Module 11

Handling GUI-Generated Events

Huang Hua

School of Computer & Information Technology
Beijing Jiaotong University
hhua@bjtu.edu.cn

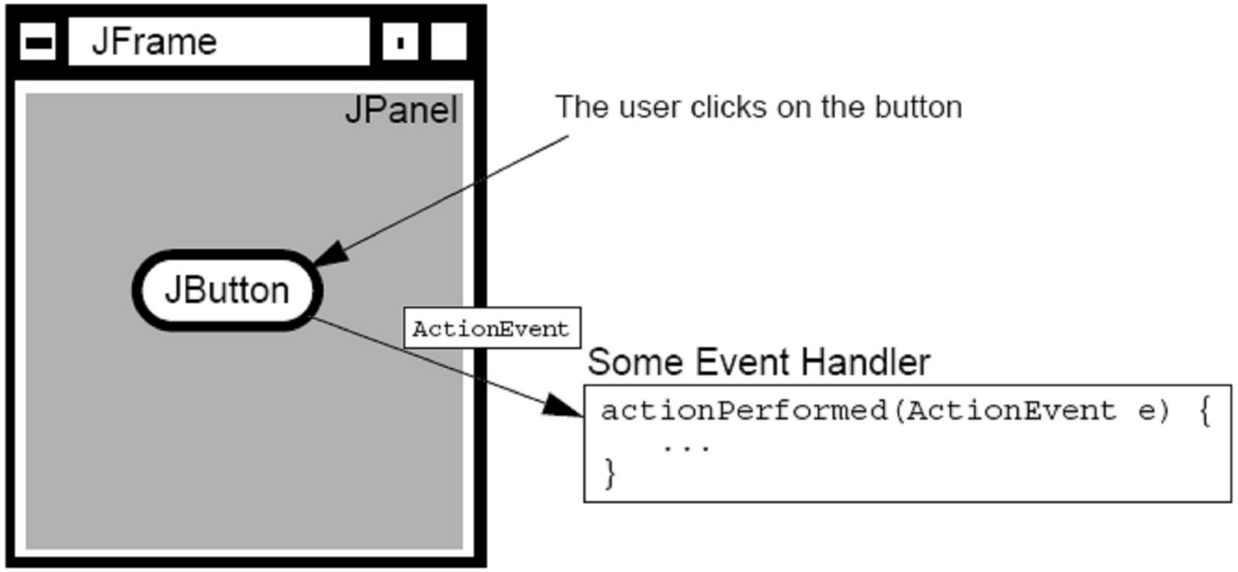
Objectives

- Define events and event handling
- Examine the Java SE event model
- Describe GUI behavior
- Determine the user action that originated an event
- Develop event listeners
- Describe concurrency in Swing-based GUIs and describe the features of the `SwingWorker` class
- Describe how to construct a menu bar, menu, and menu items in a Java GUI
- Understand how to change the color and font of a component

What Is an Event?

- Events – Objects that describe what happened
- Event sources – The **generator** of an event

  Mouse click on a **`JButton`** component generates an **`ActionEvent`** instance with the **button as the source**
- Event handlers – A method that receives an event object, deciphers it, and processes the user's interaction



