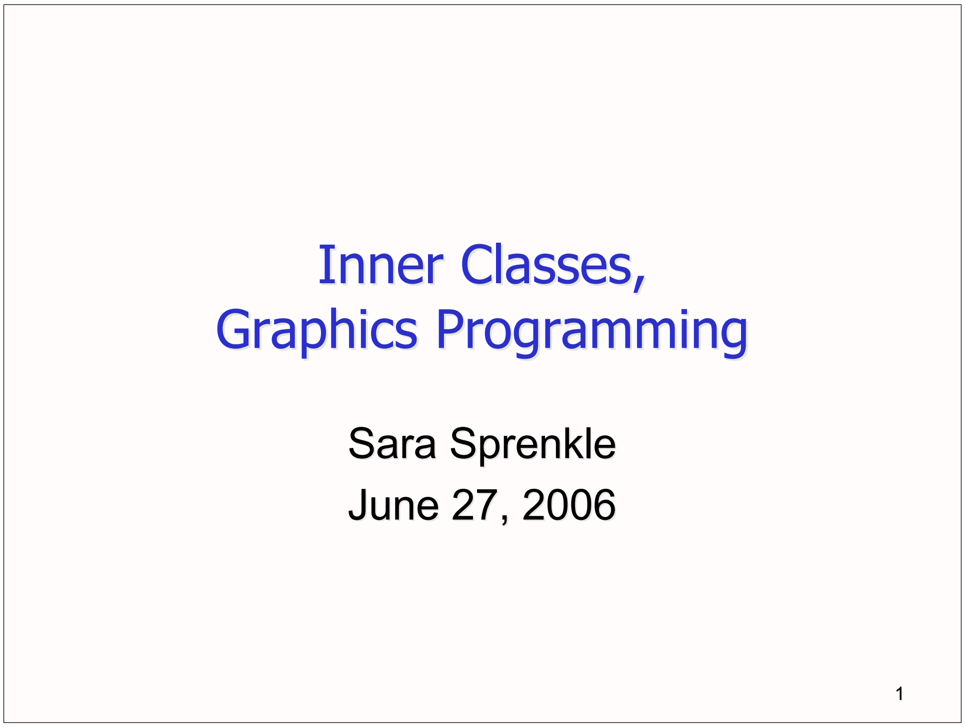
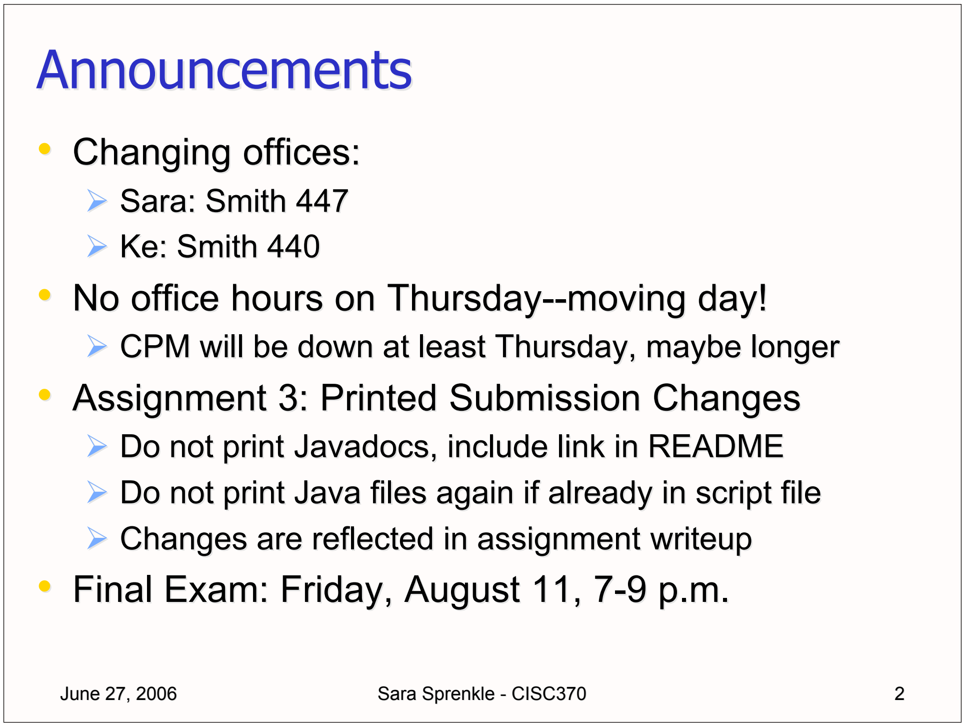


Inner Classes, Graphics Programming

Sara Sprenkle

June 27, 2006



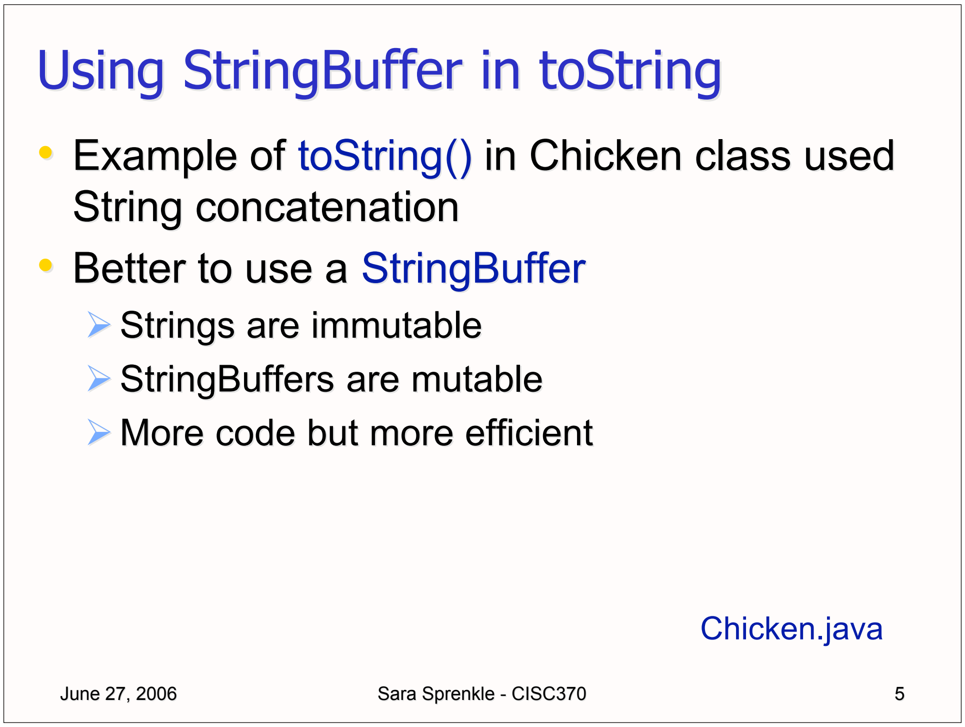
Announcements

- Changing offices:
  - Sara: Smith 447
  - Ke: Smith 440
- No office hours on Thursday--moving day!
  - CPM will be down at least Thursday, maybe longer
- Assignment 3: Printed Submission Changes
  - Do not print Javadocs, include link in README
  - Do not print Java files again if already in script file
  - Changes are reflected in assignment writeup
- Final Exam: Friday, August 11, 7-9 p.m.

Review

- Streams
- Collections

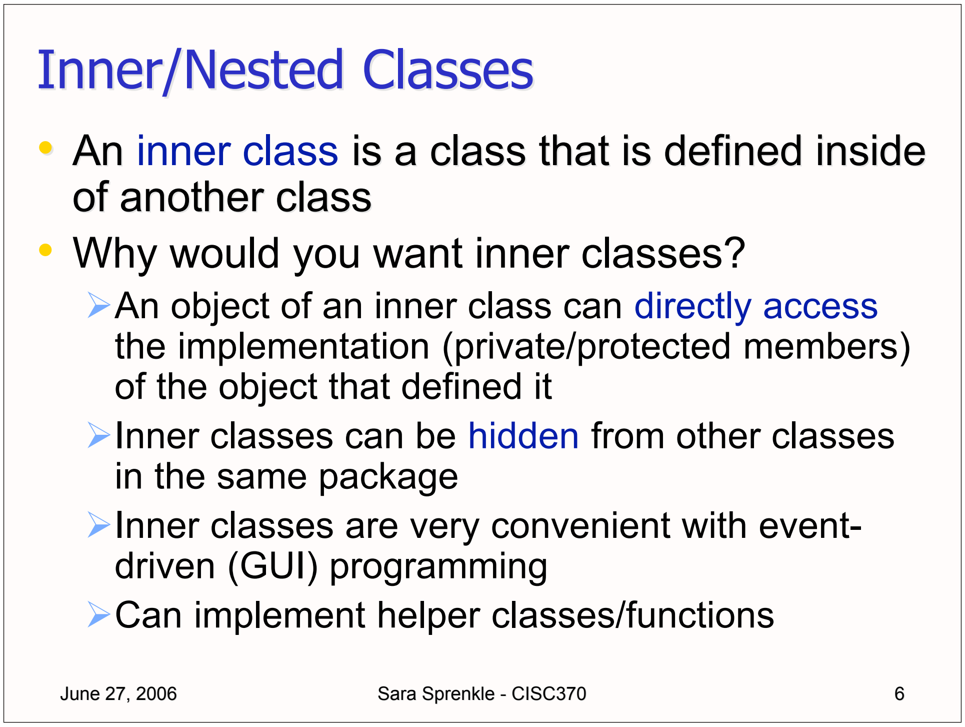
Quiz



Using StringBuffer in toString

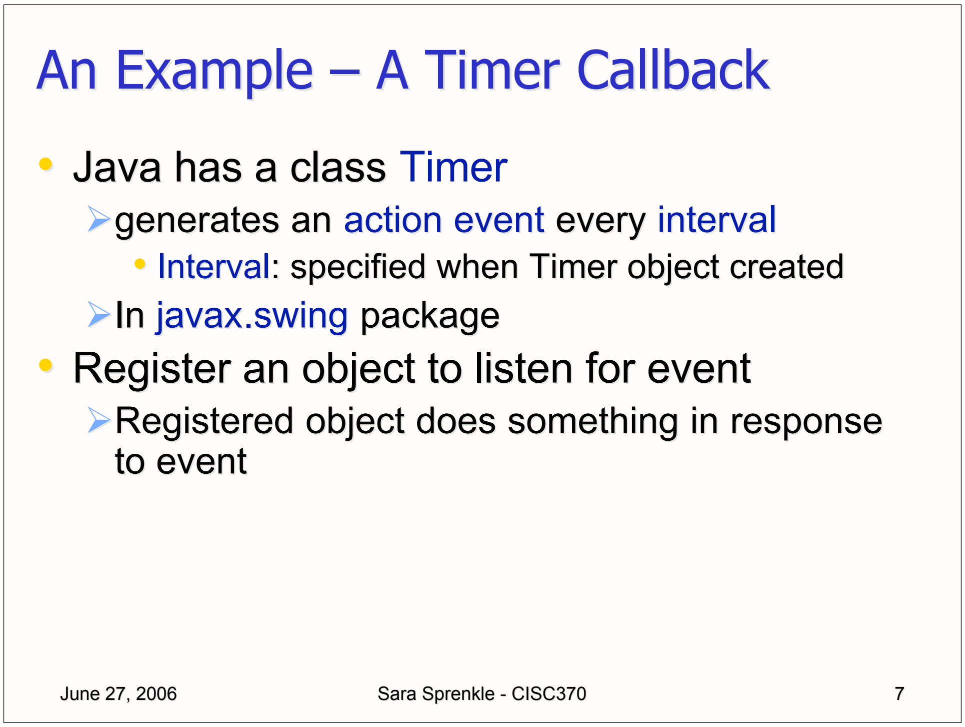
- Example of toString() in Chicken class used String concatenation
- Better to use a StringBuffer
  - Strings are immutable
  - StringBuffers are mutable
  - More code but more efficient

Chicken.java



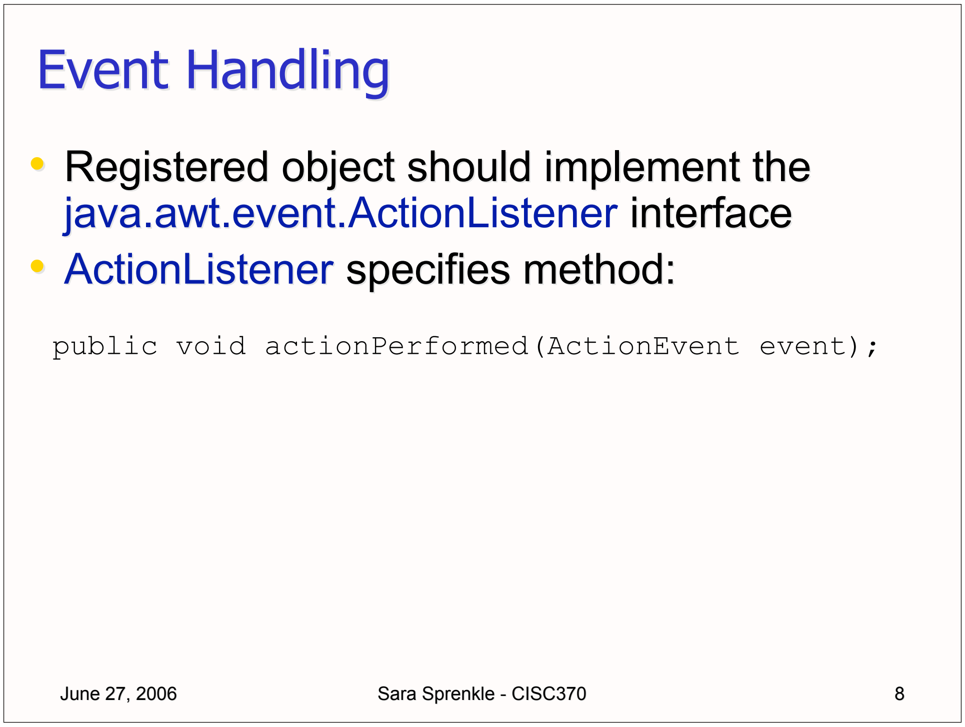
Inner/Nested Classes

- An inner class is a class that is defined inside of another class
- Why would you want inner classes?
  - An object of an inner class can directly access the implementation (private/protected members) of the object that defined it
  - Inner classes can be hidden from other classes in the same package
  - Inner classes are very convenient with event-driven (GUI) programming
  - Can implement helper classes/functions



