ACM Student Package

- Library of Java classes that simplify input, output and interaction with users.

- Intended for use by students learning Java programming.
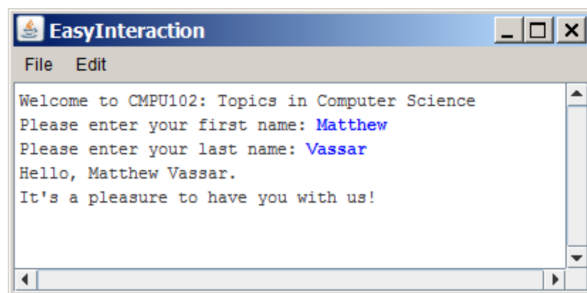
- Available on class wiki.

EasyInteraction.java

```java
import acm.program.*;

public class EasyInteraction extends ConsoleProgram {
    /* Displays a pop-up frame for program execution */
    public void run() {
        println(" Welcome to CMPU102: Data Structures and Java\n");
        String firstName = readLine(" Please enter your first name: ");
        String lastName = readLine(" Please enter your last name: \n");
        println(" Hello, " + firstName + " " + lastName + "." );
        println(" It's a pleasure to have you with us!");
    }

    /* Standard Java entry point */
    public static void main(String[] args) {
        /* Calling the start method of a ConsoleProgram (e.g., EasyInteraction)
         * invokes the system to call the run method. */
        EasyInteraction EI = new EasyInteraction();
        EI.setFont(new Font("sansserif", 0, 18));
        EI.start();
    }
}
```
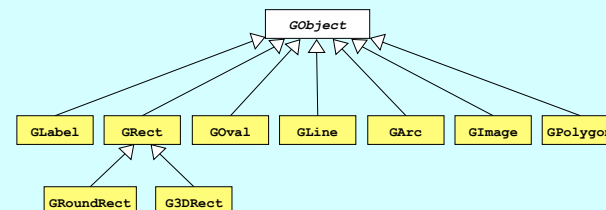
The **main** method creates a new instance of **EasyInteraction** and invokes its **start** method. The **start** method calls the **run** method, which carries on with the program.

EasyInteraction.java

EasyInteraction
File   Edit
```
Welcome to CMPU102: Topics in Computer Science
Please enter your first name: Matthew
Please enter your last name: Vassar
Hello, Matthew Vassar.
It's a pleasure to have you with us!
```

Using the "Shape" Classes

- The shape classes are the **GObject** subclasses that appear in yellow at the bottom of the hierarchy diagram.

```
                          GObject

  GLabel   GRect   GOval   GLine   GArc   GImage   GPolygon

      GRoundRect   G3DRect
```

- Each of the shape classes corresponds precisely to a method in the **Graphics** class in the **java.awt** package. Once you have learned to use the shape classes, you will easily be able to transfer that knowledge to Java's standard graphics tools. Instead of calling methods on a Graphics object to make a shape, you will instantiate one of the shape objects shown above.

The `GLabel` Class

```
public class HelloProgram extends GraphicsProgram {
   public void run() {
      GLabel label = new GLabel("hello, world", 100, 75);
      label.setFont("SansSerif-36");
      label.setColor(Color.RED);
      add(label);
   }
   public static void main(String[] args) {
      HelloProgram hp = new HelloProgram();
      hp.start();
   }
}
```

HelloProgram

hello, world

Centering Labels

```
public class HelloProgram extends GraphicsProgram {
   public void run() {
      GLabel label = new GLabel("hello, world");
      label.setFont("SansSerif-36");
      label.setColor(Color.RED);
      double x = (getWidth() - label.getWidth()) / 2;
      double y = (getHeight() - label.getAscent()) / 2;
      add(label, x, y);
   }
   public static void main(String[] args) {
      HelloProgram hp = new HelloProgram();
      hp.start();
   } }
```

HelloProgram

hello, world
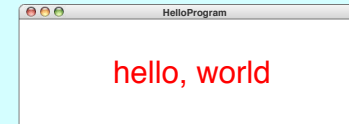
The `GRect` Class

- The **GRect** class implements the **GFillable**, **GResizable**, and **GScalable**.

- Like every other shape class, the **GRect** constructor comes in two forms. The first includes both the location and the size:

  new GRect(*x*, *y*, *width*, *height***)**

  This form makes sense when you know in advance where the rectangle belongs.

- The second constructor defers setting the location:

  new GRect(*width*, *height***)**

  This form is more convenient when you want to create a rectangle and then decide where to put it later.

The `GOval` Class

- The **GOval** class represents an elliptical shape defined by the boundaries of its enclosing rectangle.

- As an example, the following **run** method creates the largest oval that fits within the canvas:

```
public void run() {
   GOval oval = new GOval(getWidth(), getHeight());
   oval.setFilled(true);
   oval.setColor(Color.GREEN);
   add(oval, 0, 0);
}
```

LargestOval
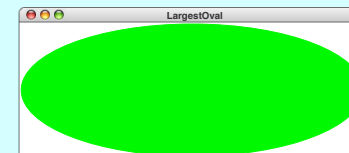
The `GLine` Class

- The **GLine** class represents a line segment that connects two points. The constructor call looks like this:

 new GLine(x_0, y_0, x_1, y_1)

  The points (x_0, y_0) and (x_1, y_1) are called the **start point** and the **end point**, respectively.

- The **GLine** class does not support filling or resizing but does implement the **GScalable** interface. When you scale a line, its start point remains fixed.

- Given a **GLine** object, you can get the coordinates of the two points by calling **getStartPoint** and **getEndPoint**. Both of these methods return a **GPoint** object.

- The **GLine** class also exports the methods **setStartPoint** and **setEndPoint**, which are illustrated on the next slide.