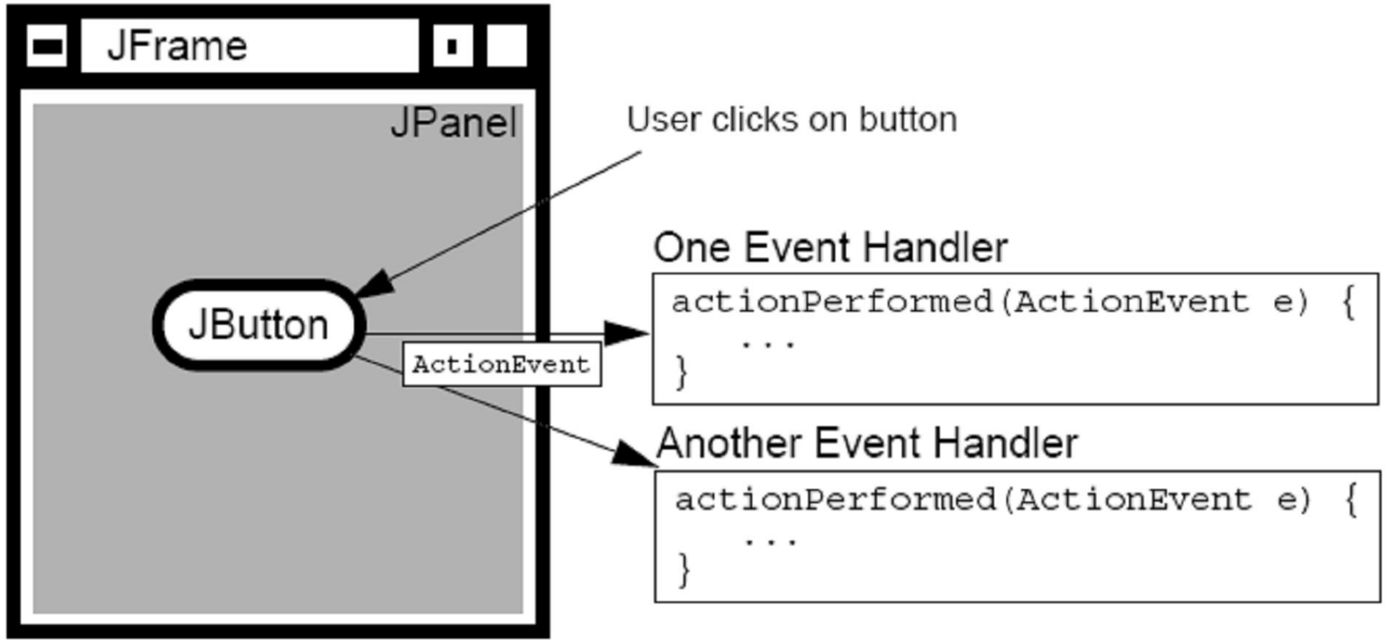
```
actionPerformed(ActionEvent e) {
    ...
}
```

ActionEvent

- **`getSource`** Inherited from EventObject, returns the object on which the Event initially occurred
- **`getActionCommand`**, returns the command name associated with the action
- **`getModifiers`**, returns any modifiers held during the action (Alt, Ctrl, Meta, Shift)
- **`getWhen`**, returns the timestamp when the event occurred
- **`paramString`**, returns a string identifying action and the associated command

Delegation Model

- An event can be sent to many event handlers.



```
actionPerformed(ActionEvent e) {
    ...
}
```

```
actionPerformed(ActionEvent e) {
    ...
}
```

- Event handlers register with components when they are interested in events generated by that component.

Delegation Model(Cont’d)

- Client objects (handlers) register with a GUI component that they want to observe.
- GUI components trigger only the handlers for the type of event that has occurred.
- Most components can trigger more than one type of event.
- The delegation model distributes the work among multiple classes.

A Listener Example

```
import java.awt.*;
import javax.swing.*;
public class TestButton {
  private JFrame f;
  private JButton b;

  public TestButton() {
    f = new JFrame("Test");
    b = new JButton("Press Me!");
    b.setActionCommand("ButtonPressed");
  }

  public void launchFrame() {
    b.addActionListener(new ButtonHandler());
    f.add(b,BorderLayout.CENTER);
    f.pack();
    f.setVisible(true);
  }
```

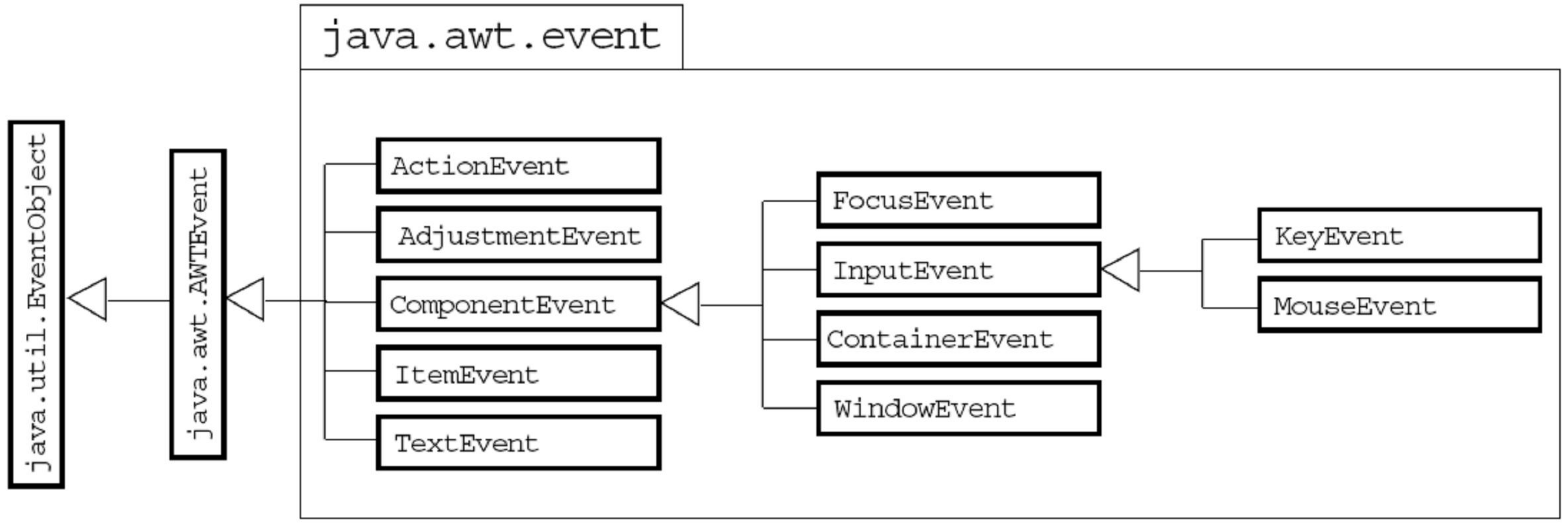
A Listener Example(Cont'd)

```

  public static void main(String args[]) {
    TestButton guiApp = new TestButton();
    guiApp.launchFrame();
  }
}
```