An Example – A Timer Callback

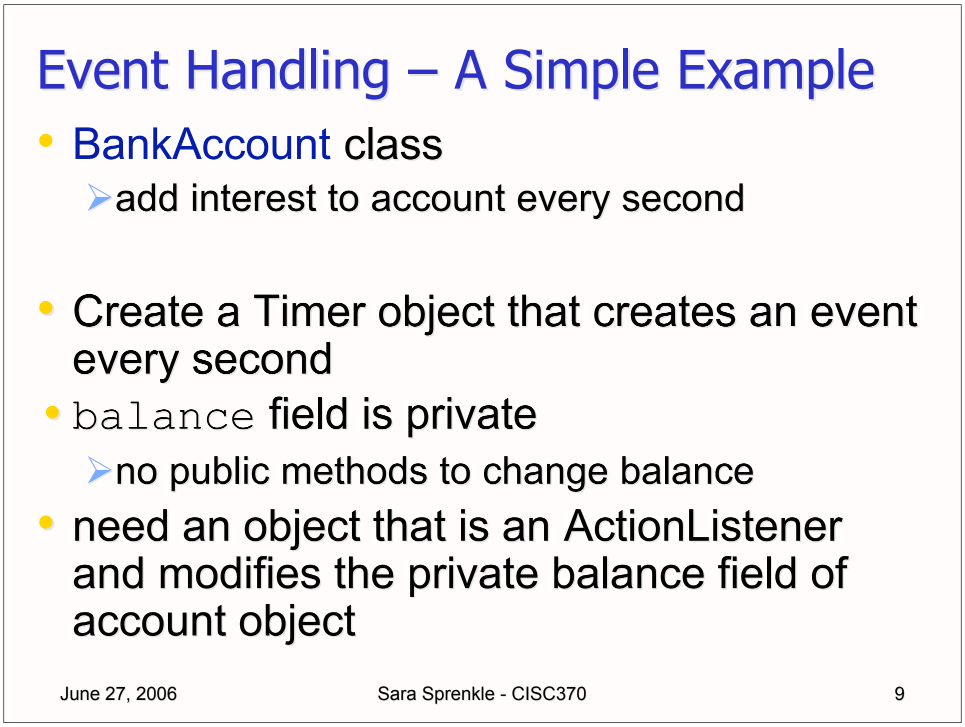
- Java has a class Timer
  - generates an action event every interval
    - Interval: specified when Timer object created
  - In javax.swing package
- Register an object to listen for event
  - Registered object does something in response to event



Event Handling

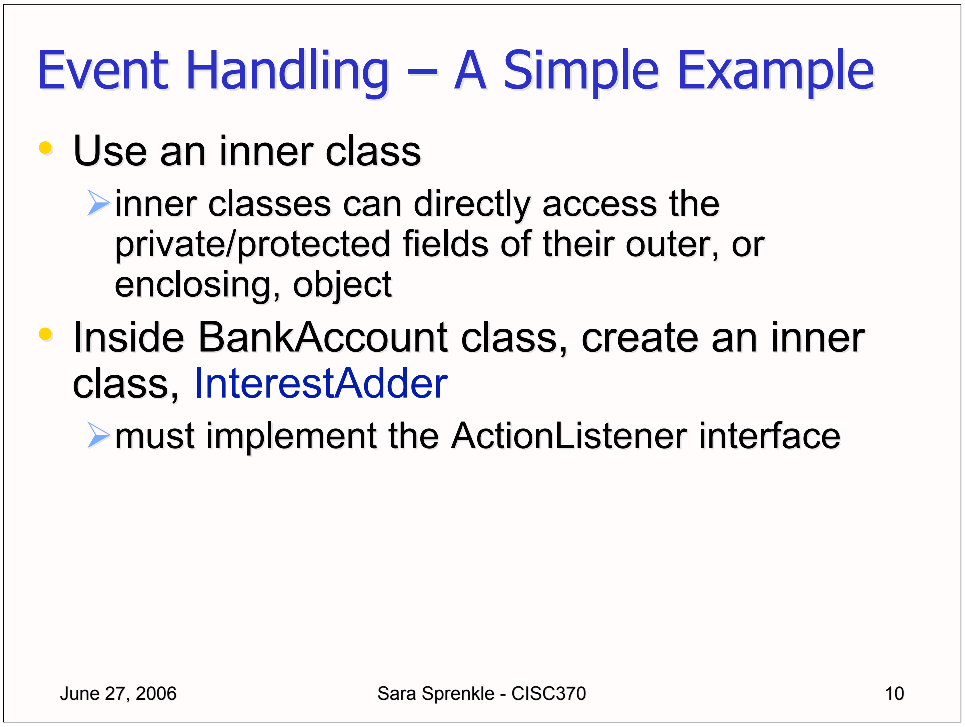
- Registered object should implement the java.awt.event.ActionListener interface
- ActionListener specifies method:

```
public void actionPerformed(ActionEvent event);
```



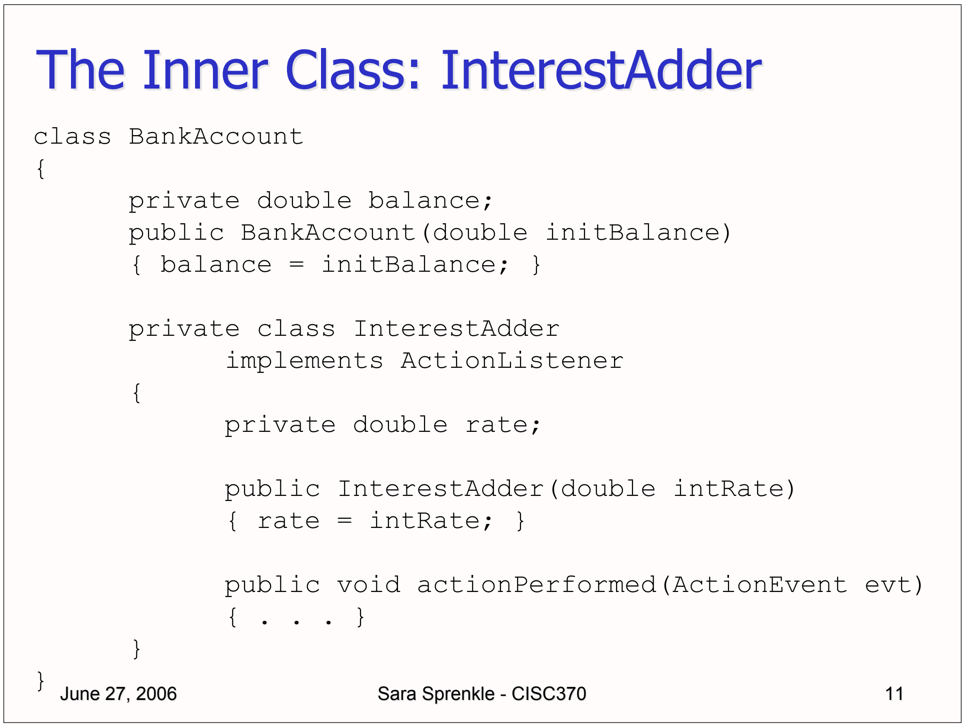
Event Handling – A Simple Example

- BankAccount class
  - add interest to account every second
- Create a Timer object that creates an event every second
- `balance` field is private
  - no public methods to change balance
- need an object that is an ActionListener and modifies the private balance field of account object



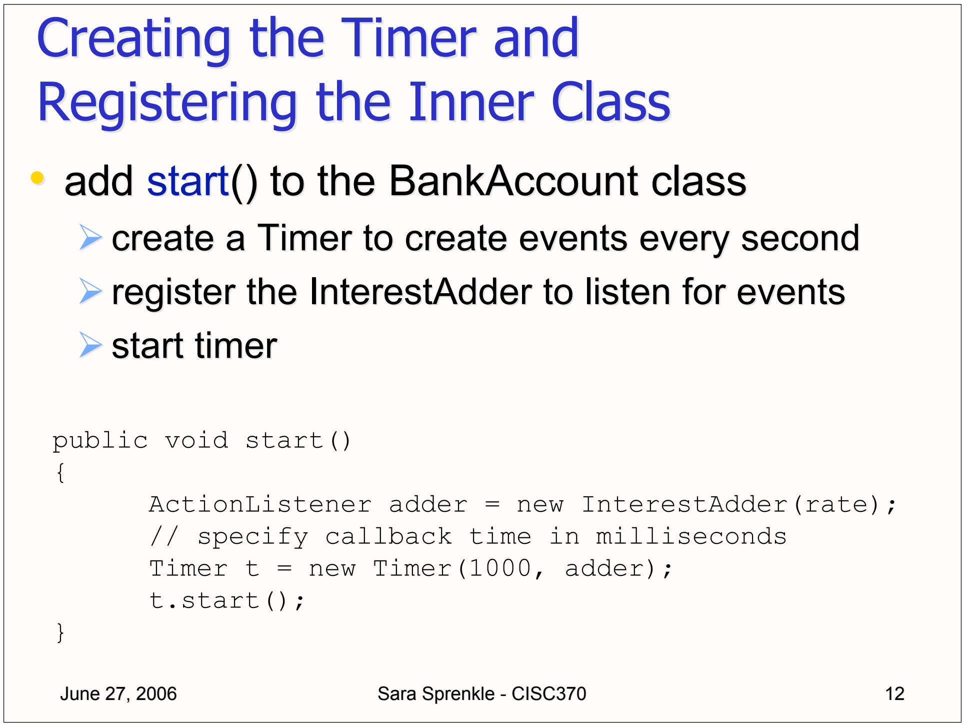
Event Handling – A Simple Example

- Use an inner class
  - inner classes can directly access the private/protected fields of their outer, or enclosing, object
- Inside BankAccount class, create an inner class, InterestAdder
  - must implement the ActionListener interface



The Inner Class: InterestAdder

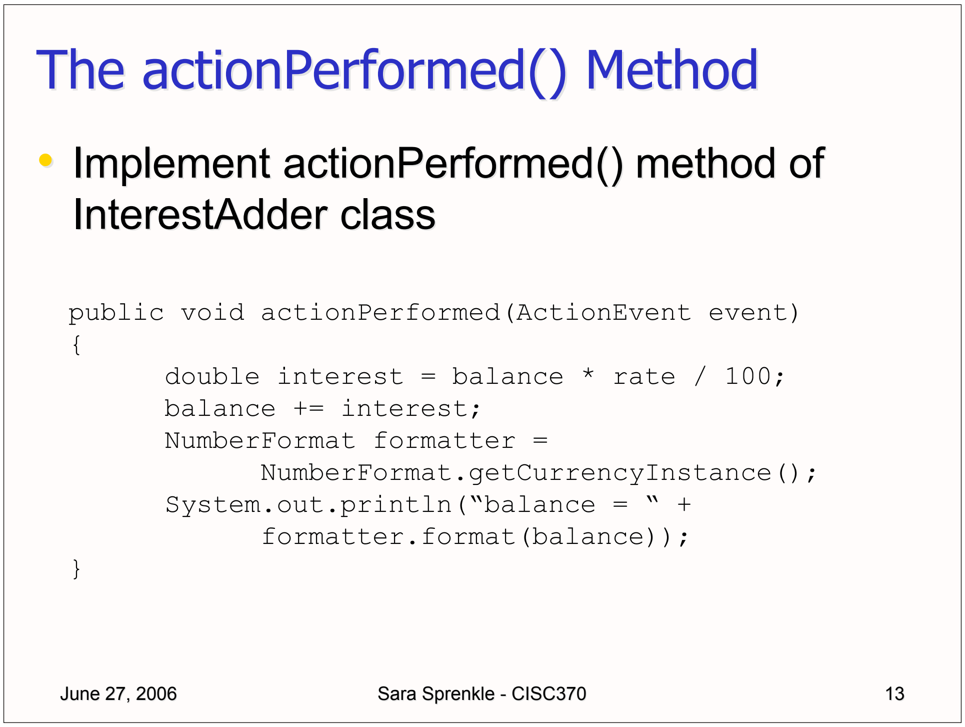
```
class BankAccount
{
      private double balance;
      public BankAccount(double initBalance)
      { balance = initBalance; }

      private class InterestAdder
            implements ActionListener
      {
            private double rate;

            public InterestAdder(double intRate)
            { rate = intRate; }

            public void actionPerformed(ActionEvent evt)
            { . . . }
      }
}
```



Creating the Timer and Registering the Inner Class

- add start() to the BankAccount class
  - create a Timer to create events every second
  - register the InterestAdder to listen for events
  - start timer

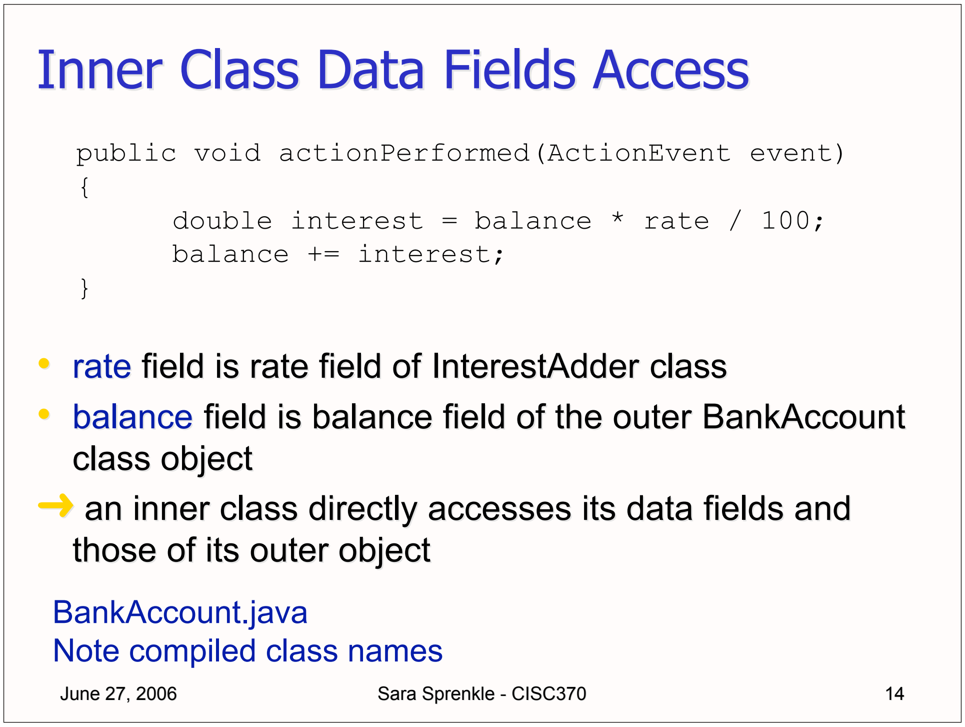
```
public void start()
{
      ActionListener adder = new InterestAdder(rate);
      // specify callback time in milliseconds
      Timer t = new Timer(1000, adder);
      t.start();
}
```