Setting Points in a `GLine`

The following **run** method illustrates the difference between the **setLocation** method (which moves both points together) and **setStartPoint**/**setEndPoint** (which move only one):

```
public void run() {
   GLine line = new GLine(0, 0, 100, 100);
   add(line);
   line.setLocation(200, 50);
   line.setStartPoint(200, 150);
   line.setEndPoint(300, 50);
}
```

LineGeometryExample
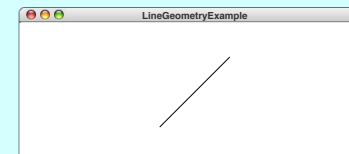
The `GImage` Class

- The **GImage** class is used to display an image from a file. The constructor has the form

  new GImage(*image file*, *x*, *y*)

  where *image file* is the name of a file containing a stored image and *x* and *y* are the coordinates of the upper left corner of the image.

- When Java executes the constructor, it looks for the file in the current directory and then in a subdirectory named **images**.

- To make sure that your programs will run on a wide variety of platforms, it is best to use one of the two most common image formats: the Graphical Interchange Format (GIF) and the Joint Photographic Experts Group (JPEG) format. Typically, your image file name will end with the suffix **.gif** for GIF files and either **.jpg** or **.jpeg** for JPEG files.

Creating Compound Objects

- The **GCompound** class in the **acm.graphics** package makes it possible to combine several graphical objects so that the resulting structure behaves as a single **GObject**.

- The easiest way to think about the **GCompound** class is as a combination of a **GCanvas** and a **GObject**. A **GCompound** is like a **GCanvas** in that you can add objects to it, but it is also like a **GObject** in that you can add it to a canvas.

- As was true in the case of the **GPolygon** class, a **GCompound** object has its own coordinate system that is expressed relative to a **reference point**. When you add new objects to the **GCompound**, you use the local coordinate system based on the reference point. When you add the **GCompound** to the canvas as a whole, all you have to do is set the location of the reference point; the individual components will automatically appear in the right locations relative to that point.

Creating a Face Object

- The first example of the **GCompound** class is the **DrawFace** program, which is illustrated at the bottom of this slide.

- The figure consists of a **GOval** for the face and each of the eyes, a **GPolygon** for the nose, and a **GRect** for the mouth. These objects, however, are not added directly to the canvas but to a **GCompound** that represents the face as a whole.

- This primary advantage of using the **GCompound** strategy is that doing so allows you to manipulate the face as a unit.
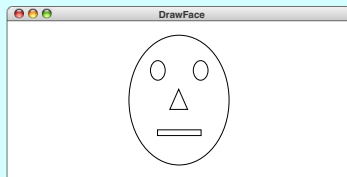
The **GFace** Class

```
import acm.graphics.*;

/** Defines a compound GFace class */
public class GFace extends GCompound {

/** Creates a new GFace object with the specified dimensions */
   public GFace(double width, double height) {
      head = new GOval(width, height);
      leftEye = new GOval(EYE_WIDTH * width, EYE_HEIGHT * height);
      rightEye = new GOval(EYE_WIDTH * width, EYE_HEIGHT * height);
      nose = createNose(NOSE_WIDTH * width, NOSE_HEIGHT * height);
      mouth = new GRect(MOUTH_WIDTH * width, MOUTH_HEIGHT * height);
      add(head, 0, 0);
      add(leftEye, 0.25 * width - EYE_WIDTH * width / 2,
                   0.25 * height - EYE_HEIGHT * height / 2);
      add(rightEye, 0.75 * width - EYE_WIDTH * width / 2,
                    0.25 * height - EYE_HEIGHT * height / 2);
      add(nose, 0.50 * width, 0.50 * height);
      add(mouth, 0.50 * width - MOUTH_WIDTH * width / 2,
                 0.75 * height - MOUTH_HEIGHT * height / 2);
   }
```

The **GFace** Class

```
/* Creates a triangle for the nose */
   private GPolygon createNose(double width, double height) {
      GPolygon poly = new GPolygon();
      poly.addVertex(0, -height / 2);
      poly.addVertex(width / 2, height / 2);
      poly.addVertex(-width / 2, height / 2);
      return poly;
   }

/* Constants specifying feature size as a fraction of the head size */
   private static final double EYE_WIDTH    = 0.15;
   private static final double EYE_HEIGHT   = 0.15;
   private static final double NOSE_WIDTH   = 0.15;
   private static final double NOSE_HEIGHT  = 0.10;
   private static final double MOUTH_WIDTH  = 0.50;
   private static final double MOUTH_HEIGHT = 0.03;

/* Private instance variables */
   private GOval head;
   private GOval leftEye, rightEye;
   private GPolygon nose;
   private GRect mouth;
}
```

Specifying Behavior of a **GCompound**

- The **GCompound** class is useful for defining graphical objects that involve behavior beyond that common to all **GObject**s.

- The **GStoplight** on the next slide implements a stoplight object that exports methods to set and get which lamp is on. The following code illustrates its use:

```
public void run() {
   GStoplight stoplight = new GStoplight();
   add(stoplight, getWidth() / 2, getHeight() / 2);
   stoplight.setColor("RED");
}
```


4
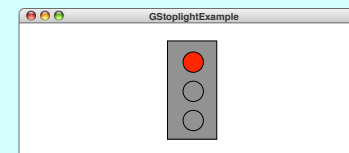
The **GStoplight** Class

```
/**
 * Defines a GObject subclass that displays a stoplight.  The
 * state of the stoplight must be one of the Color values RED,
 * YELLOW, or GREEN.
 */
public class GStoplight extends GCompound {

/** Creates a new Stoplight object, which is initially GREEN */
    public GStoplight() {
        GRect frame = new GRect(FRAME_WIDTH, FRAME_HEIGHT);
        frame.setFilled(true);
        frame.setFillColor(Color.GRAY);
        add(frame, -FRAME_WIDTH / 2, -FRAME_HEIGHT / 2);
        double dy = FRAME_HEIGHT / 4 + LAMP_RADIUS / 2;
        redLamp = createFilledCircle(0, -dy, LAMP_RADIUS);
        add(redLamp);
        yellowLamp = createFilledCircle(0, 0, LAMP_RADIUS);
        add(yellowLamp);
        greenLamp = createFilledCircle(0, dy, LAMP_RADIUS);
        add(greenLamp);
        setState(Color.GREEN);
    }
```

