Nested **for** Statements

- The body of a control statement can contain other statements. Such statements are said to be **nested**.
- Many applications require nested **`for`** statements. The next slide, for example, shows a program to display a standard checkerboard in which the number of rows and number of columns are given by the constants **`N_ROWS`** and **`N_COLUMNS`**.
- The **`for`** loops in the **`Checkerboard`** program look like this:

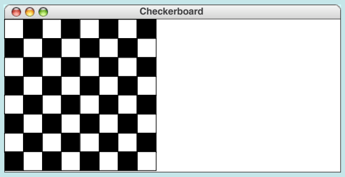
```
for (int i = 0; i < N_ROWS; i++) {
   for (int j = 0; j < N_COLUMNS; j++) {
      Display the square at row i and column j.
   }
}
```

- Because the entire inner loop runs for each cycle of the outer loop, the program displays **`N_ROWS`**×**`N_COLUMNS`** squares.

The **Checkerboard** Program

```
public void run() {
   double sqSize = (double) getHeight() / N_ROWS;
   for (int i = 0; i < N_ROWS; i++) {
      for (int j = 0; j < N_COLUMNS; j++) {
         double x = j * sqSize;
         double y = i * sqSize;
         GRect sq = new GRect(x, y, sqSize, sqSize);
         sq.setFilled((i + j) % 2 != 0);
         add(sq);
      }
   }
}
```

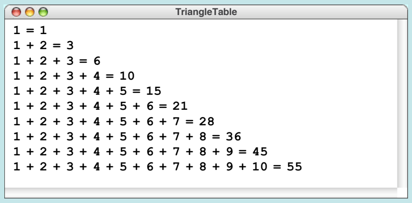
sqSize	i	j	x	y	sq
30.0	8	8	210.0	210.0	□



skip simulation

Exercise: Triangle Number Table

Write a program that duplicates the sample run shown at the bottom on this slide, which displays the sum of the first *N* integers for each value of *N* from 1 to 10. As the output suggests, these numbers can be arranged to form a triangle and are therefore called **triangle numbers**.



```
1 = 1
1 + 2 = 3
1 + 2 + 3 = 6
1 + 2 + 3 + 4 = 10
1 + 2 + 3 + 4 + 5 = 15
1 + 2 + 3 + 4 + 5 + 6 = 21
1 + 2 + 3 + 4 + 5 + 6 + 7 = 28
1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 = 36
1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45
1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55
```

Design Issues: Triangle Number Table

As you think about the design of the **`TriangleTable`** program, it will help to keep the following thoughts in mind:

- The program involves two nested loops. The outer loop runs through each of the values of *N* from 1 to the maximum; the inner loop prints a series of values on each output line.
- The individual elements of each output line are easier to display if you call **`print`** instead of **`println`**. The **`print`** method is similar to **`println`** but doesn't return the cursor position to the beginning of the next line in the way that **`println`** does. Using **`print`** therefore makes it possible to string several output values together on the same line.
- The n^{th} output line contains n values before the equal sign but only $n-1$ plus signs. Your program therefore cannot print a plus sign on each cycle of the inner loop but must instead skip one cycle.

Simple Graphical Animation

The **`while`** and **`for`** statements make it possible to implement simple graphical animation. The basic strategy is to create a set of graphical objects and then execute the following loop:

```
for (int i = 0; i < N_STEPS; i++) {
   update the graphical objects by a small amount
   pause(PAUSE_TIME);
}
```

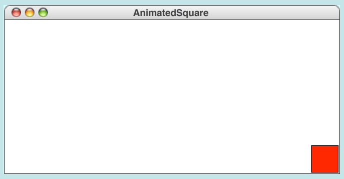
On each cycle of the loop, this pattern updates each animated object by moving it slightly or changing some other property of the object, such as its color. Each cycle is called a **time step**.

After each time step, the animation pattern calls **`pause`**, which delays the program for some number of milliseconds (expressed here as the constant **`PAUSE_TIME`**). Without the call to **`pause`**, the program would finish faster than the human eye can follow.

The **AnimatedSquare** Program

```
public void run() {
   GRect square = new GRect(0, 0, SQUARE_SIZE, SQUARE_SIZE);
   square.setFilled(true);
   square.setFillColor(Color.RED);
   add(square);
   double dx = (getWidth() - SQUARE_SIZE) / N_STEPS;
   double dy = (getHeight() - SQUARE_SIZE) / N_STEPS;
   for (int i = 0; i < N_STEPS; i++) {
      square.move(dx, dy);
      pause(PAUSE_TIME);
   }
}
```

i	dx	dy	square
101	3.0	1.7	🟥



skip simulation

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. Call **`addKeyListeners`** from the constructor
  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

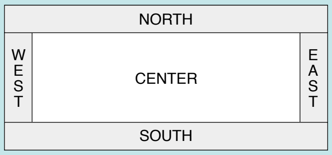
- The process of determining which key generated the event depends on the type of key event you are using.
- If you are coding the **`keyTyped`** method, the usual strategy is to call **`getKeyChar`** on the event, which returns the character generated by that key. The **`getKeyChar`** method takes account 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`**).

Creating a Simple GUI

- In addition to mouse and keyboard events, application programs may 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**, 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 other GUI layouts 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:



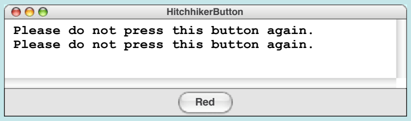
- The **`CENTER`** region is typically where the action takes place. A **`ConsoleProgram`** adds a console to the **`CENTER`** region, and a **`GraphicsProgram`** puts a **`GCanvas`** there.
- The other regions are visible only if you add an interactor to them. The examples in the text use the **`SOUTH`** 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.



```
Please do not press this button again.
Please do not press this button again.
```

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 init() {
      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 `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

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

```
new JButton(label)
```

  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 constructor. 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.

JButton to clear the screen

- Suppose we want to add a Clear button that erases the screen.
- Adding the button is accomplished in the constructor:

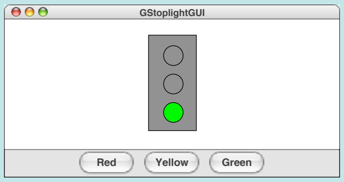
```
public DrawStarMap() {
   add(new JButton("Clear"), SOUTH);
   addActionListeners();
}
```

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

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

Exercise: Interactive Stoplight

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 init() {
      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 `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);
   }
```

page 1 of 3 skip code

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;
   }
```

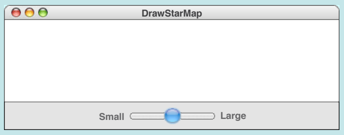
The GStoplight Class

```
/* 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;
}
```

The JLabel Class

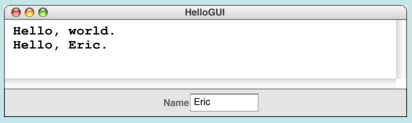
- The interactors you display on the screen sometimes don't provide the user with enough information. In such cases, it is useful to include **`JLabel`** objects, which appear as text strings in the user interface but do not respond to any events.
- As an example, if you wanted to label a slider so that 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);
```



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.



```
Hello, world.
Hello, 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 init() {
      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);
```