The actionPerformed() Method

- Implement actionPerformed() method of InterestAdder class

```
public void actionPerformed(ActionEvent event)
{
      double interest = balance * rate / 100;
      balance += interest;
      NumberFormat formatter =
            NumberFormat.getCurrencyInstance();
      System.out.println(“balance = “ +
            formatter.format(balance));
}
```



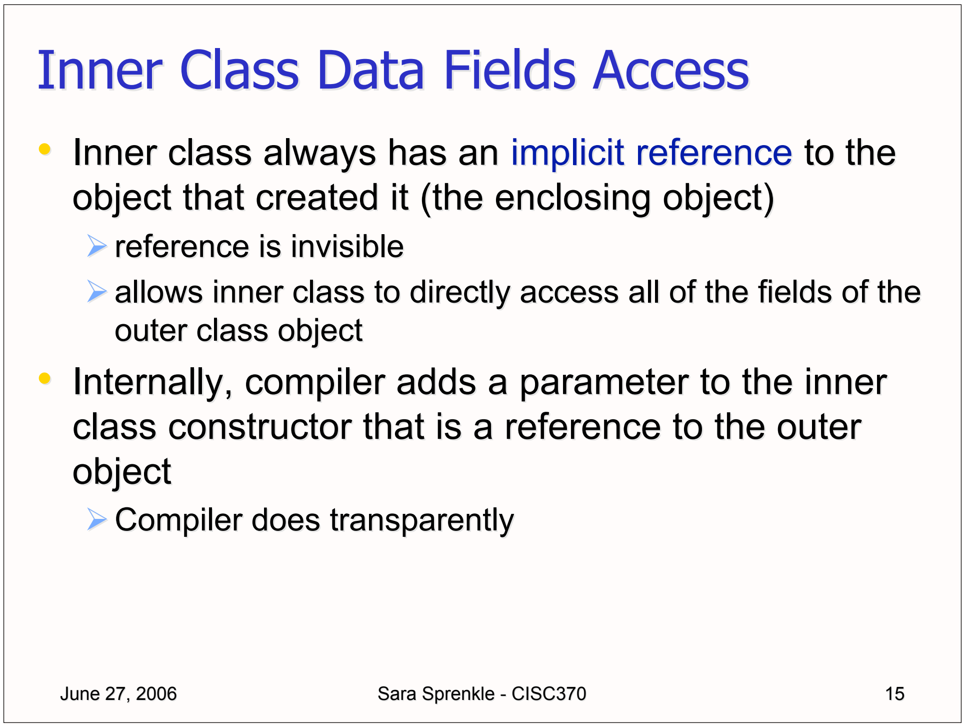
Inner Class Data Fields Access

```
public void actionPerformed(ActionEvent event)
{
      double interest = balance * rate / 100;
      balance += interest;
}
```

- rate field is rate field of InterestAdder class
- balance field is balance field of the outer BankAccount class object

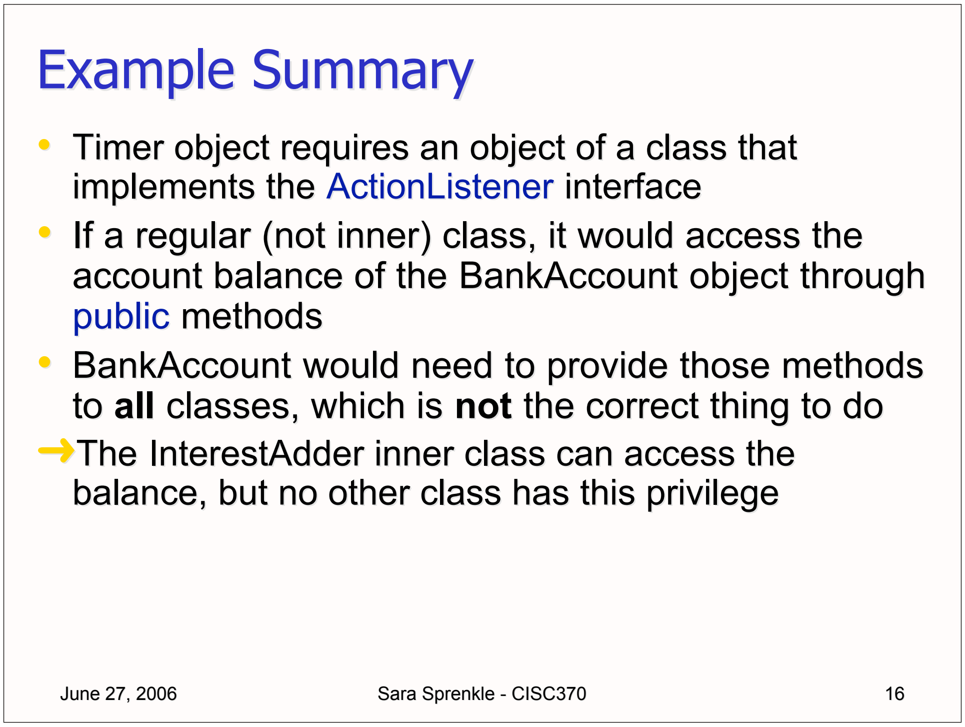
➜ an inner class directly accesses its data fields and those of its outer object

BankAccount.java
Note compiled class names



Inner Class Data Fields Access

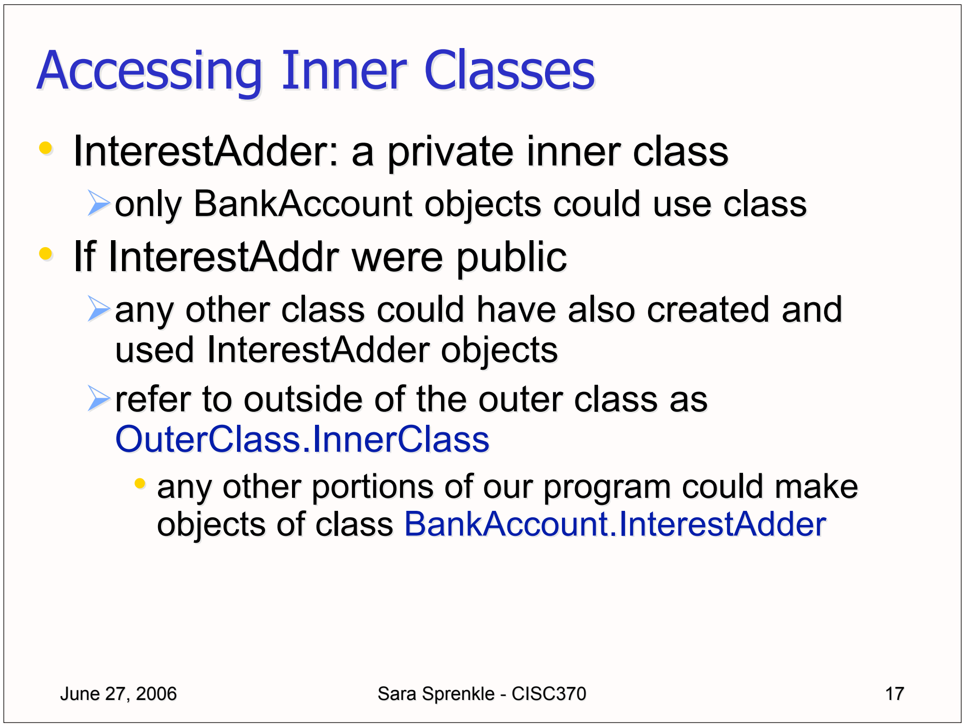
- Inner class always has an implicit reference to the object that created it (the enclosing object)
  - reference is invisible
  - allows inner class to directly access all of the fields of the outer class object
- Internally, compiler adds a parameter to the inner class constructor that is a reference to the outer object
  - Compiler does transparently



Example Summary

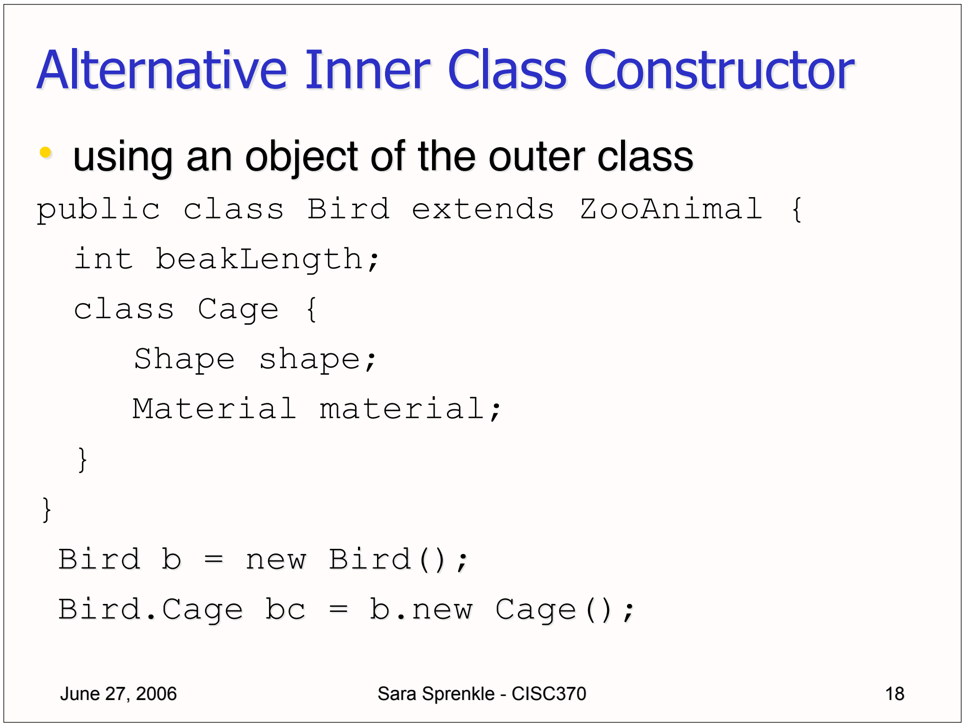
- Timer object requires an object of a class that implements the ActionListener interface
- If a regular (not inner) class, it would access the account balance of the BankAccount object through public methods
- BankAccount would need to provide those methods to **all** classes, which is **not** the correct thing to do

➜The InterestAdder inner class can access the balance, but no other class has this privilege



Accessing Inner Classes

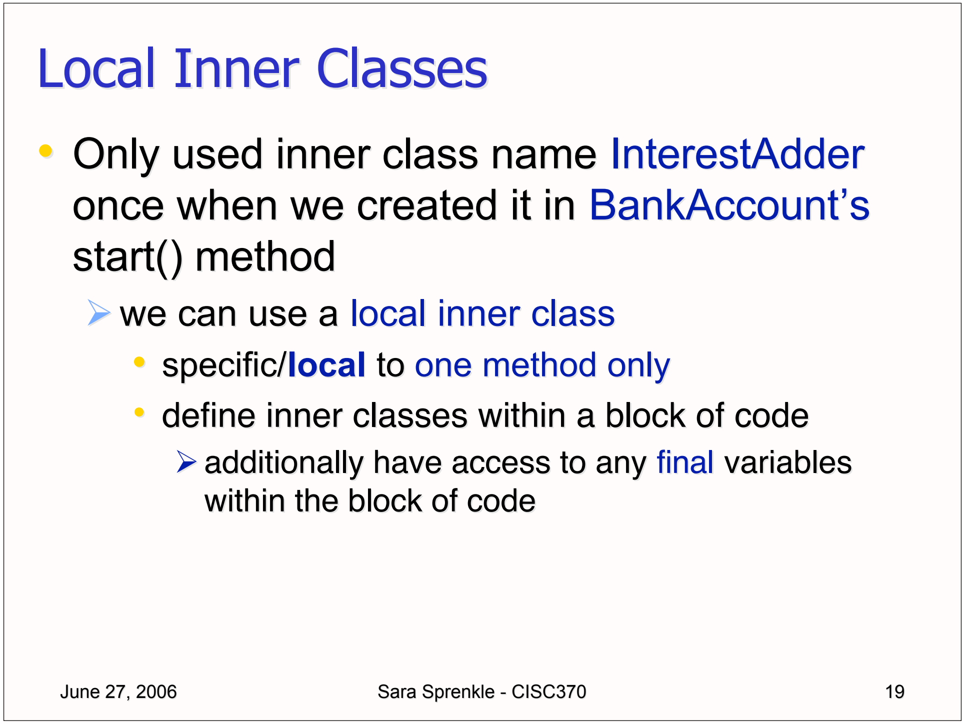
- InterestAdder: a private inner class
  - only BankAccount objects could use class
- If InterestAddr were public
  - any other class could have also created and used InterestAdder objects
  - refer to outside of the outer class as OuterClass.InnerClass
    - any other portions of our program could make objects of class BankAccount.InterestAdder



Alternative Inner Class Constructor

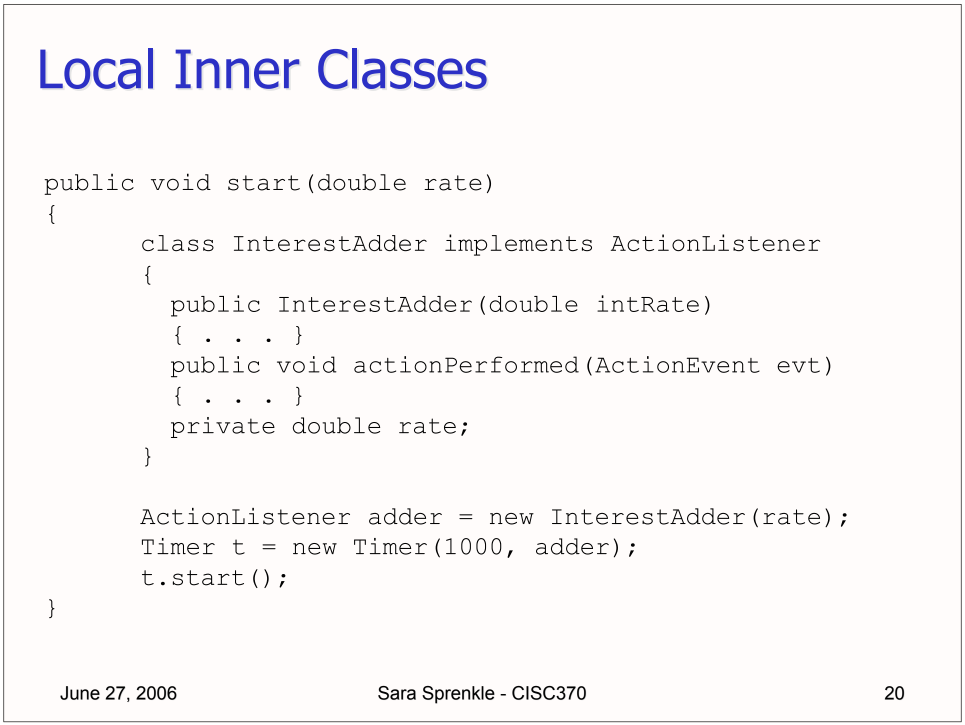
- using an object of the outer class

```
public class Bird extends ZooAnimal {
  int beakLength;
  class Cage {
     Shape shape;
     Material material;
  }
}
 Bird b = new Bird();
 Bird.Cage bc = b.new Cage();
```