The **GStoplight** Class

```
/** Sets the state of the stoplight */
    public void setState(Color color) {
        if (color.equals(Color.RED)) {
            redLamp.setFillColor(Color.RED);
            yellowLamp.setFillColor(Color.GRAY);
            greenLamp.setFillColor(Color.GRAY);
        } else if (color.equals(Color.YELLOW)) {
            redLamp.setFillColor(Color.GRAY);
            yellowLamp.setFillColor(Color.YELLOW);
            greenLamp.setFillColor(Color.GRAY);
        } else if (color.equals(Color.GREEN)) {
            redLamp.setFillColor(Color.GRAY);
            yellowLamp.setFillColor(Color.GRAY);
            greenLamp.setFillColor(Color.GREEN);
        }
        state = color;
    }

/** Returns the current state of the stoplight */
    public Color getState() {
        return state;
    }
```

The **GStoplight** Class

```
/* Creates a filled circle centered at (x, y) with radius r */
    private GOval createFilledCircle(double x, double y, double r) {
        GOval circle = new GOval(x - r, y - r, 2 * r, 2 * r);
        circle.setFilled(true);
        return circle;
    }

/* Private constants */
    private static final double FRAME_WIDTH = 50;
    private static final double FRAME_HEIGHT = 100;
    private static final double LAMP_RADIUS = 10;

/* Private instance variables */
    private Color state;
    private GOval redLamp;
    private GOval yellowLamp;
    private GOval greenLamp;
}
```

Exercise: Labeled Rectangles

Define a class **GLabeledRect** that consists of an outlined rectangle with a label centered inside. Your class should include constructors that are similar to those for **GRect** but include an extra argument for the label. It should also export **setLabel**, **getLabel**, and **setFont** methods. The following **run** method illustrates the use of the class:

```
public void run() {
    GLabeledRect rect = new GLabeledRect(100, 50, "hello");
    rect.setFont("SansSerif-18");
    add(rect, 150, 50);
}
```


5

Solution: The `GLabeledRect` Class

```java
/** Defines a graphical object combining a rectangle and a label */
public class GLabeledRect extends GCompound {

/** Creates a new GLabeledRect object */
    public GLabeledRect(int width, int height, String text) {
        frame = new GRect(width, height);
        add(frame);
        label = new GLabel(text);
        add(label);
        recenterLabel();
    }

/** Creates a new GLabeledRect object at a given point */
    public GLabeledRect(int x, int y, int width, int height,
                        String text) {
        this(width, height, text);
        setLocation(x, y);
    }

/** Sets the label font */
    public void setFont(String font) {
        label.setFont(font);
        recenterLabel();
    }
```

Solution: The `GLabeledRect` Class

```java
/** Sets the text of the label */
    public void setLabel(String text) {
        label.setLabel(text);
        recenterLabel();
    }

/** Gets the text of the label */
    public String getLabel() {
        return label.getLabel();
    }

/* Recenters the label in the window */
    private void recenterLabel() {
        double x = (frame.getWidth() - label.getWidth()) / 2;
        double y = (frame.getHeight() + label.getAscent()) / 2;
        label.setLocation(x, y);
    }

/* Private instance variables */
    private GRect frame;
    private GLabel label;
}
```

Event Types

- Java events come in many different types. The event types used in this book include the following:
  - **Mouse events**, which occur when the user moves or clicks the mouse
  - **Keyboard events**, which occur when the user types on the keyboard
  - **Action events**, which occur in response to user-interface actions

- Each event type is associated with a set of methods that specify how listeners should respond. These methods are defined in a **listener interface** for each event type.

- As an example, one of the methods in the mouse listener interface is `mouseClicked`. As you would expect, Java calls that method when you click the mouse.

- Listener methods like `mouseClicked` take a parameter that contains more information about the event. In the case of `mouseClicked`, the argument is a `MouseEvent` indicating the location at which the click occurred.

A Simple Event-driven Program

The easiest way to illustrate event handling is by example. The following program listens for mouse clicks and draws a star at the point that each mouse click occurs:

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

/** Draws a star whenever the user clicks the mouse */
public class DrawStarMap extends GraphicsProgram {

    public void init() {
        addMouseListeners();
    }

    public void mouseClicked(MouseEvent e) {
        GStar star = new GStar(STAR_SIZE);
        star.setFilled(true);
        add(star, e.getX(), e.getY());
    }

/* Private constants */
    private static final double STAR_SIZE = 20;
}
```

The **DrawStarMap** Program

This slide simulates the operation of the **DrawStarMap** program.

– The **addMouseListeners** call enables mouse-event reporting.

– Clicking the mouse generates a *mouse clicked* event.

– That event triggers a call to the **mouseClicked** method.

– The program responds by adding a new **GStar** to the canvas.

– Subsequent mouse clicks are treated in exactly the same way.
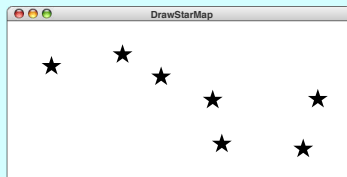
Responding to Mouse Events

- The **DrawStarMap** program on the preceding slide offers a useful illustration of how you can make programs respond to mouse events. The general steps you need are:

  1. Define a **run** method that calls **addMouseListeners**.

  2. Write new definitions of any listener methods you need.

- The most common mouse events are shown in the following table, along with the name of the appropriate listener method:

mouseClicked(e**)**	Called when the user clicks the mouse
mousePressed(e**)**	Called when the mouse button is pressed
mouseReleased(e**)**	Called when the mouse button is released
mouseMoved(e**)**	Called when the user moves the mouse
mouseDragged(e**)**	Called when the mouse is dragged with the button down

The parameter e is a **MouseEvent** object, which provides more data about the event, such as the location of the mouse.

Mouse Listeners in the ACM Libraries

- At a more detailed level, Java's approach to mouse listeners is not as simple as the previous slide implies. To maximize efficiency, Java defines two distinct listener interfaces:

  – The **MouseListener** interface responds to mouse events that happen relatively infrequently, such as clicking the mouse button.

  – The **MouseMotionListener** interface responds to the much more rapid-fire events that occur when you move or drag the mouse.

- The packages in the ACM Java Libraries adopt the following strategies to make mouse listeners easier to use:

  – The **GraphicsProgram** class includes empty definitions for every method in the **MouseListener** and the **MouseMotionListener** interfaces. Doing so means that you don't need to define all of these methods but can instead simply override the ones you need.

  – The **GraphicsProgram** class also defines the **addMouseListeners** method, which adds the program as a listener for both types of events.

  The net effect of these simplifications is that you don't have to think about the difference between these two interfaces.


A Simple Line-Drawing Program

In all likelihood, you have at some point used an application that allows you to draw lines with the mouse. In Java, that program takes less than a page of code.