Code for the event listener looks like the following:

```
import java.awt.event.*;

public class ButtonHandler implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    System.out.println("Action occurred");
    System.out.println("Button's command is: "
                        + e.getActionCommand());
  }
}
```

Event Categories



Method Categories and Interfaces

Category	Interface Name	Methods
Action	ActionListener	actionPerformed(ActionEvent)
Item	ItemListener	itemStateChanged(ItemEvent)

Mouse	MouseListener	mousePressed(MouseEvent)
		mouseReleased(MouseEvent)
		mouseEntered(MouseEvent)
		mouseExited(MouseEvent)
		mouseClicked(MouseEvent)

Mouse motion	MouseMotionListener	mouseDragged(MouseEvent)
		mouseMoved(MouseEvent)

Key	KeyListener	keyPressed(KeyEvent)
		keyReleased(KeyEvent)
		keyTyped(KeyEvent)

Focus	FocusListener	focusGained(FocusEvent)
		focusLost(FocusEvent)

Adjustment	AdjustmentListener	adjustmentValueChanged (AdjustmentEvent)

Where is double click?

Method Categories and Interfaces(Cont'd)

Component	ComponentListener	componentMoved(ComponentEvent)
		componentHidden(ComponentEvent)
		componentResized(ComponentEvent)
		componentShown(ComponentEvent)

Window	WindowListener	windowClosing(WindowEvent)
		windowOpened(WindowEvent)
		windowIconified(WindowEvent)
		windowDeiconified(WindowEvent)
		windowClosed(WindowEvent)
		windowActivated(WindowEvent)
		windowDeactivated(WindowEvent)

Container	ContainerListener	componentAdded(ContainerEvent)
		componentRemoved(ContainerEvent)

Window state	WindowStateListener	windowStateChanged(WindowEvent e)

Window focus	WindowFocusListener	windowGainedFocus(WindowEvent e)
		windowLostFocus(WindowEvent e)

Mouse wheel	MouseWheelListener	mouseWheelMoved(MouseWheelEvent e)

Method Categories and Interfaces(Cont'd)

Category	Interface Name	Methods
Input methods	`InputMethodListener`	`caretPositionChanged (InputMethodEvent e)` `inputMethodTextChnaged (InputMethodEvent e)`
Hierarchy	`HierarchyListener`	`hierarchyChanged(HierarchyEvent e)`
Hierarchy bounds	`HierarchyBoundsListener`	`ancestorMoved(HierarchyEvent e)` `ancestorResized(HierarchyEvent e)`
AWT	`AWTEventListener`	`eventDispatched(AWTEvent e)`
Text	`TextListener`	`textValueChanged(TextEvent)`

Complex Example

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public  class TwoListener
  implements MouseMotionListener, MouseListener{
  private JFrame f;
  private JTextField tf;

public TwoListener() {
  f = new JFrame("Two listeners example");
  tf = new JTextField(30);
}

public void launchFrame() {
   JLabel label = new JLabel("Click and drag the  mouse");
  // Add components to the frame
  f.add(label, BorderLayout.NORTH);
  f.add(tf, BorderLayout.SOUTH);
  // Add this object as a listener
  f.addMouseMotionListener(this);
```

Complex Example(Cont'd)