Local Inner Classes

- Only used inner class name InterestAdder once when we created it in BankAccount's start() method
  - we can use a local inner class
    - specific/**local** to one method only
    - define inner classes within a block of code
      - additionally have access to any final variables within the block of code



Local Inner Classes

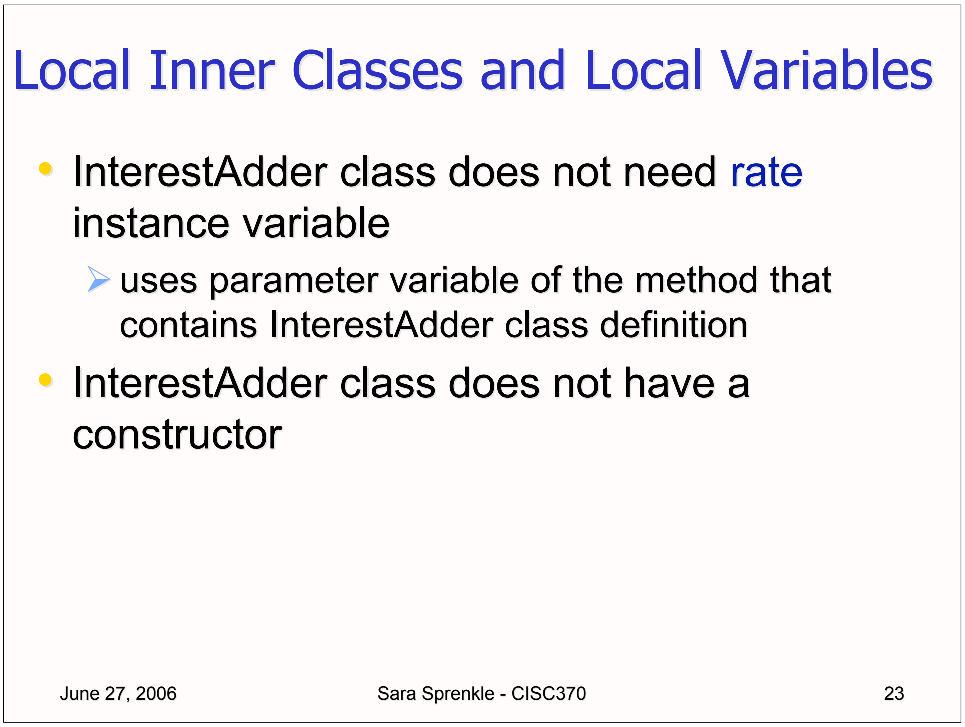
```
public void start(double rate)
{
      class InterestAdder implements ActionListener
      {
        public InterestAdder(double intRate)
        { . . . }
        public void actionPerformed(ActionEvent evt)
        { . . . }
        private double rate;
      }

      ActionListener adder = new InterestAdder(rate);
      Timer t = new Timer(1000, adder);
      t.start();
}
```

Local Inner Classes

- Local classes do not have an access modifier
  - automatically restricted to the block (method) in which they are defined
- InterestAdder class is completely hidden from outside world
  - no method besides start() knows about the class
- We could also change the local inner class to access the local variables on their enclosing method
  - must make such variables final first

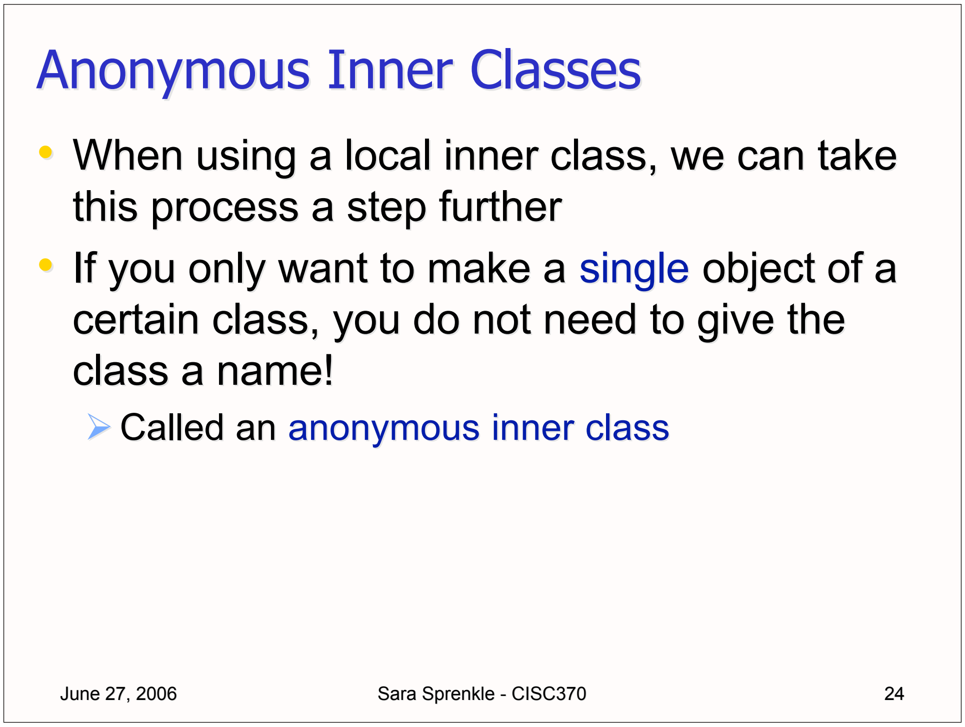
Local Inner Classes and Local Variables

```
public void start(final double rate)
{
      // local to start method
      class InterestAdder implements ActionListener
      {
        public void actionPerformed(ActionEvent evt)
        {
            double interest = balance * rate / 100;
            balance += interest;
        }
      }

      ActionListener adder = new InterestAdder();
      Timer t = new Timer(1000, adder);
      t.start();
}
```



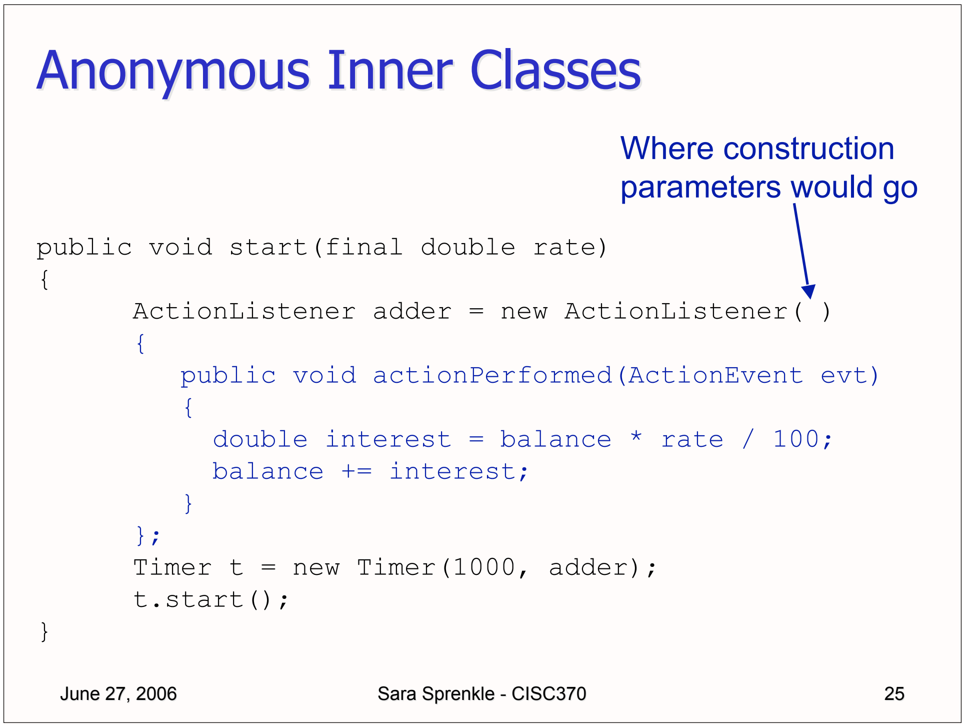
Local Inner Classes and Local Variables

- InterestAdder class does not need rate instance variable
  - uses parameter variable of the method that contains InterestAdder class definition
- InterestAdder class does not have a constructor



Anonymous Inner Classes

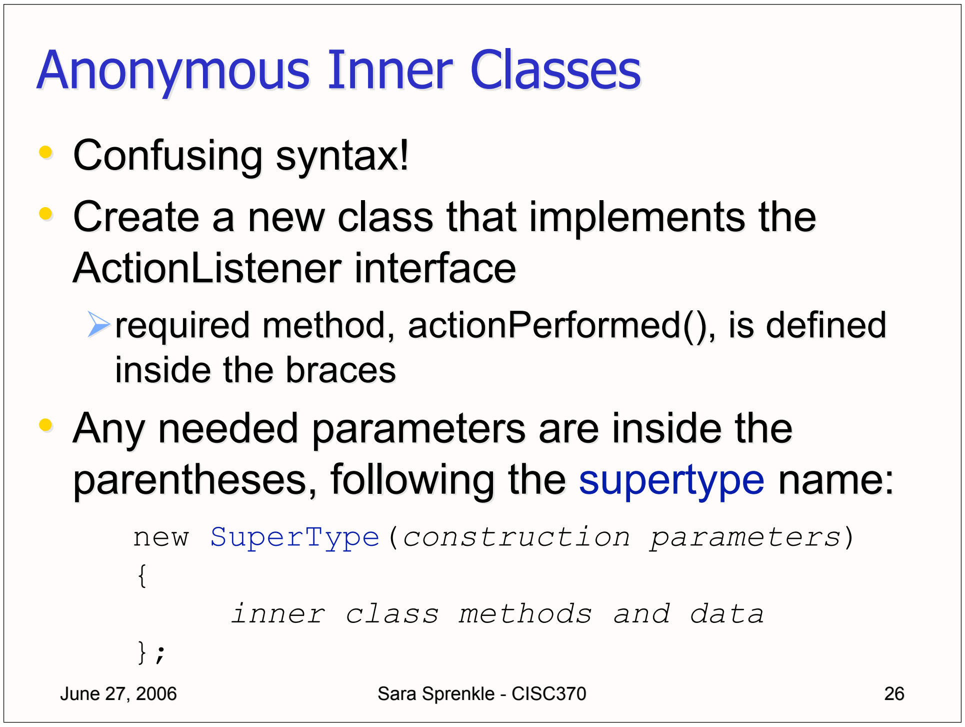
- When using a local inner class, we can take this process a step further
- If you only want to make a single object of a certain class, you do not need to give the class a name!
  - Called an anonymous inner class



Anonymous Inner Classes

Where construction parameters would go

```
public void start(final double rate)
{
       ActionListener adder = new ActionListener( )
       {
          public void actionPerformed(ActionEvent evt)
          {
            double interest = balance * rate / 100;
            balance += interest;
          }
       };
       Timer t = new Timer(1000, adder);
       t.start();
}
```



Anonymous Inner Classes

- Confusing syntax!
- Create a new class that implements the ActionListener interface
  - required method, actionPerformed(), is defined inside the braces
- Any needed parameters are inside the parentheses, following the supertype name:

```
new SuperType(construction parameters)
{
      inner class methods and data
};
```

Anonymous Inner Classes

- Supertype can be an interface or a class
  - If an interface, inner class implements the interface and required methods
  - If a class, the inner class extends that class
- Anonymous inner classes do **not** have constructors
  - Parameters are passed to the superclass constructor
  - If your inner class implements an interface, rather than extend a class, you cannot have construction parameters

Anonymous Inner Classes

- Carefully differentiate between
  - construction of a new object of a class
  - construction of an object of an anonymous inner class that extends that class…

```
// this is a Person object
Person queen = new Person(“Mary”);

// this is an object of an anonymous
// inner class extending the Person class
Person count = new Person(“Dracula”) {. . .};
```

Static Inner Classes

- If inner classes are declared as static, they do not have the implicit reference to an instance of the outer class
  - Not associated with an instance of the enclosing class

```
Bird.Cage bc = Bird.Cage()
```

- Useful for grouping classes, similar to packages

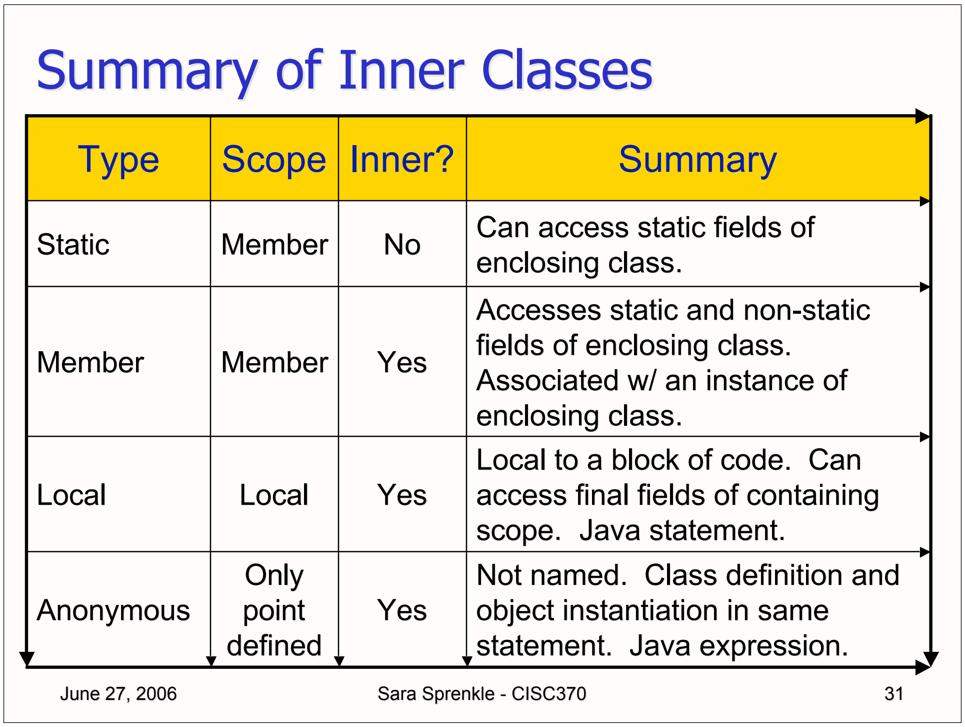
Enums

- More powerful than enums in C
  - New to Java 1.5
- Enums in Java are like inner class declarations

```
enum Color { Red, Yellow, Green };
Color current = Color.Red;
```