```
public class DrawLines extends GraphicsProgram {
/* Initializes the program by enabling the mouse listeners */
    public void run() {
        addMouseListeners();
    }

/* Called on mouse press to create a new line */
    public void mousePressed(MouseEvent e) {
        line = new GLine(e.getX(), e.getY(), e.getX(), e.getY());
        add(line);
    }

/* Called on mouse drag to extend the endpoint */
    public void mouseDragged(MouseEvent e) {
        line.setEndPoint(e.getX(), e.getY());
    }

/* Private instance variables */
    private GLine line;
}
```

## Simulating the `DrawLines` Program

- – The `addMouseListeners` call in `init` enables mouse events.
- – Depressing the mouse button generates a *mouse pressed* event.
- – The `mousePressed` call adds a zero-length line to the canvas.
- – Dragging the mouse generates a series of *mouse dragged* events.
- – Each `mouseDragged` call extends the line to the new position.
- – Releasing the mouse stops the dragging operation.
- – Repeating these steps adds new lines to the canvas.
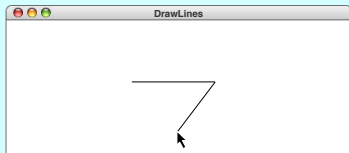
Dragging Objects on the Canvas

The `DragObjects` program on the next slide uses mouse events to drag objects around the canvas.

- – Pressing the mouse button selects an object.
- – Dragging the mouse moves the selected object.
- – Repeating these steps makes it possible to drag other objects.
- – Clicking the mouse moves the selected object to the front.
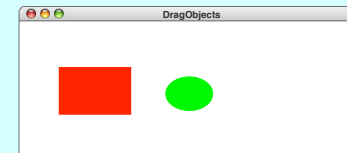
The `DragObjects` Program

```
import acm.graphics.*;
import acm.program.*;
import java.awt.*;
import java.awt.event.*;

/** This class displays a mouse-draggable rectangle and oval */
public class DragObjects extends GraphicsProgram {

/* Initializes the program */
   public void run() {
      GRect rect = new GRect(100, 100, 150, 100);
      rect.setFilled(true);
      rect.setColor(Color.RED);
      add(rect);
      GOval oval = new GOval(300, 115, 100, 70);
      oval.setFilled(true);
      oval.setColor(Color.GREEN);
      add(oval);
      addMouseListeners();
   }
```

The `DragObjects` Program

```
/* Called on mouse press to record the coordinates of the click */
   public void mousePressed(MouseEvent e) {
      last = new GPoint(e.getPoint());
      gobj = getElementAt(last);
   }

/* Called on mouse drag to reposition the object */
   public void mouseDragged(MouseEvent e) {
      if (gobj != null) {
         gobj.move(e.getX() - last.getX(), e.getY() - last.getY());
         last = new GPoint(e.getPoint());
      }
   }

/* Called on mouse click to move this object to the front */
   public void mouseClicked(MouseEvent e) {
      if (gobj != null) gobj.sendToFront();
   }

/* Private instance variables */
   private GObject gobj;          /* The object being dragged */
   private GPoint last;           /* The last mouse position  */

}
```

Responding to Keyboard Events

- The general strategy for responding to keyboard events is similar to that for mouse events, even though the events are different. Once again, you need to take the following steps:

  1. Define a `run` method that calls `addKeyListeners`.
  2. Write new definitions of any listener methods you need.

- The most common key events are:

`keyPressed(e)`	Called when the user presses a key
`keyReleased(e)`	Called when the key comes back up
`keyTyped(e)`	Called when the user types (presses and releases) a key

  In these methods, *e* is a `KeyEvent` object, which indicates which key is involved along with additional data to record which modifier keys (SHIFT, CTRL, and ALT) were down at the time of the event.

Identifying the Key

- The process of determining which key generated the event depends on the type of key event you are using.

- If you are coding the `keyTyped` method, the usual strategy is to call `getKeyChar` on the event, which returns the character generated by that key. The `getKeyChar` method takes account of modifier keys, so that typing the `a` key with the SHIFT key down generates the character `'A'`.

- When you implement the `keyPressed` and `keyReleased` methods, you need to call `getKeyCode` instead. This method returns an integer code for one of the keys. A complete list of the key codes appears in Figure 10–6 on page 361. Common examples include the ENTER key (`VK_ENTER`), the arrow keys (`VK_LEFT`, `VK_RIGHT`, `VK_UP`, `VK_DOWN`), and the function keys (`VK_F1` through `VK_F12`).

Using the Arrow Keys

- Adding the following method to the `DragObjects` program makes it possible to adjust the position of the selected object using the arrow keys:

