridLayout in the following order (for a 5x5 grid):

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
</tr>
</tbody>
</table>

GridLayout also accepts nested components:
Layout Managers
GridLayout (Example)

```
this.setLayout(new GridLayout(2, 3));

JButton button1 = new JButton("Button 1");
JButton button2 = new JButton("Button 2");
JButton button3 = new JButton("Button 3");
JButton button4 = new JButton("Button 4");
JButton button5 = new JButton("Button 5");
JButton button6 = new JButton("Button 6");

this.add(button1);
this.add(button2);
this.add(button3);
this.add(button4);
this.add(button5);
this.add(button6);
```
Layout Managers
GridLayout (Exercise)

- Use GridLayout(2,3)
- Add a panel to each cell
- Set layout of each panel to FlowLayout.
- Add a JLabel and a JButton to each panel
Outline

• Introduction
• Creating Windows
• Inheritance
• Equipping GUI Classes with a main method
• Adding components
• Interface
• Event Driven Programming
• Foreground and Background Colors
• Layout Managers
• Radio Buttons
• Check Boxes
• Borders
• Focus on Problem Solving: Extending Classes from JPanel
Radio Buttons

- JRadioButton constructors:
  JRadioButton(String text)
  JRadioButton(String text, boolean selected)

- Example:
  JRadioButton radio1 = new JRadioButton("Choice 1");
  or
  JRadioButton radio1 = new JRadioButton("Choice 1", true);
Radio Buttons

ButtonGroup

- Radio buttons normally are grouped together.
- In a radio button group only one of the radio buttons in the group may be selected at any time.
- Clicking on a radio button selects it and automatically deselects any other radio button in the same group.
- An instance of the ButtonGroup class is used to group radio buttons.
Radio Buttons
ButtonGroup

- The ButtonGroup object creates the *mutually exclusive* relationship between the radio buttons that it contains.

```java
JRadioButton radio1 = new JRadioButton("Choice 1", true);
JRadioButton radio2 = new JRadioButton("Choice 2");
JRadioButton radio3 = new JRadioButton("Choice 3");
ButtonGroup group = new ButtonGroup();
group.add(radio1);
group.add(radio2);
group.add(radio3);
```
Radio Buttons

ButtonGroup

- ButtonGroup objects are not containers like JPanel objects, or content frames.
- If you wish to add the radio buttons to a panel or a content frame, you must add them individually.

```java
panel.add(radio1);
panel.add(radio2);
panel.add(radio3);
```
Radio Buttons
Handle Event for Radio Button

- JRadioButton objects generate an action event when they are clicked.
- To respond to an action event, you must write an action listener class, just like a JButton event handler.
Radio Buttons
Determining Selected Radio Buttons

• The JRadioButton class’s isSelected method returns a boolean value indicating if the radio button is selected.

```java
if (radio.isSelected()) {
    // Code here executes if the radio button is selected.
}
```
Radio Buttons
Selecting a Radio Button in Code

• It is also possible to select a radio button in code with the JRadioButton class’s doClick method.

• When the method is called, the radio button is selected just as if the user had clicked on it.

• As a result, an action event is generated.
  
  radio.doClick();
Outline

• Introduction
• Creating Windows
• Inheritance
• Equipping GUI Classes with a main method
• Adding components
• Interface
• Event Driven Programming
• Foreground and Background Colors
• Layout Managers
• Radio Buttons
• Check Boxes
• Borders
• Focus on Problem Solving: Extending Classes from JPanel
CheckBox

- Check boxes are often displayed in groups but they are not usually grouped in a ButtonGroup.
- The status of each checkbox is independent with each other.
The JCheckBox class is used to create check boxes.

Two JCheckBox constructors:

\[
\text{JCheckBox(String text)}
\]

\[
\text{JCheckBox(String text, boolean selected)}
\]

Example:

```
JCheckBox check1 = new JCheckBox("Macaroni");
```
or

```
JCheckBox check1 = new JCheckBox("Macaroni",
```
CheckBox
Handle Event for CheckBox

- When a JCheckBox object is selected or deselected, it generates an *item event*.
- There must be a ItemListener class what is registered with the CheckBox object.
- Item Listener classes must implement the ItemListener interface.
### CheckBox

Handle Event for CheckBox

- Create an object of the class
- Register the item listener object with the JCheckBox component.
- On an event, the itemStateChanged method of the item listener object is automatically run
  - The event object is passed in as an argument.
The isSelected method will determine whether a JCheckBox component is selected.
The method returns a boolean value.

```java
if (checkBox.isSelected()) {
   // Code here executes if the check box is selected.
}
```
CheckBox
Selecting Check Boxes in Code

• It is possible to select check boxes in code with the JCheckBox class’s doClick method.
• When the method is called, the check box is selected just as if the user had clicked on it.
• As a result, an item event is generated.

`checkBox.doClick();`
Outline

• Introduction
• Creating Windows
• Inheritance
• Equipping GUI Classes with a main method
• Adding components
• Interface
• Event Driven Programming
• Foreground and Background Colors
• Layout Managers
• Radio Buttons
• Check Boxes
• Borders
• Focus on Problem Solving: Extending Classes from JPanel
**Borders**

- Windows have a more organized look if related components are grouped inside borders.

- You can add a border to any component that is derived from the JComponent class.
  - Any component derived from JComponent inherits a method named setBorder
Borders

• The setBorder method is used to add a border to the component.
• The setBorder method accepts a Border object as its argument.
• A Border object contains detailed information describing the appearance of a border.
• The BorderFactory class, which is part of the javax.swing package, has static methods that return various types of borders.
## Borders

<table>
<thead>
<tr>
<th>Border</th>
<th>BorderFactory Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compound border</td>
<td>createCompoundBorder</td>
<td>A border that has two parts: an inside edge and an outside edge. The inside and outside edges can be any of the other borders.</td>
</tr>
<tr>
<td>Empty border</td>
<td>createEmptyBorder</td>
<td>A border that contains only empty space.</td>
</tr>
<tr>
<td>Etched border</td>
<td>createEtchedBorder</td>
<td>A border with a 3D appearance that looks “etched” into the background.</td>
</tr>
<tr>
<td>Line border</td>
<td>createLineBorder</td>
<td>A border that appears as a line.</td>
</tr>
<tr>
<td>Lowered bevel border</td>
<td>createLoweredBevelBorder</td>
<td>A border that looks like beveled edges. It has a 3D appearance that gives the illusion of being sunken into the surrounding background.</td>
</tr>
<tr>
<td>Matte border</td>
<td>createMatteBorder</td>
<td>A line border that can have edges of different thicknesses.</td>
</tr>
<tr>
<td>Raised bevel border</td>
<td>createRaisedBevelBorder</td>
<td>A border that looks like beveled edges. It has a 3D appearance that gives the illusion of being raised above the surrounding background.</td>
</tr>
<tr>
<td>Titled border</td>
<td>createTitledBorder</td>
<td>An etched border with a title.</td>
</tr>
</tbody>
</table>
Outline

• Introduction
• Creating Windows
• Inheritance
• Equipping GUI Classes with a main method
• Adding components
• Interface
• Event Driven Programming
• Foreground and Background Colors
• Layout Managers
• Radio Buttons
• Check Boxes
• Borders
• Focus on Problem Solving: Extending Classes from JPanel
Extending Classes from JPanel