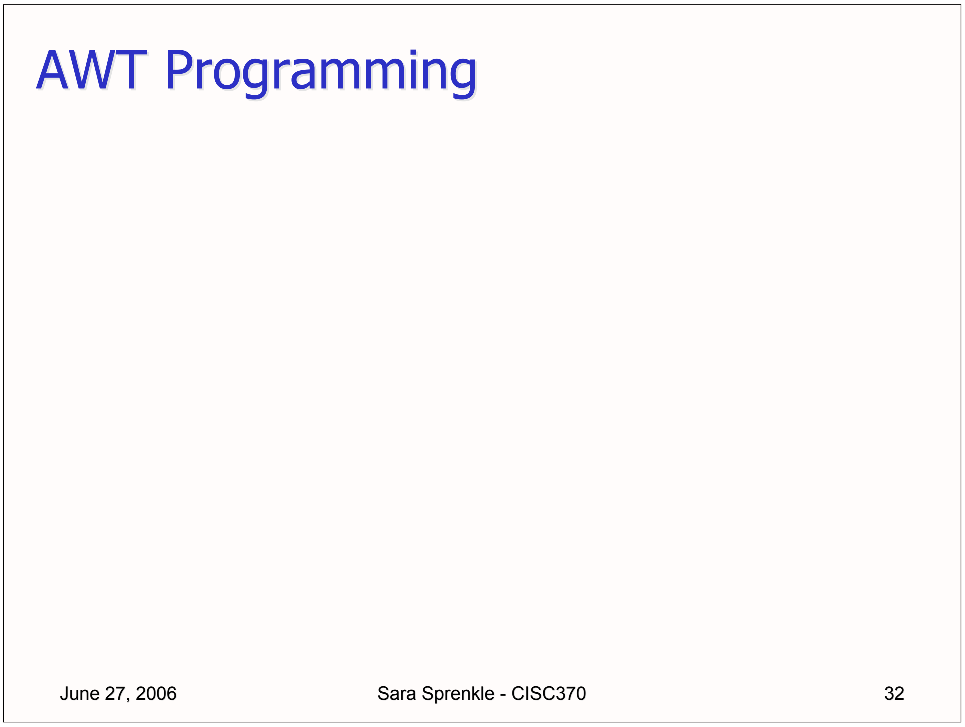
- Has static values() method
  - Returns array of values in order declared
- Can add functionality to enum
- Can be used in switch statements

cards.Card.java
cards.Deal.java
Planet.java
Operation.java



Summary of Inner Classes

Type	Scope	Inner?	Summary
Static	Member	No	Can access static fields of enclosing class.
Member	Member	Yes	Accesses static and non-static fields of enclosing class. Associated w/ an instance of enclosing class.
Local	Local	Yes	Local to a block of code. Can access final fields of containing scope. Java statement.
Anonymous	Only point defined	Yes	Not named. Class definition and object instantiation in same statement. Java expression.



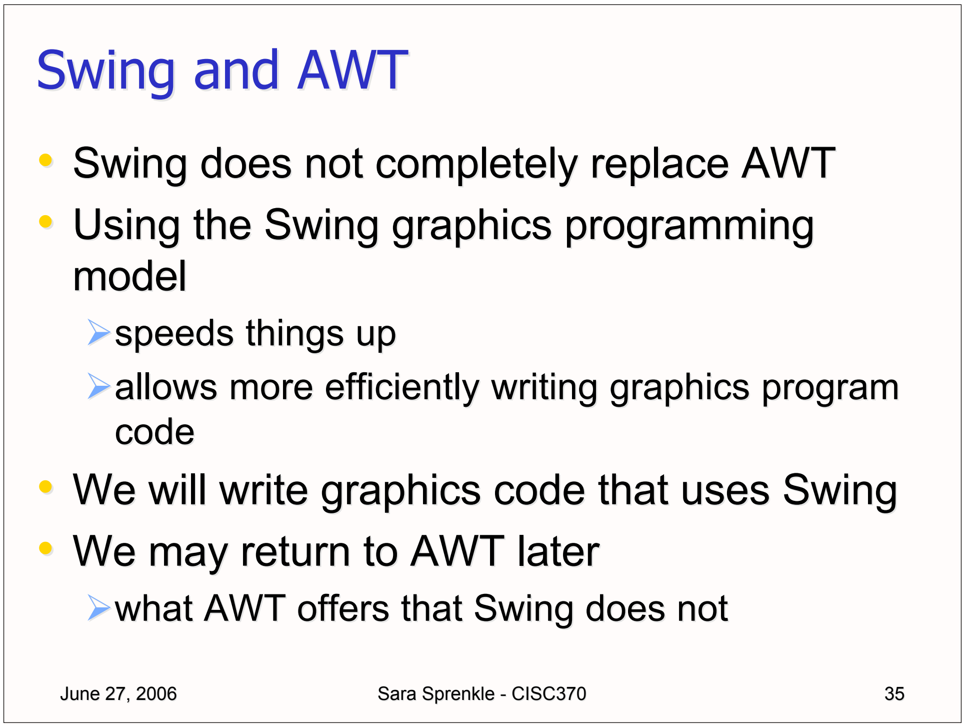
AWT Programming

AWT Programming

- Prior to Java 2 (version 1.2), all graphics programming was done with the Abstract Window Toolkit (AWT)
- AWT relies on peer entities to draw its graphics components
  - e.g., an AWT window maps to a system peer window (a AWT window maps to a Windows or X-Windows window)
- Operating system draws the peer entity based on what is in the AWT entity

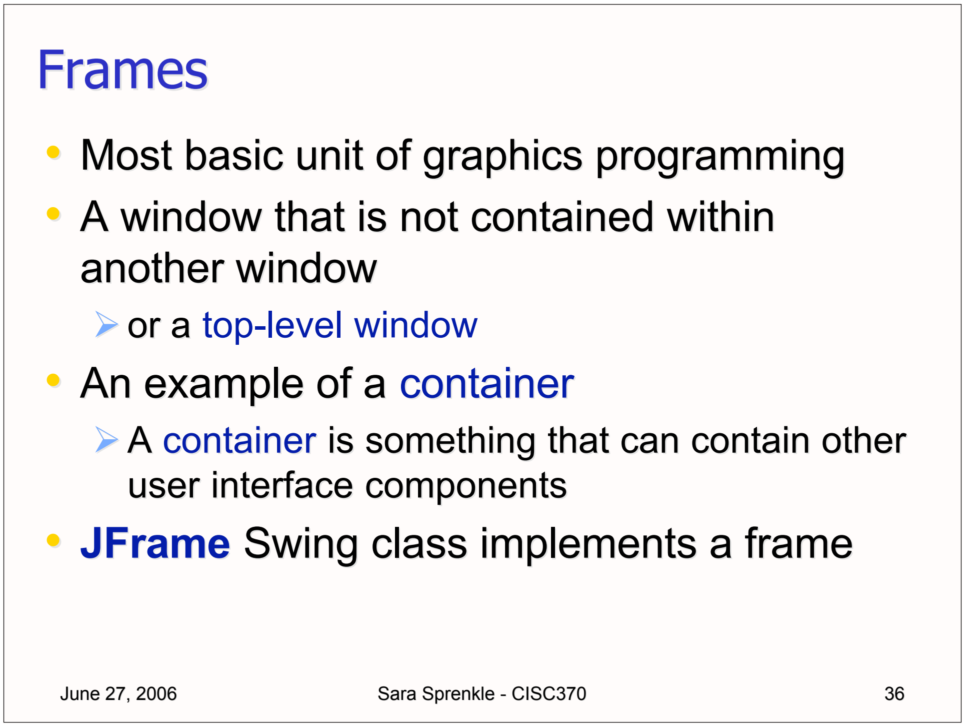
AWT Programming

- Drawing peer entities is a very slow process
- A bug in the peer graphics code, e.g., as the AWT code that makes the window entity, could cause hard-to-reproduce and platform-dependent errors
- Java 2 introduced Swing, javax.swing
- Swing still uses AWT frames
  - directly draws on them
  - operating system does not
  - Makes graphics process platform-independent
  - Improves speed



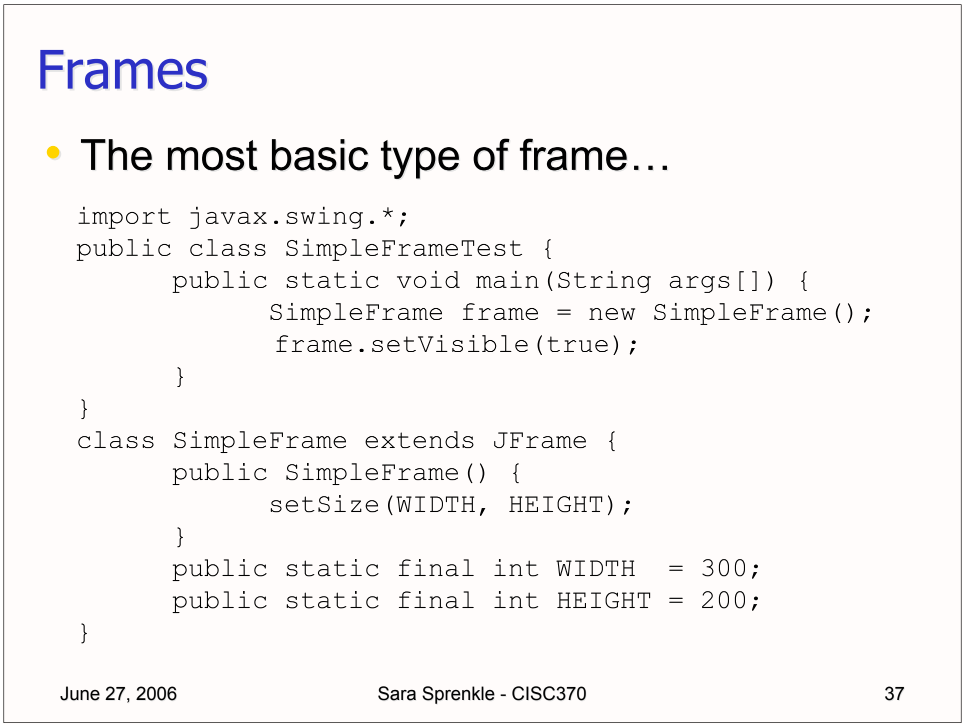
Swing and AWT

- Swing does not completely replace AWT
- Using the Swing graphics programming model
  - ➢speeds things up
  - ➢allows more efficiently writing graphics program code
- We will write graphics code that uses Swing
- We may return to AWT later
  - ➢what AWT offers that Swing does not



Frames

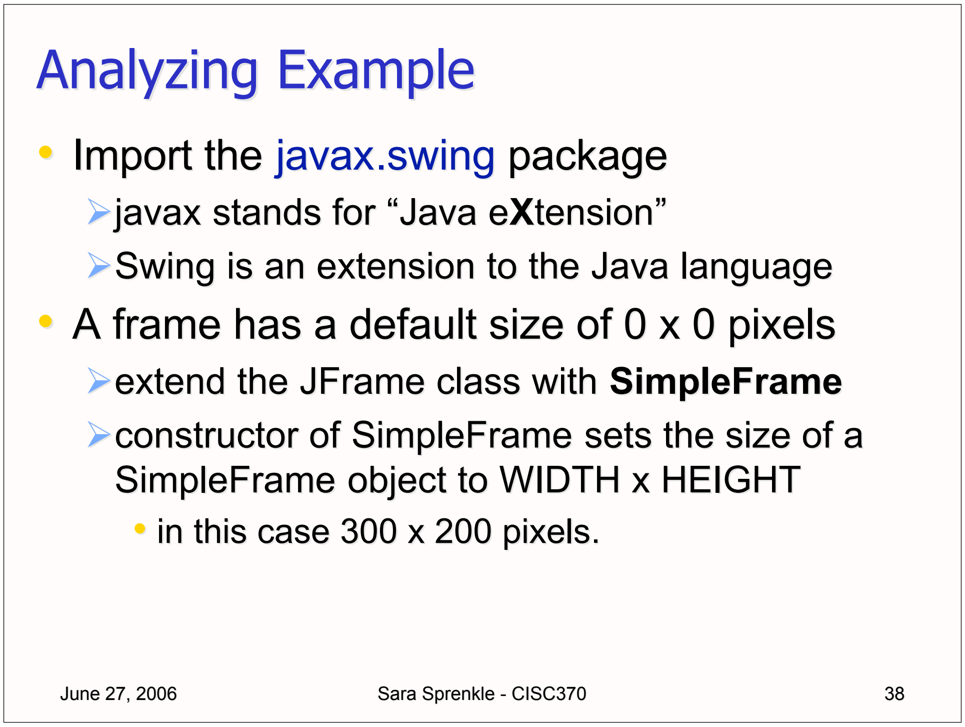
- Most basic unit of graphics programming
- A window that is not contained within another window
  - ➢or a top-level window
- An example of a container
  - ➢A container is something that can contain other user interface components
- **JFrame** Swing class implements a frame



Frames

- The most basic type of frame…

```
import javax.swing.*;
public class SimpleFrameTest {
        public static void main(String args[]) {
                SimpleFrame frame = new SimpleFrame();
                frame.setVisible(true);
        }
}
class SimpleFrame extends JFrame {
        public SimpleFrame() {
                setSize(WIDTH, HEIGHT);
        }
        public static final int WIDTH  = 300;
        public static final int HEIGHT = 200;
}
```