```
public void keyPressed(KeyEvent e) {
    if (gobj != null) {
        switch (e.getKeyCode()) {
            case KeyEvent.VK_UP:    gobj.move(0, -1); break;
            case KeyEvent.VK_DOWN:  gobj.move(0, +1); break;
            case KeyEvent.VK_LEFT:  gobj.move(-1, 0); break;
            case KeyEvent.VK_RIGHT: gobj.move(+1, 0); break;
        }
    }
}
```

- This method has no effect unless you enable key events in the program by calling `addKeyListeners` in the `init` method.

Creating a Simple GUI

- There is more to creating a modern interactive program than responding to mouse and keyboard events. Most application programs today include a **graphical user interface** or **GUI** (pronounced *gooey*) consisting of buttons and other on-screen controls. Collectively, these controls are called **interactors**.

- Java defines many types of interactors, most of which are part of a collection called the **Swing library**, which is described in section 10.6. You create a GUI by constructing the Swing interactors you need and then arranging them appropriately in the program window.

- The text outlines two strategies for arranging interactors on the screen. The simple approach is to create a **control strip** along one of the edges of the window, as described on the next slide. You can, however, create more general GUIs by using Java's layout managers, as described in section 10.7.

Creating a Control Strip

- When you create an instance of any **Program** subclass, Java divides the window area into five regions as follows:

	NORTH	
W E S T	CENTER	E A S T
	SOUTH	
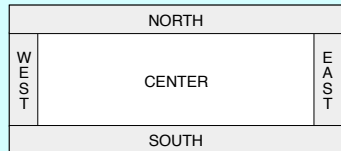
- The **CENTER** region is typically where the action takes place. A **ConsoleProgram** adds a console to the **CENTER** region, and a **GraphicsProgram** puts a **GCanvas** there.

- The other regions are visible only if you add an interactor to them. The examples in the text use the **SOUTH** region as a **control strip** containing a set of interactors, which are laid out from left to right in the order in which they were added.

Creating a GUI with a Single Button

> *Arthur listened for a short while, but being unable to understand the vast majority of what Ford was saying he began to let his mind wander, trailing his fingers along the edge of an incomprehensible computer bank, he reached out and pressed an invitingly large red button on a nearby panel. The panel lit up with the words "Please do not press this button again."*
> —Douglas Adams, *Hitchhiker's Guide to the Galaxy,* 1979

The **HitchhikerButton** program on the next slide uses this vignette from *Hitchhiker's Guide to the Galaxy* to illustrate the process of creating a GUI without focusing on the details. The code creates a single button and adds it to the **SOUTH** region. It then waits for the user to click the button, at which point the program responds by printing a simple message on the console.

# The **HitchhikerButton** Program

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

/*
 * This program puts up a button on the screen, which triggers a
 * message inspired by Douglas Adams's novel.
 */
public class HitchhikerButton extends ConsoleProgram {

/* Initializes the user-interface buttons */
    public void run() {
        add(new JButton("Red"), SOUTH);
        addActionListeners();
    }

/* Responds to a button action */
    public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand().equals("Red")) {
            println("Please do not press this button again.");
        }
    }
}
```

The Swing Interactor Hierarchy

The following diagram shows the Swing classes used in this text. With the exception of **IntField** and **DoubleField**, all of these classes live in the **javax.swing** package.
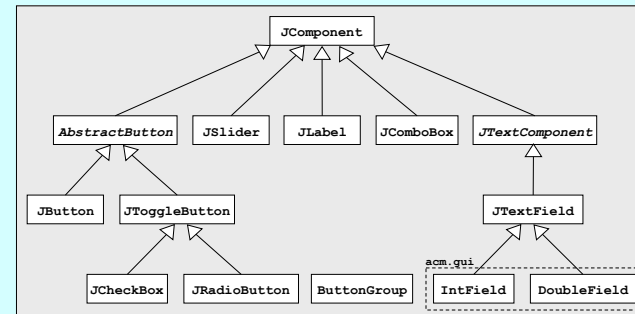
The **JButton** Class

- The most common interactor in GUI-based applications is an on-screen button, which is implemented in Swing by the class **JButton**. A **JButton** object looks something like

  <div align="center">

  `Push Me`

  </div>

- The constructor for the **JButton** class is

  <div align="center">

  `new JButton(label)`

  </div>

  where *label* is a string telling the user what the button does. The button shown earlier on this slide is therefore created by

  `JButton pushMeButton = new JButton("Push Me");`

- When you click on a button, Java generates an **action event**, which in turn invokes a call to **actionPerformed** in any listeners that are waiting for action events.

Detecting Action Events

- Before you can detect action events, you need to enable an action listener for the buttons on the screen. The easiest strategy is to call **addActionListeners** at the end of the **run** method. This call adds the program as a listener to all the buttons on the display.

- You specify the response to a button click by overriding the definition of **actionPerformed** with a new version that implements the correct actions for each button.

- If there is more than one button in the application, you need to be able to tell which one caused the event. There are two strategies for doing so:

  1. Call **getSource** on the event to obtain the button itself.

  2. Call **getActionCommand** on the event to get the **action command** string, which is initially set to the button label.

Adding Features to **DrawStarMap**

- The text illustrates the various Swing interactors by adding new features to the **DrawStarMap** application. The first step is adding a Clear button that erases the screen.

- Adding the button is accomplished in the **init** method:

```
public void init() {
    add(new JButton("Clear"), SOUTH);
    addActionListeners();
}
```

- The response to the button appears in **actionPerformed**:

```
public void actionPerformed(ActionEvent e) {
    if (e.getActionCommand().equals("Clear")) {
        removeAll();
    }
}
```

Exercise: Interactive Stoplight

Using the **GStoplight** class defined in the slides for Chapter 9, write a **GraphicsProgram** that creates a stoplight and three buttons labeled Red, Yellow, and Green, as shown in the sample run below. Clicking on a button should send a message to the stoplight to change its state accordingly.
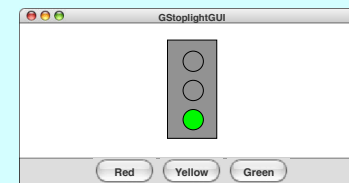
Solution: Interactive Stoplight

```
public class GStoplightGUI extends GraphicsProgram {

   public void run() {
      stoplight = new GStoplight();
      add(stoplight, getWidth() / 2, getHeight() / 2);
      add(new JButton("Red"), SOUTH);
      add(new JButton("Yellow"), SOUTH);
      add(new JButton("Green"), SOUTH);
      addActionListeners();
   }

   public void actionPerformed(ActionEvent e) {
      String cmd = e.getActionCommand();
      if (cmd.equals("Red")) {
         stoplight.setState(Color.RED);
      } else if (cmd.equals("Yellow")) {
         stoplight.setState(Color.YELLOW);
      } else if (cmd.equals("Green")) {
         stoplight.setState(Color.GREEN);
      }
   }

/* Private instance variables */
   private GStoplight stoplight;
}
```
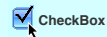
The **JToggleButton** Class

- The **JToggleButton** class is another type of button that is similar to **JButton** but maintains an on/off state. On the screen, a **JToggleButton** looks just like a **JButton** except for the fact that it stays on after you release the mouse button.