```
    f.addMouseListener(this);
    // Size the frame and make it visible
    f.setSize(300, 200);
    f.setVisible(true);
  }

  // These are MouseMotionListener events
  public void mouseDragged(MouseEvent e) {
    String s ="Mouse dragging:  X = " + e.getX()
               + " Y = " + e.getY();
    tf.setText(s);
  }

  public void mouseEntered(MouseEvent e) {
    String s = "The mouse entered";
    tf.setText(s);
  }

```

Complex Example(Cont’d)

```
  public void mouseExited(MouseEvent e) {
    String s = "The mouse has left the building";
    tf.setText(s);
  }

  //Unused MouseMotionListener method.
  //All methods of a listener must be present  in  the
  //class even if they are not used.
  public void mouseMoved(MouseEvent e) { }

  // Unused MouseListener methods.
  public void mousePressed(MouseEvent e) { }
  public void mouseClicked(MouseEvent e) { }
  public void mouseReleased(MouseEvent e) { }

  public static void main(String args[]) {
    TwoListener two = new TwoListener();
    two.launchFrame();
  }
}
```

Multiple Listeners

- Multiple listeners cause unrelated parts of a program to react to the same event.
- The handlers of all registered listeners are called when the event occurs.
- The order in which the handler methods are called is undefined.

Event Adapters

- The listener classes that you define can extend adapter classes and override only the methods that you need.
- An example is:

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

public  class MouseClickHandler extends MouseAdapter {

  //We just need the mouseClick handler, so we use
  //an adapter to avoid having to write all the
  //event handler methods

  public void mouseClicked(MouseEvent e) {
    // Do stuff with the mouse click...
  }
}
```

Event Handling Using Inner Classes

```
import  java.awt.*;
import  java.awt.event.*;
import  javax.swing.*;
public  class TestInner {
  private JFrame f;
  private JTextField tf; // used byinner class

  public TestInner() {
    f = new JFrame("Inner classes example");
    tf = new JTextField(30);
  }

  class MyMouseMotionListener extends MouseMotionAdapter{
    public void mouseDragged(MouseEvent e) {
      String s = "Mouse dragging: X = "+ e.getX()
                    + " Y = " + e.getY();
      tf.setText(s);
    }
  }
```

Event Handling Using Inner Classes

```

  public void launchFrame() {
    JLabel label = new JLabel("Click and drag the mouse");
    // Add components to the frame
    f.add(label, BorderLayout.NORTH);
    f.add(tf, BorderLayout.SOUTH);
    // Add a listener that uses an Inner class
    f.addMouseMotionListener(new MyMouseMotionListener());
    f.addMouseListener(new MouseClickHandler());
    // Size the frame and make it visible
    f.setSize(300, 200);
    f.setVisible(true);
  }

  public static void main(String args[]) {
    TestInner obj = new TestInner();
    obj.launchFrame();
  }
}
```

Event Handling Using Anonymous Classes

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

public  class **TestAnonymous** {
  private JFrame f;
  private JTextField tf;

  public **TestAnonymous**() {
    f = new JFrame("Anonymous classes example");
    tf = new JTextField(30);
  }

  public static void **main**(String args[]) {
    TestAnonymous obj = new TestAnonymous();
    obj.launchFrame();
  }

```

Event Handling Using Anonymous Classes