Analyzing Example

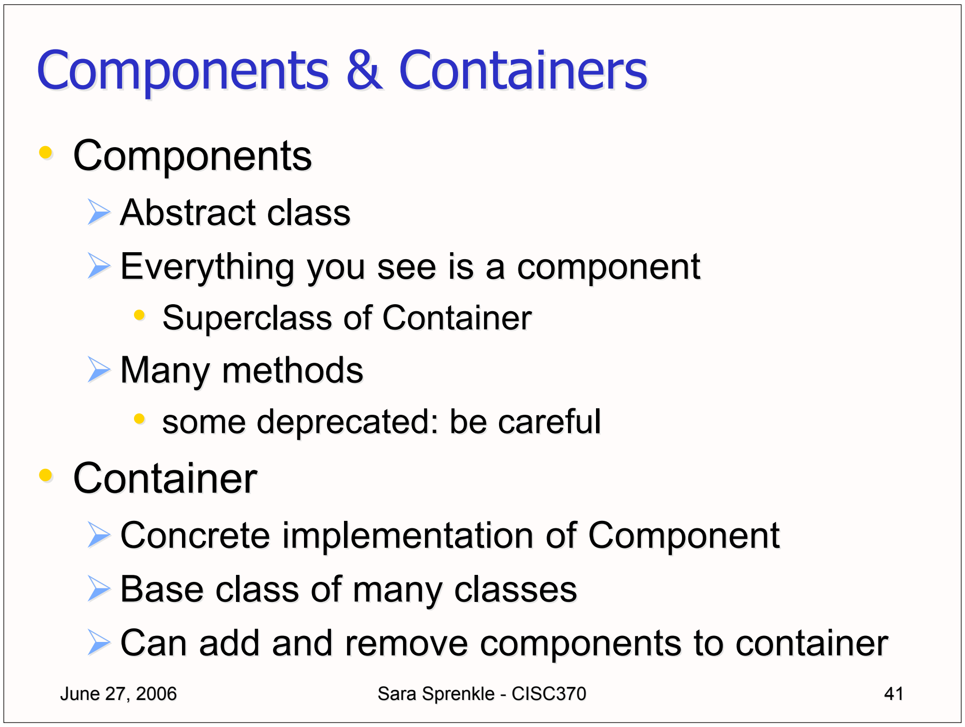
- Import the javax.swing package
  - javax stands for "Java e**X**tension"
  - Swing is an extension to the Java language
- A frame has a default size of 0 x 0 pixels
  - extend the JFrame class with **SimpleFrame**
  - constructor of SimpleFrame sets the size of a SimpleFrame object to WIDTH x HEIGHT
    - in this case 300 x 200 pixels.

Frames

- Inside the **SimpleFrameTest** class we create a new object of type **SimpleFrame**
- Creating a frame does not mean frame is displayed on screen
  - have to explicitly call setVisible(true) to have the system display the frame
- Call setVisible(true) in the method that creates the frame
  - e.g., the main() method of the test class

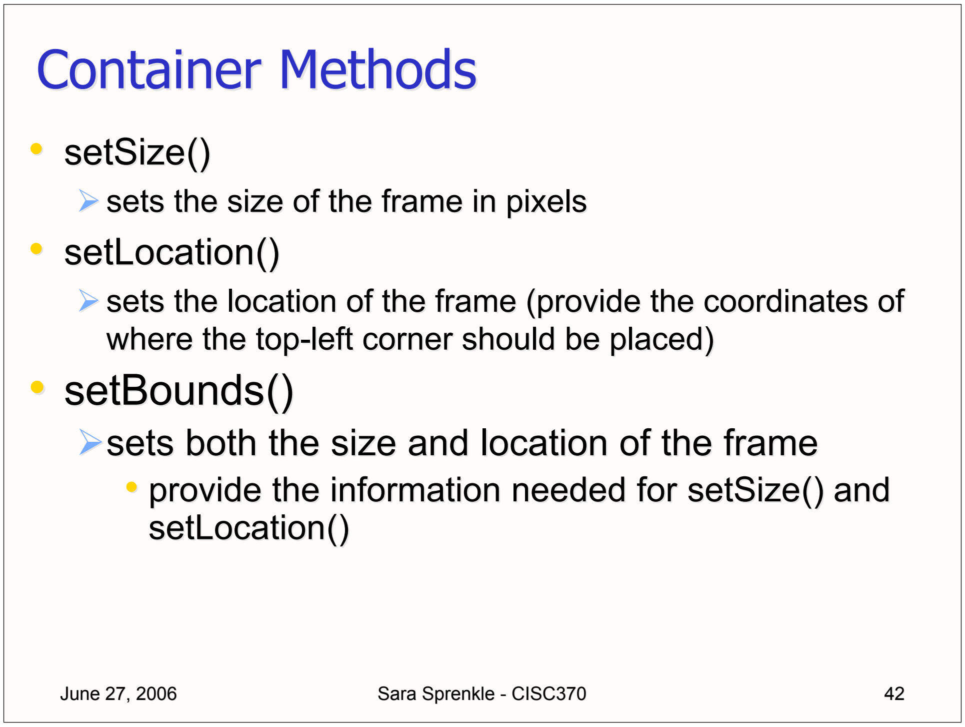
Frame Methods

- JFrame is derived from java.awt.Frame
  - Frame class is derived from Container class
  - Container: anything that can contain UI components
- JFrame object (or any class derived from a JFrame) has methods that are defined in JFrame, Frame, and Container classes
  - Can use these methods in any JFrame object



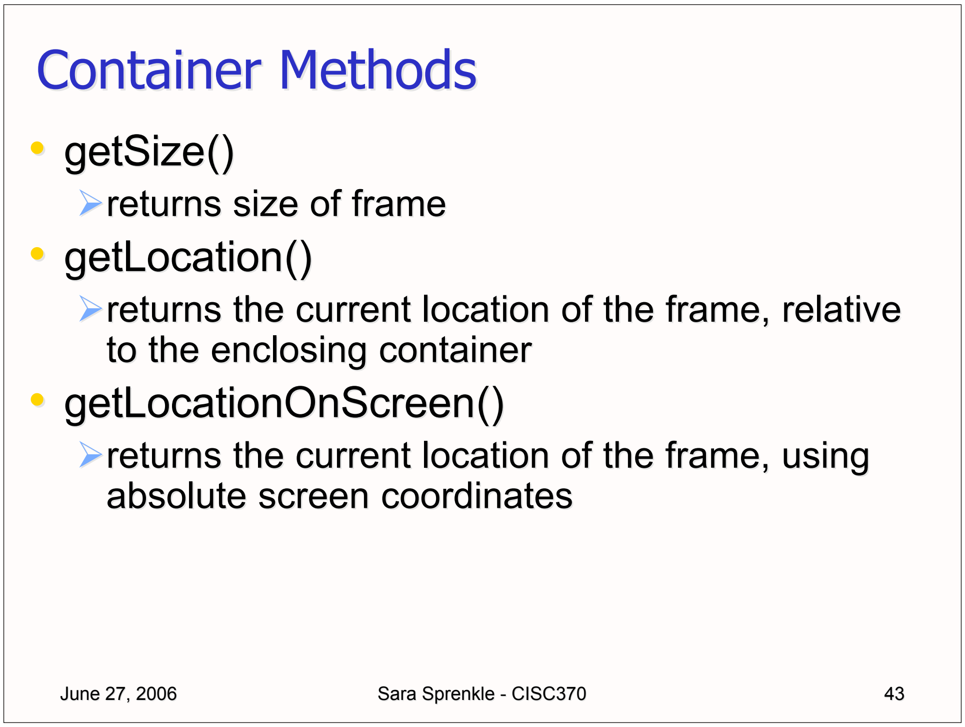
Components & Containers

- Components
  - Abstract class
  - Everything you see is a component
    - Superclass of Container
  - Many methods
    - some deprecated: be careful
- Container
  - Concrete implementation of Component
  - Base class of many classes
  - Can add and remove components to container



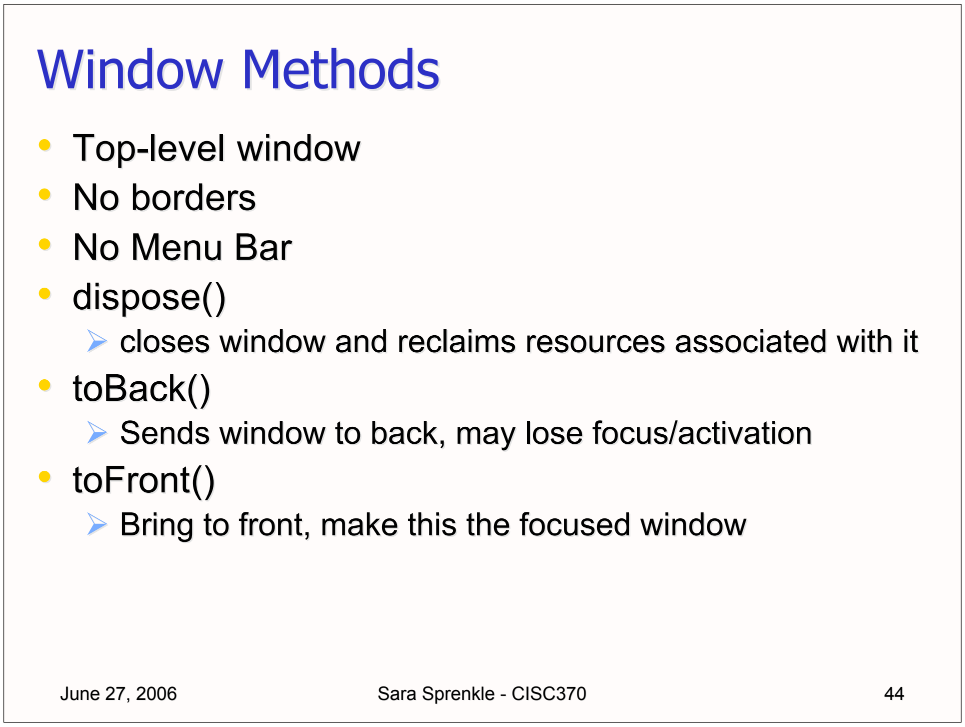
Container Methods

- setSize()
  - sets the size of the frame in pixels
- setLocation()
  - sets the location of the frame (provide the coordinates of where the top-left corner should be placed)
- setBounds()
  - sets both the size and location of the frame
    - provide the information needed for setSize() and setLocation()



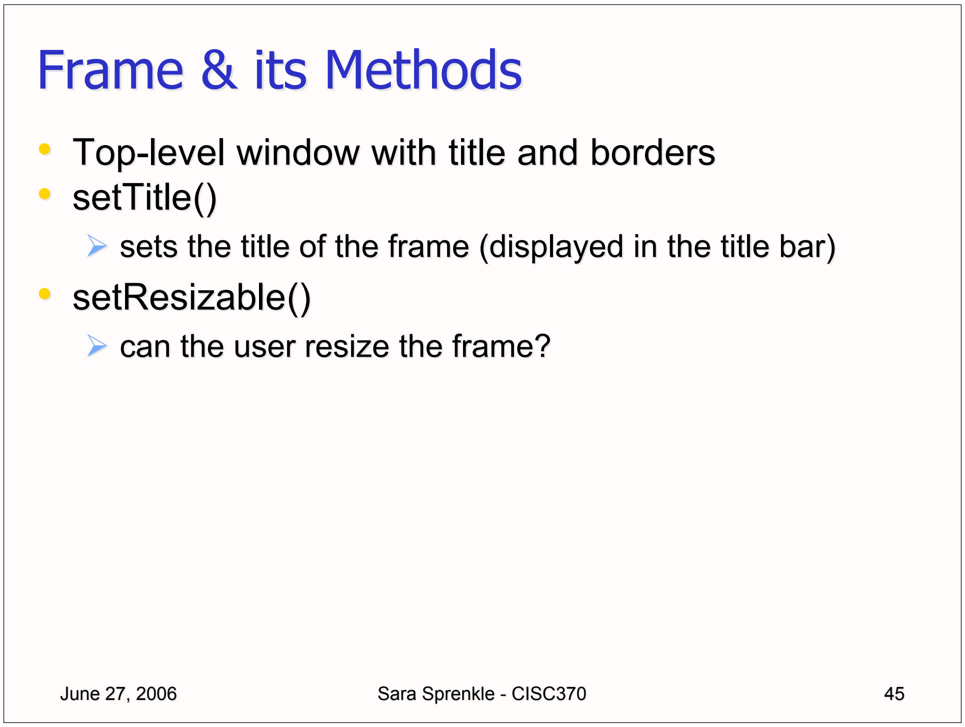
Container Methods

- getSize()
  - returns size of frame
- getLocation()
  - returns the current location of the frame, relative to the enclosing container
- getLocationOnScreen()
  - returns the current location of the frame, using absolute screen coordinates



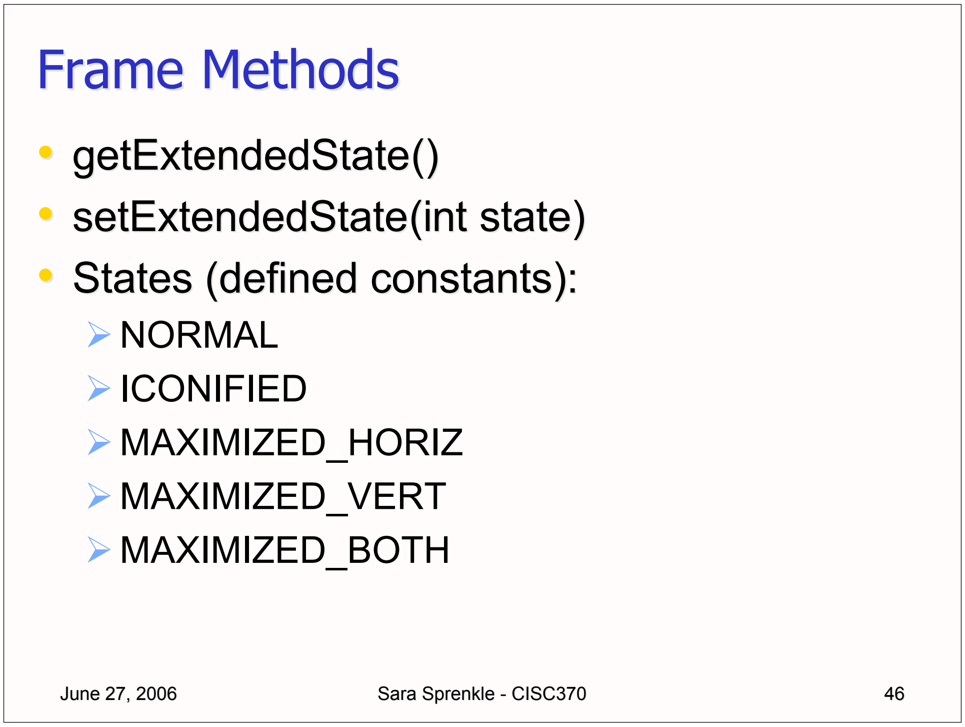
Window Methods

- Top-level window
- No borders
- No Menu Bar
- dispose()
  - closes window and reclaims resources associated with it
- toBack()
  - Sends window to back, may lose focus/activation
- toFront()
  - Bring to front, make this the focused window