- As its name suggests, a **JToggleButton** toggles back and forth between on and off when it is clicked. Clicking the first time turns it from off to on; clicking a second time turns it off.

  [Toggle]

- You can determine whether a **JToggleButton** is on by calling **isSelected**, which returns **true** if the button is on.

- The **JToggleButton** class itself is not used as much as two of its subclasses, **JCheckBox** and **JRadioButton**, which are described on the next two slides.

The **JCheckBox** Class

- The **JCheckBox** class is a subclass of **JToggleButton** and therefore inherits its behavior.

- In terms of its operation, a **JCheckBox** works exactly like an instance of its parent class. The only difference is in what the button looks like on the screen. In a **JCheckBox**, the button label appears to the right of a small square that either contains or does not contain a check mark, like this:

  ☑ CheckBox

- Because a **JCheckBox** is a **JToggleButton**, you can call the **isSelected** method to determine its state.

- Like a **JButton**, a **JCheckBox** generates action events when it is clicked. Both of these classes inherit this behavior from **AbstractButton**, which is their common superclass.

The **JRadioButton** Class

- The **JRadioButton** class also extends **JToggleButton** and behaves in much the same way. In this case, the button is displayed as a circle that is tinted and marked with a dot when it is selected, as follows:

  ⦿ Radio button

- Radio buttons are ordinarily not used individually but instead as a set. If you create a **ButtonGroup** object and then add several radio buttons to it, the Swing libraries make sure that only one of those buttons is selected at a time.

- Grouped radio buttons are used to allow the user to choose among several mutually exclusive options. As an example, the text extends the **DrawStarMap** program to allow the user to choose the size of the star by selecting a radio button:

  ◯ Small   ◯ Medium   ⦿ Large

The `JSlider` Class

- In many applications, you want to let the user adjust a value over a wide range instead of selecting among a set of options.

- The Swing libraries include several different interactors that allow the user to adjust a parameter. The text uses the `JSlider` class, which appears on the screen like this:

  The user can adjust a `JSlider` by dragging the slider knob.

- The simplest form of the `JSlider` constructor looks like this:

  ```
  new JSlider(min, max, value)
  ```

  where *min* and *max* are integers giving the minimum and maximum values of the slider and *value* is the initial value.

- You can retrieve the current value by calling `getValue`.

The `JLabel` Class

- The interactors you display on the screen sometimes don't provide the user with enough information. In such cases, it is useful to include `JLabel` objects, which appear as text strings in the user interface but do not respond to any events.

- As an example, if you wanted to label a slider so that it was clear it controlled size, you could use the following code to produce the control strip shown at the bottom of the screen:

  ```
  add(new JLabel("Small"), SOUTH);
  add(sizeSlider, SOUTH);
  add(new JLabel("Large"), SOUTH);
  ```

  DrawStarMap

  Small   Large

The `JComboBox` Class

- In some applications, you may need to allow the user to chose among a set of options that would take up too much space on the screen if you listed them all. In such situations, you can use the `JComboBox` class, which lists the available options in a popup menu that goes away once the selection is made.

- A `JComboBox` used to select T-shirt sizes might look like this on the screen:

  X-Large
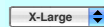
- From the user's point of view, a `JComboBox` works like this:
  - Depressing the mouse brings up a popup menu.
  - Dragging the mouse selects from the different options.
  - Releasing the mouse sets the state to the current option.

- Given that its purpose is to offer the user a choice of options, the `JComboBox` interactor is sometimes called a **chooser**.

Using the `JComboBox` Interactor

- The standard constructor for a `JComboBox` creates an empty interactor that contains no options; you then add the desired options by calling the `addItem` method for each one.

- The code to create the T-shirt size chooser looks like this:

  ```
  JComboBox sizeChooser = new JComboBox();
  sizeChooser.addItem("Small");
  sizeChooser.addItem("Medium");
  sizeChooser.addItem("Large");
  sizeChooser.addItem("X-Large");
  sizeChooser.setEditable(false);
  ```

  The last line prevents the user from typing in some other size.

- The items in a `JComboBox` need not be strings but can instead be any object. The label that appears in the popup menu is determined by applying the object's `toString` method.

- The `getSelectedItem` and `setSelectedItem` methods allow you to determine and set which item is selected.

13

The **JTextField** Class

- Although Swing's set of interactors usually make it possible for the user to control an application using only the mouse, there are nonetheless some situations in which keyboard input is necessary.

- You can accept keyboard input in a user interface by using the **JTextField** class, which provides the user with an area in which it is possible to enter a single line of text.

- The **HelloGUI** program on the next slide illustrates the use of the **JTextField** class in a **ConsoleProgram** that prints a greeting each time a name is entered in the text field.

```
HelloGUI
Hello, world.
Hello, Eric.



                    Name  Eric
```
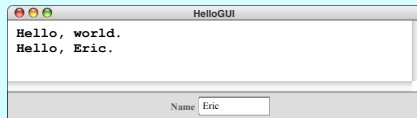
The **HelloGUI** Program

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

/** This class displays a greeting whenever a name is entered */
public class HelloGUI extends ConsoleProgram {

    public void run() {
        nameField = new JTextField(10);
        add(new JLabel("Name"), SOUTH);
        add(nameField, SOUTH);
        nameField.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == nameField) {
            println("Hello, " + nameField.getText());
        }
    }