```java
  public void launchFrame() {
    JLabel label = new JLabel("Click and drag the mouse");
    // Add components to the frame
    f.add(label, BorderLayout.NORTH);
    f.add(tf, BorderLayout.SOUTH);
    // Add a listener that uses an anonymous class
    f.addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent e) {
        String s = "Mouse dragging:  X = "+ e.getX()
                    + " Y = " + e.getY();
        tf.setText(s);
      }
    }); // <- note the closing parenthesis
    f.addMouseListener(new MouseClickHandler()); // Not shown
    // Size the frame and make it visible
    f.setSize(300, 200);
    f.setVisible(true);
  }
}
```

Concurrency In Swing

To handle a GUI efficiently, the Swing program needs different threads to:

- Execute the application code (**current threads**)
- Handle the events that arise from the GUI (**event dispatch threads**)
- Handle background tasks that might be time consuming (**worker threads**)

Each task in a worker thread is represented by an instance of `javax.swing.`**`SwingWorker`**.

The `SwingWorker` Class

The **`SwingWorker`** class has methods to service the following requirements:

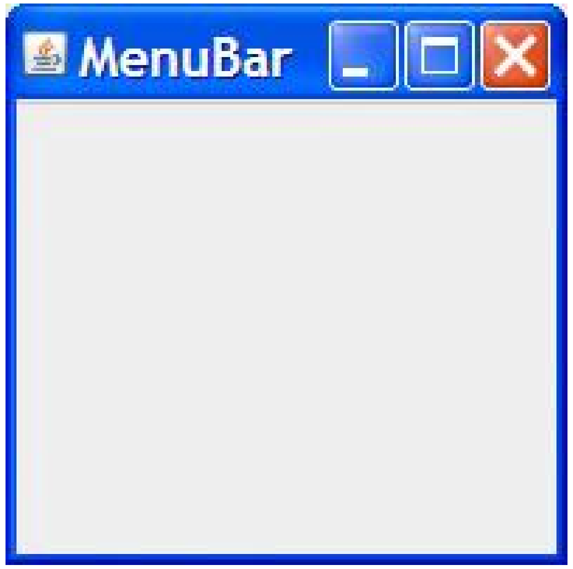
- To provide communication and coordination between worker thread tasks and the tasks on other threads:
  - Properties: **`state`** and **`progress`**
- To execute simple background tasks:
  - **`doInBackground`** method
- To execute tasks that have intermediate results:
  - **`publish`** method
- To cancel the background threads:
  - **`cancel`** method

How to Create a Menu

1. Create a `JMenuBar` object, and set it into a menu container, such as a `JFrame`.
2. Create one or more `JMenu` objects, and add them to the menu bar object.
3. Create one or more `JMenuItem` objects, and add them to the menu object.

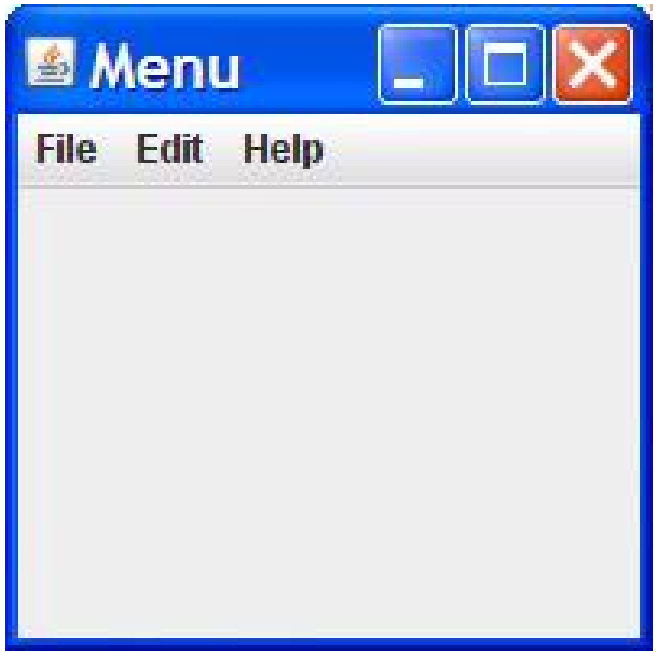
Creating a `JMenuBar`

```
Jframe f = new JFrame("MenuBar");
JmenuBar mb = new JMenuBar();
f.setJMenuBar(mb);
```



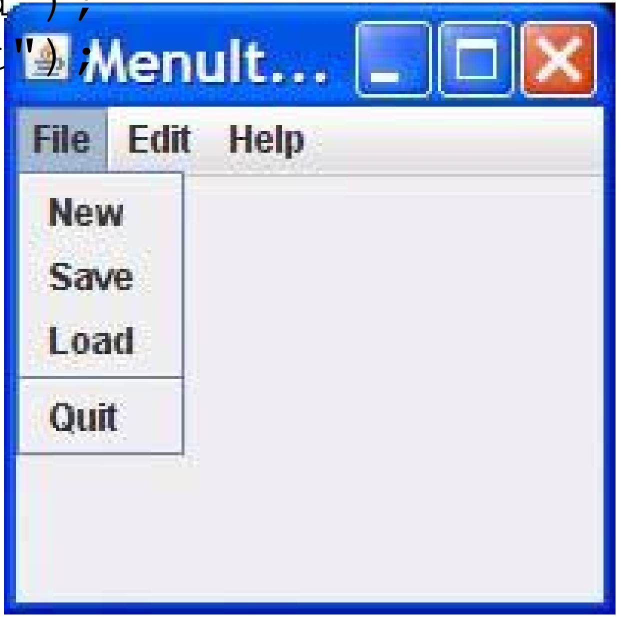
Creating a `JMenu`