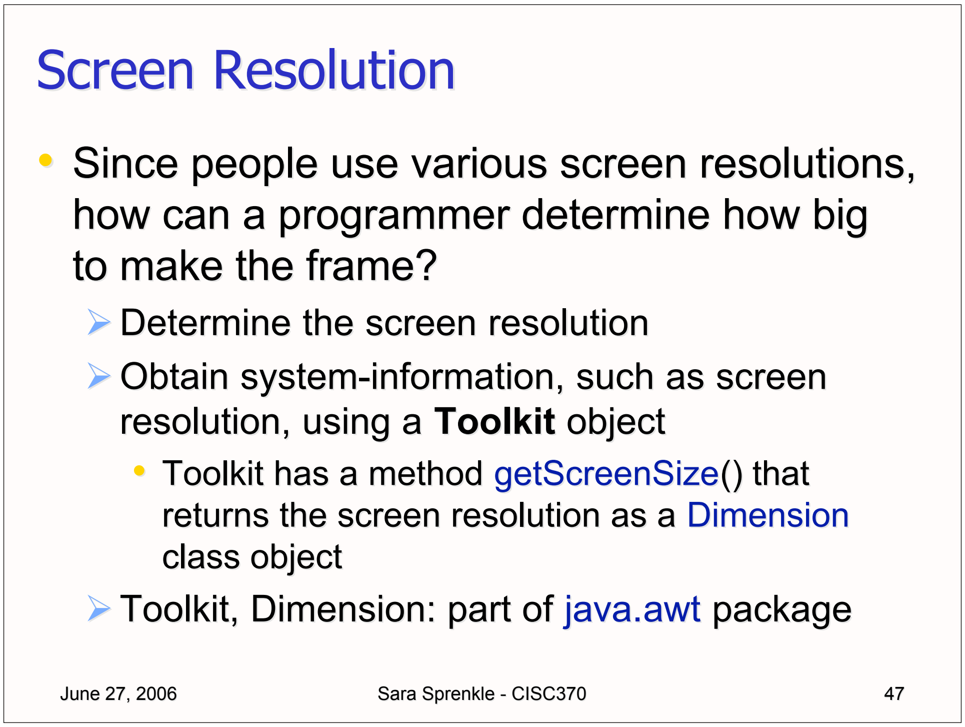
Frame & its Methods

- Top-level window with title and borders
- setTitle()
  - sets the title of the frame (displayed in the title bar)
- setResizable()
  - can the user resize the frame?



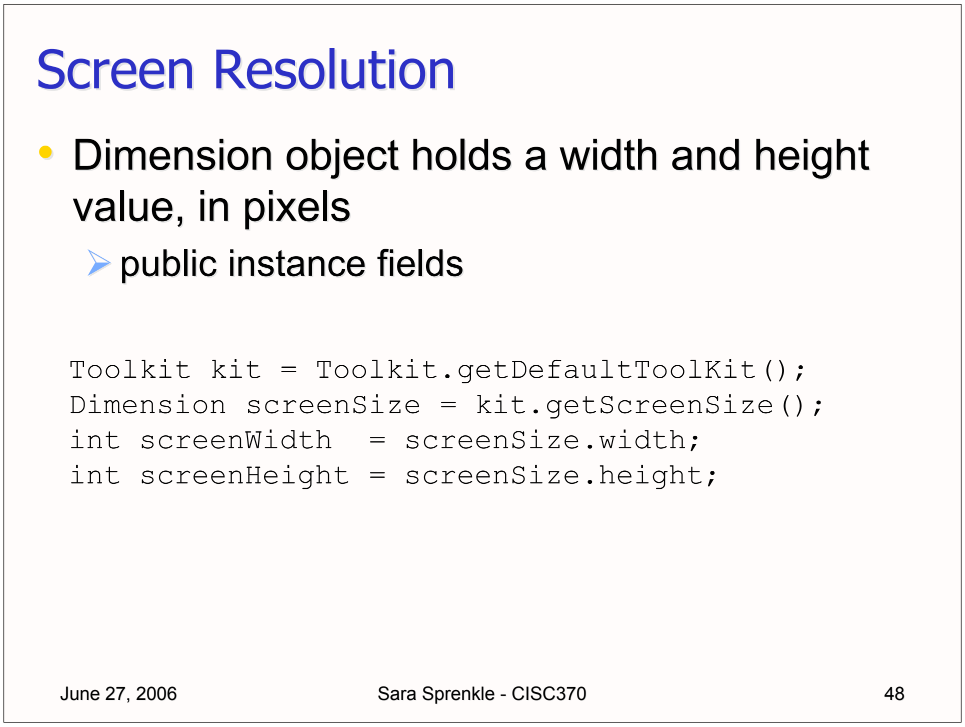
Frame Methods

- getExtendedState()
- setExtendedState(int state)
- States (defined constants):
  - NORMAL
  - ICONIFIED
  - MAXIMIZED_HORIZ
  - MAXIMIZED_VERT
  - MAXIMIZED_BOTH



Screen Resolution

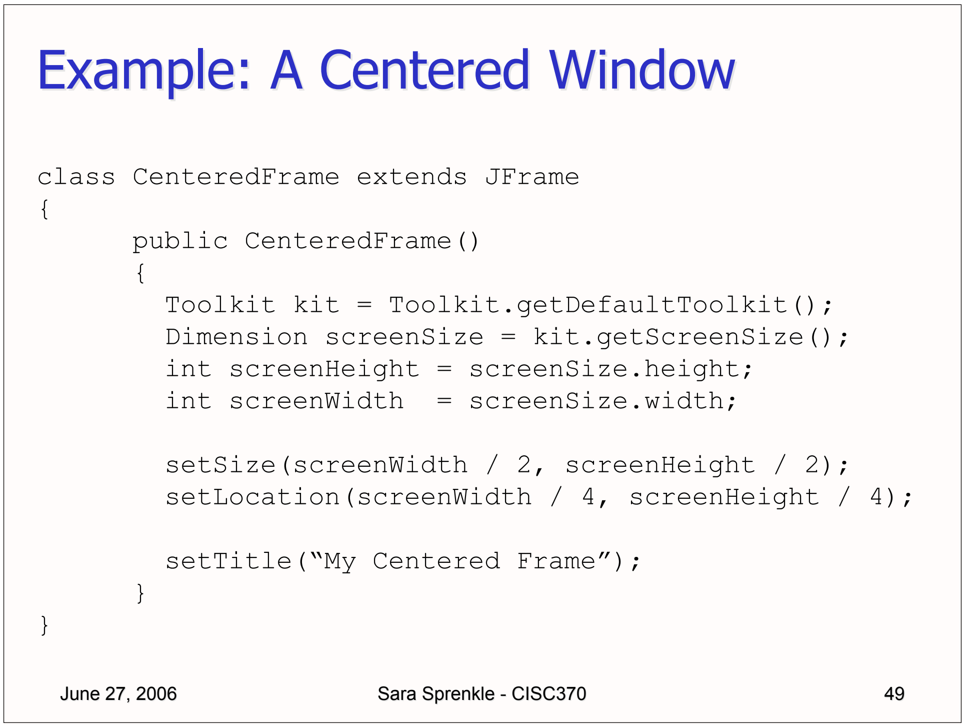
- Since people use various screen resolutions, how can a programmer determine how big to make the frame?
  - Determine the screen resolution
  - Obtain system-information, such as screen resolution, using a **Toolkit** object
    - Toolkit has a method getScreenSize() that returns the screen resolution as a Dimension class object
  - Toolkit, Dimension: part of java.awt package



Screen Resolution

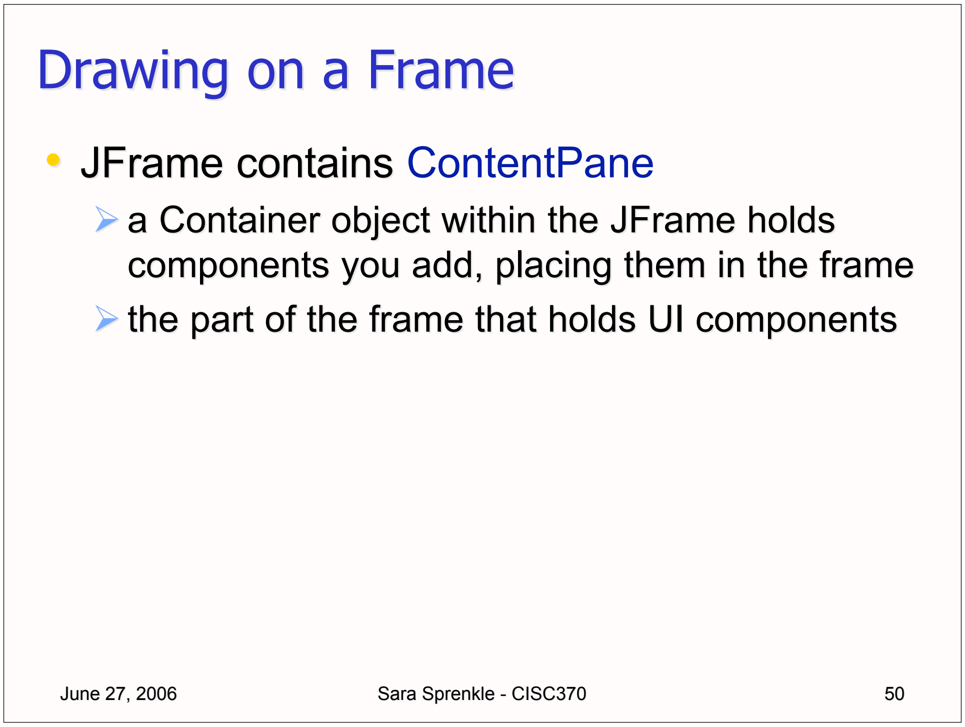
- Dimension object holds a width and height value, in pixels
  - public instance fields

```
Toolkit kit = Toolkit.getDefaultToolKit();
Dimension screenSize = kit.getScreenSize();
int screenWidth  = screenSize.width;
int screenHeight = screenSize.height;
```



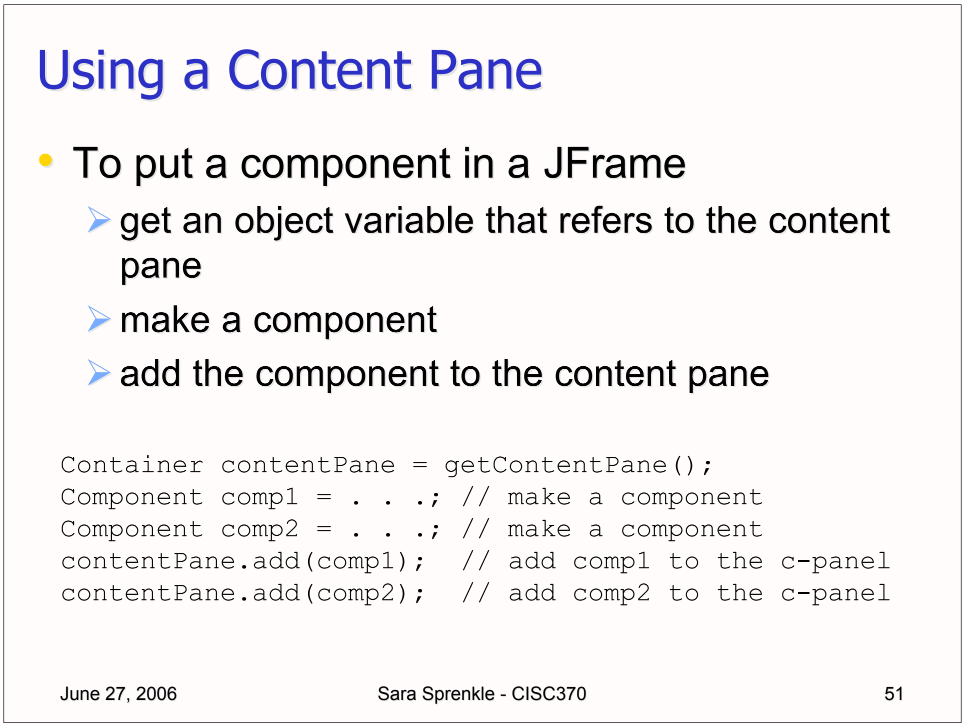
Example: A Centered Window

```
class CenteredFrame extends JFrame
{
      public CenteredFrame()
      {
        Toolkit kit = Toolkit.getDefaultToolkit();
        Dimension screenSize = kit.getScreenSize();
        int screenHeight = screenSize.height;
        int screenWidth  = screenSize.width;

        setSize(screenWidth / 2, screenHeight / 2);
        setLocation(screenWidth / 4, screenHeight / 4);

        setTitle("My Centered Frame");
      }
}
```