/* Private instance variables */
    private JTextField nameField;
}
```

Notes on the **JTextField** Class

- The constructor for the **JTextField** class has the form

  new JTextField(*columns***)**

  where *columns* is the number of text columns assigned to the field. The space often appears larger than one might expect, because Java reserves space for the widest characters.

- You can get and set the string entered in a **JTextField** by calling the **getText** and **setText** methods.

- A **JTextField** generates an action event if the user presses the ENTER key in the field. If you want your program to respond to that action event, you need to register the program as an action listener for the field. In the **HelloGUI** example, the action listener is enabled by the statement

  nameField.addActionListener(this);

Numeric Fields

- The **acm.gui** package includes two **JTextField** subclasses that simplify the process of reading numeric input within a graphical user interface. The **IntField** class interprets its text string as an **int**; the **DoubleField** class interprets the text string as a **double**.

- In addition to the usual operations on a **JTextField**, the **IntField** and **DoubleField** classes export **getValue** and **setValue** methods that get and set the numeric value of the field.

- Although it is beyond the scope of the text, the **IntField** and **DoubleField** classes support numeric formatting so that you can control the number of digits in the display. The methods that support this capability are described in the javadoc documentation for these classes.
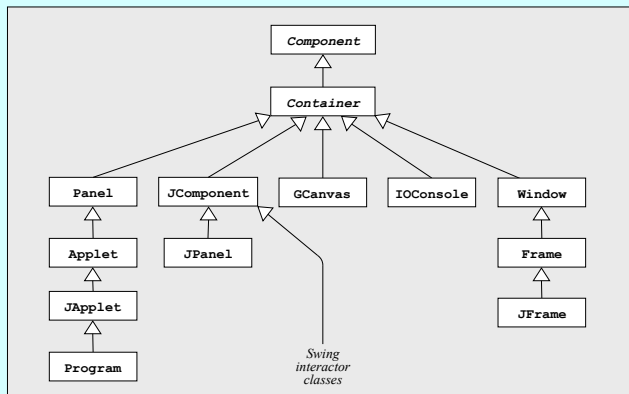
Managing Component Layout

- Although using a control strip along the edge of a program is useful for simple applications, creating a more sophisticated user interface requires you to be able to place interactors anywhere inside a window.

- Arranging interactors to form an elegant, easy-to-use interface is a difficult design challenge. One of the factors that complicates this type of design is the fact that the size of the program window can change over time. A layout that makes sense for a large window may not be appropriate for a small one.

- Java seeks to solve the problem of changing window size by using **layout managers**, which are responsible for arranging interactors and other components when the windows that contain them change size.

Components and Containers

- Understanding how layout managers work is significantly easier if you first understand the relationship between two classes—`Component` and `Container`—that are fundamental to Java's windowing system.

- The `Component` class forms the root of Java's window system hierarchy in the sense that anything that appears in a window is a subclass of `Component`.

- The `Container` class is a subclass of `Component` that can contain other `Component`s, thereby making it possible to nest components inside structures of arbitrary depth.

- As you can see from the hierarchy diagram on the next slide, many of the classes you have seen in the text are subclasses of both `Component` and `Container`. In particular, all Swing interactors, the `GCanvas` class, and the `Program` class are both components and containers.

Classes in the Component Hierarchy

Layout Managers

- In Java, each `Container` has a **layout manager**, which is an object that takes responsibility for arranging the components in that container.

- The layout manager for a container is invoked automatically when the size of the container changes. Although automatic invocation is sufficient for most applications, you may at some point encounter situations in which you need to invoke the layout process by calling `validate` on the container.

- A layout manager uses the following factors to arrange the components in a container:
  - The specific policy set by the layout manager
  - The amount of space available in the container
  - The preferred size of each component
  - Any constraints specified when a component was added

Assigning a New Layout Manager

- You can assign a new layout manager to a **Container** by calling the **setLayout** method with a new layout manager object that is usually constructed specifically for that purpose.

- The **Program** class overrides the definition of **setLayout** so it forwards the request to the **CENTER** region of the program rather than setting the layout for the program itself. This strategy makes it possible to use a control strip even if you call **setLayout**.

- Although it is possible to write layout managers of your own, you can usually rely on the standard layout managers supplied with Java's libraries. The next few slides describe the **BorderLayout**, **FlowLayout**, and **GridLayout** managers. The more flexible **TableLayout** manager is covered in the slides for section 10.8.

The **BorderLayout** Manager

- A **BorderLayout** manager divides its container into five regions, as follows:

	NORTH	
W E S T	CENTER	E A S T
	SOUTH	
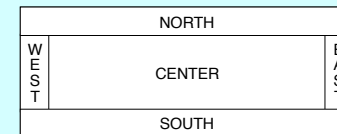
- When you add a component to a container managed by a **BorderLayout**, you need to specify the region, as in

```
container.add(component, BorderLayout.SOUTH);
```

- A **BorderLayout** manager creates the layout by giving the **NORTH** and **SOUTH** components their preferred space and then doing the same for the **WEST** and **EAST** components. Any remaining space is then assigned to the **CENTER** component.
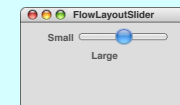
The **FlowLayout** Manager

- The **FlowLayout** manager is in many ways the simplest manager to use and is particularly convenient for getting programs running quickly.

- The **FlowLayout** manager arranges its components in rows from top to bottom and then from left to right within each row. If there is space within the current row for the next component, the **FlowLayout** manager puts it there. If not, the layout manager centers the components on the current row and starts the next one. The **FlowLayout** manager also leaves a little space between each component so that the components don't all run together.

- The problem with the **FlowLayout** manager is that it has no way to make sure that the divisions between the lines come at appropriate places, as illustrated by the example on the next slide.

Limitations of **FlowLayout**

- The following program creates a slider and two labels:

```
public class FlowLayoutSlider extends Program {
    public void run() {
        setLayout(new FlowLayout());
        add(new JLabel("Small"));
        add(new JSlider(0, 100, 50));
        add(new JLabel("Large"));
    }
}
```

- If the program window is wide enough, everything looks fine.

- If, however, you make the program window very narrow, the break between the interactors comes at an awkward place.

16

The `GridLayout` Manager

- The `GridLayout` manager is easiest to illustrate by example. The following `init` method arranges six buttons in a grid with two rows and three columns:

```
public void run() {
   setLayout(new GridLayout(2, 3));
   for (int i = 1; i <= 6; i++) {
      add(new JButton("Button " + i));
   }
}
```

- As you can see from the sample run at the bottom of the slide, the buttons are expanded to fill the cell in which they appear.

# The Inadequacy of Layout Managers

- The main problem with Java's layout managers is that none of the library classes offer the right combination of simplicity and flexibility.

- The simple managers—`BorderLayout`, `FlowLayout`, and `GridLayout`—don't have enough power to design effective user-interface layouts. Unfortunately, the `GridBagLayout` manager, which has the necessary flexibility to create good layout designs, is extremely difficult to use.

- To address the lack of a simple but powerful layout manager, the ACM Java Task Force designed a new `TableLayout` manager, which offers all the power of `GridBagLayout` but is much easier to use. The `TableLayout` manager and its features are covered in the next few slides.

Using the `TableLayout` Class

- The `TableLayout` manager has much in common with the `GridLayout` manager. Both managers arrange components into a two-dimensional grid.

- Like `GridLayout`, the `TableLayout` constructor takes the number of rows and columns in the grid:

new TableLayout(*rows*, *columns*)

- The most noticeable difference between `GridLayout` and `TableLayout` is that `TableLayout` does not expand the components to fit the cells. Thus, if you changed the earlier six-button example to use `TableLayout`, you would see

---
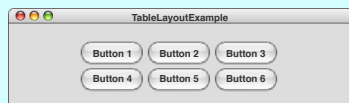
Specifying Constraints

- The real advantage of the `TableLayout` manager is that it allows clients to specify constraints that control the layout. The constraints are expressed as a string, which is passed as a second parameter to the `add` method.

- For example, to add a component **c** to the current table cell and simultaneously indicate that the column should have a minimum width of 100 pixels, you could write

add(c, "width=100");

- To add a label that spans three columns (as a header would likely do), you could write

add(new JLabel("Heading"), "gridwidth=3");

- The `TableLayout` constraints are listed on the next slide.

Available `TableLayout` Constraints

gridwidth=_columns_   or   **gridheight=**_rows_
Indicates that this table cell should span the indicated number of columns or rows.

width=_pixels_   or   **height=**_pixels_
The **width** specification indicates that the width of this column should be at least the specified number of pixels. The **height** specification similarly indicates the minimum row height.

weightx=_weight_   or   **weighty=**_weight_
If the total size of the table is less than the size of its enclosure, **TableLayout** will ordinarily center the table in the available space. If any of the cells, however, are given nonzero **weightx** or **weighty** values, the extra space is distributed along that axis in proportion to the weights specified.

fill=_fill_
Indicates how the component in this cell should be resized if its preferred size is smaller than the cell size. The legal values are **NONE**, **HORIZONTAL**, **VERTICAL**, and **BOTH**, indicating the axes along which stretching should occur; the default is **BOTH**.

anchor=_anchor_
If a component is not being filled along a particular axis, the **anchor** specification indicates where the component should be placed in its cell. The default value is **CENTER**, but you may also use any of the standard compass directions (**NORTH**, **SOUTH**, **EAST**, **WEST**, **NORTHEAST**, **NORTHWEST**, **SOUTHEAST**, or **SOUTHWEST**).

A Temperature Conversion Program

The `TemperatureConverter` program on the next slide uses the `TableLayout` manager to create a simple user interface for a program that converts temperatures back and forth from Celsius to Fahrenheit. The steps involved in using the program are:

1. Enter an integer into either of the numeric fields.
2. Hit ENTER or click the conversion button.
3. Read the result from the other numeric field.
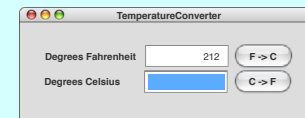
Code for the Temperature Converter

```
/**
 * This program allows users to convert temperatures back and forth
 * from Fahrenheit to Celsius.
 */
public class TemperatureConverter extends Program {

/* Initializes the graphical user interface */
   public void run() {
      setLayout(new TableLayout(2, 3));
      fahrenheitField = new IntField(32);
      fahrenheitField.setActionCommand("F -> C");
      fahrenheitField.addActionListener(this);
      celsiusField = new IntField(0);
      celsiusField.setActionCommand("C -> F");
      celsiusField.addActionListener(this);
      add(new JLabel("Degrees Fahrenheit"));
      add(fahrenheitField);
      add(new JButton("F -> C"));
      add(new JLabel("Degrees Celsius"));
      add(celsiusField);
      add(new JButton("C -> F"));
      addActionListeners();
   }
```

Code for the Temperature Converter

```
/* Listens for a button action */
   public void actionPerformed(ActionEvent e) {
      String cmd = e.getActionCommand();
      if (cmd.equals("F -> C")) {
         int f = fahrenheitField.getValue();
         int c = GMath.round((5.0 / 9.0) * (f - 32));
         celsiusField.setValue(c);
      } else if (cmd.equals("C -> F")) {
         int c = celsiusField.getValue();
         int f = GMath.round((9.0 / 5.0) * c + 32);
         fahrenheitField.setValue(f);
      }
   }

/* Private instance variables */
   private IntField fahrenheitField;
   private IntField celsiusField;
}
```

18

Layout for the Calculator Program

- As a second example of the `TableLayout` manager, the text develops a program that implements a simple four-function calculator, as shown at the bottom of this slide.

- Although the entire `Calculator` program is interesting as an example of object-oriented design, this chapter focuses on the user interface, which is created by the `run` method on the next slide.

Setting up the Calculator Display

```
public void run() {
    setLayout(new TableLayout(5, 4));
    display = new CalculatorDisplay();
    add(display, "gridwidth=4 height=" + BUTTON_SIZE);
    addButtons();
    addActionListeners();
}

private void addButtons() {
    String constraint = "width=" + BUTTON_SIZE + " height=" + BUTTON_SIZE;
    add(new DigitButton(7), constraint);
    add(new DigitButton(8), constraint);
    add(new DigitButton(9), constraint);
    add(new AddButton(), constraint);
    add(new DigitButton(4), constraint);
    add(new DigitButton(5), constraint);
    add(new DigitButton(6), constraint);
    add(new SubtractButton(), constraint);
    add(new DigitButton(1), constraint);
    add(new DigitButton(2), constraint);
    add(new DigitButton(3), constraint);
    add(new MultiplyButton(), constraint);
    add(new ClearButton(), constraint);
    add(new DigitButton(0), constraint);
    add(new EqualsButton(), constraint);
    add(new DivideButton(), constraint);
}
```