```
Jframe f = new JFrame("MenuBar");
JmenuBar mb = new JMenuBar();
Jmenu m1 = new JMenu("File");
Jmenu m2 = new JMenu("Edit");
Jmenu m3 = new JMenu("Help");
mb.add(m1);
mb.add(m2);
mb.add(m3);
f.setJMenuBar(mb);
```



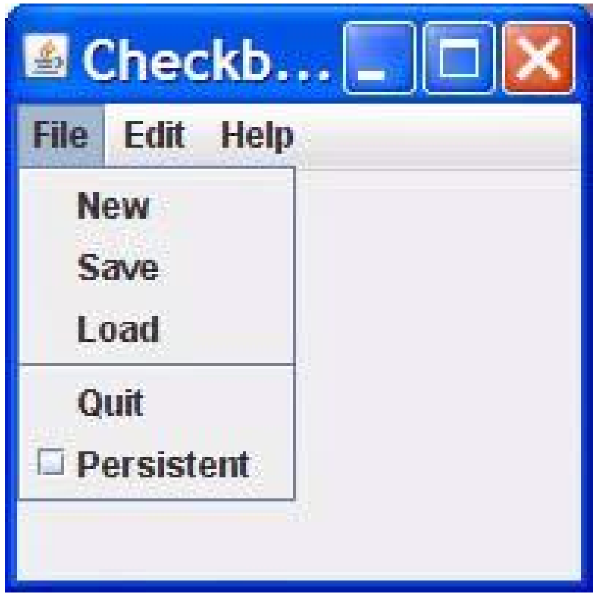
Creating a `JMenuItem`

```
JMenuItem mi1 = new JMenuItem("New");
JMenuItem mi2 = new JMenuItem("Save");
JMenuItem mi3 = new JMenuItem("Load");
JMenuItem mi4 = new JMenuItem("Quit");
mi1.addActionListener(this);
mi2.addActionListener(this);
mi3.addActionListener(this);
mi4.addActionListener(this);
m1.add(mi1);
m1.add(mi2);
m1.add(mi3);
m1.addSeparator();
m1.add(mi4);
```



Creating a `JCheckBoxMenuItem`

```
f = new JFrame("CheckboxMenuItem");
mb = new JMenuBar();
m1 = new JMenu("File");
m2 = new JMenu("Edit");
m3 = new JMenu("Help");
mb.add(m1);
mb.add(m2);
mb.add(m3);
f.setJMenuBar(mb);
......
......
......
......
......
......
mi5 = new JCheckBoxMenuItem("Persistent");
mi5.addItemListener(this);
m1.add(mi5);
```



Controlling Visual Aspects

Commands to control visual aspects of the GUI include:

- Colors: **`java.awt.Color`**

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

predefined colors `Color.red,`

```
setForeground()
setBackground()
```

- Example:

```
Color purple = new Color(255, 0, 255);
JButton b = new JButton("Purple");
b.setBackground(purple);
```

Controlling Visual Aspects

- Fonts(Physical and Logical Fonts):
  You can use the **`setFont()`** method to specify the font used for displaying text
  `Dialog`, **`DialogInput`**, **`Serif`**, and **`SansSerif`** etc. are valid font names(5 Logical Fonts, see Font class for more.)
- Example:

```
Font font = new Font("TimesRoman", Font.PLAIN, 14);
```

- Use the GraphicsEnvironment class to retrieve the set of all available fonts:

```
GraphicsEnvironment ge =
GraphicsEnvironment.getLocalGraphicsEnvironment();
Font[] fonts = ge.getAllFonts();
```

Controlling Visual Aspects

- The **`Toolkit`** class is an abstract superclass of all actual implementations of the Abstract Window Toolkit
- **Subclasses** of Toolkit are used to bind the various components to particular native toolkit implementations
- Useful methods:

```
getDefaultToolkit
getImage(String filename)
getScreenResolution
getScreenSize
getPrintJob
```

- Others:

```
SwingUtilities: convertPointFromScreen(Point p,
Component c), convertPointToScreen(Point p,
Component c)
```

Miscellaneous

- Desktop
  DesktopDemo.java
- SystemTray
  TrayIconDemo.java
- Splash Example
  SplashDemo.java

Summary

- Events and event handling
- Event model
- Event listeners
- Menu bar, menu, and menu items
- Visual aspects controlling

Questions or Comments?