Drawing on a Frame

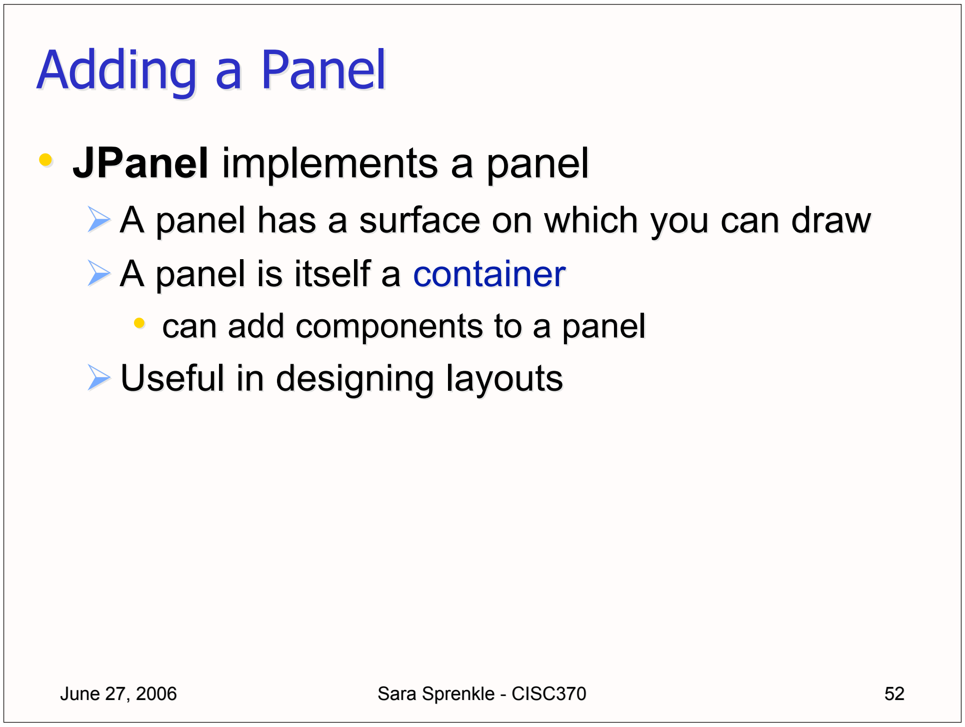
- JFrame contains ContentPane
  - a Container object within the JFrame holds components you add, placing them in the frame
  - the part of the frame that holds UI components



Using a Content Pane

- To put a component in a JFrame
  - get an object variable that refers to the content pane
  - make a component
  - add the component to the content pane

```
Container contentPane = getContentPane();
Component comp1 = . . .; // make a component
Component comp2 = . . .; // make a component
contentPane.add(comp1);  // add comp1 to the c-panel
contentPane.add(comp2);  // add comp2 to the c-panel
```



Adding a Panel

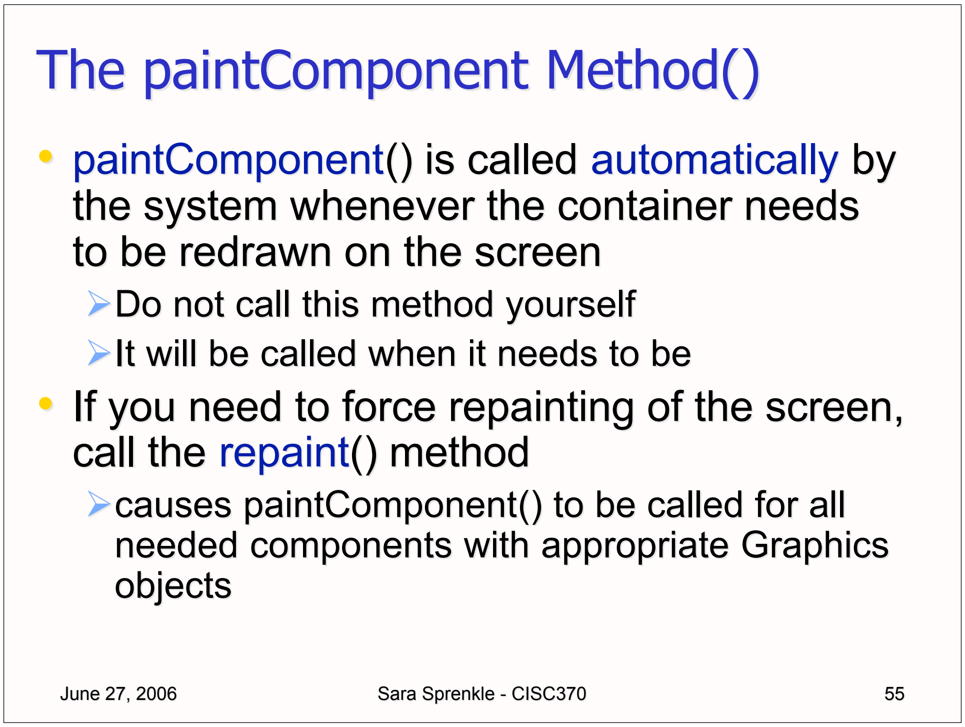
- **JPanel** implements a panel
  - A panel has a surface on which you can draw
  - A panel is itself a container
    - can add components to a panel
  - Useful in designing layouts

Drawing on a Panel

- To draw on a panel:
  - Define a new class that extends the JPanel class
  - Override the paintComponent() method in derived class
- paintComponent() method takes one parameter
  - an object of type **Graphics**
- Graphics object: a collection of settings for drawing images and text, such as colors and fonts
- All drawing in Java must go through a Graphics object

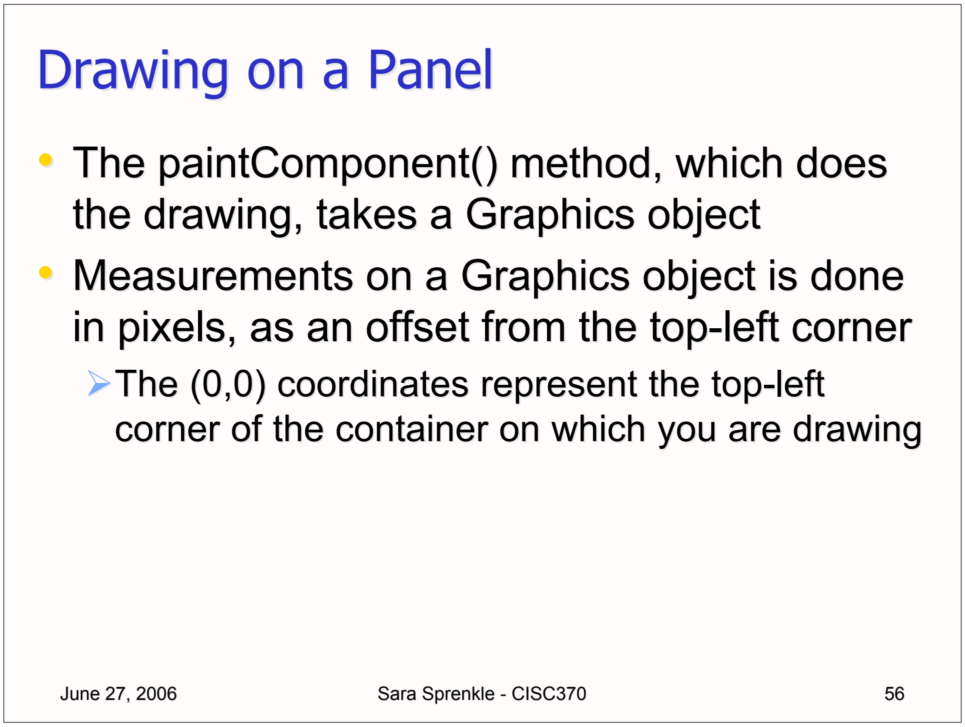
Drawing on a Panel

```
class MyPanel extends JPanel
{
        public void paintComponent(Graphics g)
        {

                // code for drawing goes here

        }
}
```



The paintComponent Method()

- paintComponent() is called automatically by the system whenever the container needs to be redrawn on the screen
  - Do not call this method yourself
  - It will be called when it needs to be
- If you need to force repainting of the screen, call the repaint() method
  - causes paintComponent() to be called for all needed components with appropriate Graphics objects



Drawing on a Panel

- The paintComponent() method, which does the drawing, takes a Graphics object
- Measurements on a Graphics object is done in pixels, as an offset from the top-left corner
  - The (0,0) coordinates represent the top-left corner of the container on which you are drawing

Drawing on a Panel

- Displaying text is a special type of drawing, called rendering text
- To render text on a panel, call drawString()

```
class HelloWorldPanel extends JPanel
{
      public static final int MESSAGE_X = 75;
      public static final int MESSAGE_Y = 100;

      public void paintComponent(Graphics g)
      {
            super.paintComponent(g);

            g.drawString("Hello World.",
                  MESSAGE_X, MESSAGE_Y);
      }
}
```

Drawing on a Panel

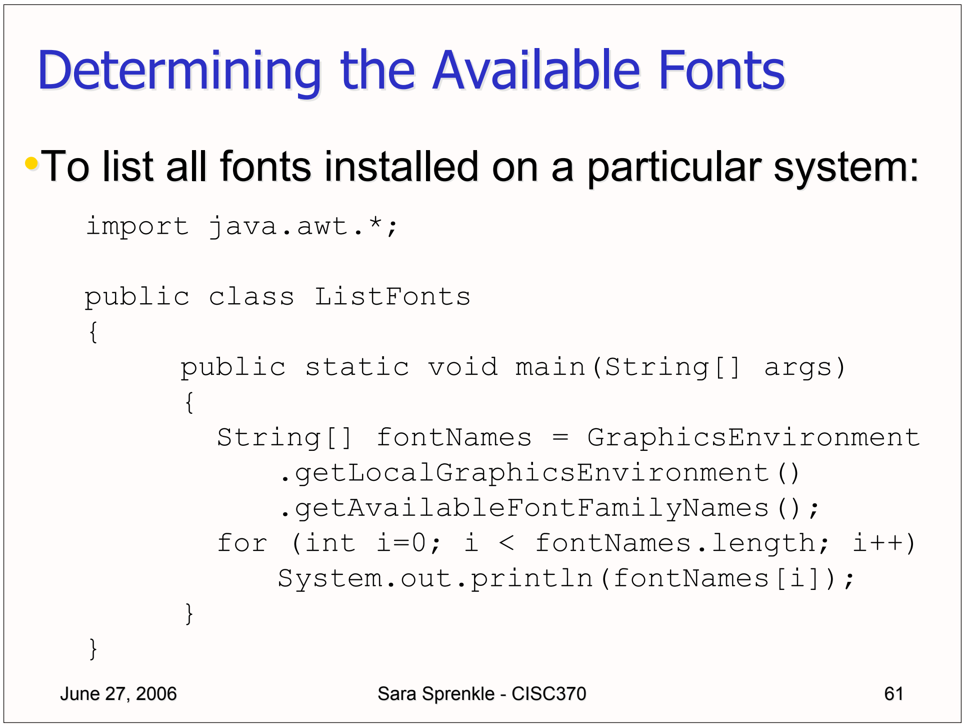
- Notice we call the superclass (JPanel) paintComponent() method
- The JPanel class has its own idea on how to draw/paint the panel
  - ➢Fills in the background color
- To make sure background color gets filled, call the superclass paintComponent() method
  - ➢Every JPanel should color its background

Changing the Text Font

- Previous code drew text using default system font
- We can also change the font
- We need to determine which fonts are installed on machine running the program

Determining the Available Fonts

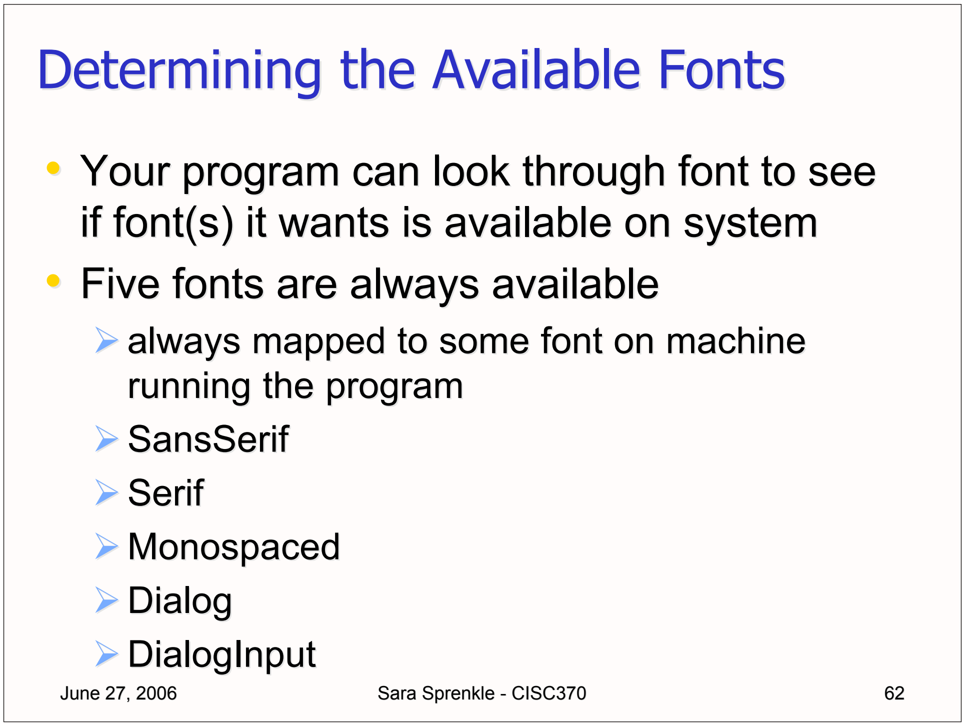
- **GraphicsEnvironment** class
  - Represents the graphical environment of the system
  - call getAvailableFontFamilyNames()
    - Returns an array of Strings
    - Each String contains the name of a font installed on the system



Determining the Available Fonts

- To list all fonts installed on a particular system:

```
import java.awt.*;

public class ListFonts
{
      public static void main(String[] args)
      {
        String[] fontNames = GraphicsEnvironment
            .getLocalGraphicsEnvironment()
            .getAvailableFontFamilyNames();
        for (int i=0; i < fontNames.length; i++)
            System.out.println(fontNames[i]);
      }
}
```



Determining the Available Fonts

- Your program can look through font to see if font(s) it wants is available on system
- Five fonts are always available
  - always mapped to some font on machine running the program
  - SansSerif
  - Serif
  - Monospaced
  - Dialog
  - DialogInput

Creating a Font Object

- After you know the type of font you want, make a **Font** object that represents the font on the system
  - ➢ constructor for a Font object takes three arguments:
    - a String with the font name
    - a constant (defined in the Font class) that describes the font style (plain, **bold**, *italic*, or ***bold italic***)
    - an integer for the point size

Creating a Font Object

```
Font sansbold14 = new Font(“SansSerif”, Font.BOLD, 14);
Font helvi12    = new Font(“Helvetica”, Font.ITALIC, 12);
```

- After a Font object has been created, you can change the font that the Graphics object uses by calling setFont()
- For example…

```
Font sansbold14 = new Font(“SansSerif”, Font.BOLD, 14);
g.setFont(sansbold14);
g.drawString(“Hello there in SansSerif.”, 75, 100);
```