Computer Science 2

Lecture 4

Programming Graphics and Event Handling


Very, VERY sorry

Extra material in Canvas

Overview

1. Console applications vs Graphical applications
2. Graphical applications
   - Graphical classes
   - Classes Color, BasicStroke and Font
   - Examples
3. Reading input
4. Event-handling model: events, sources, and listeners
5. Inner classes in listeners
6. Mouse adapter class
7. GUI components example

Learning goals

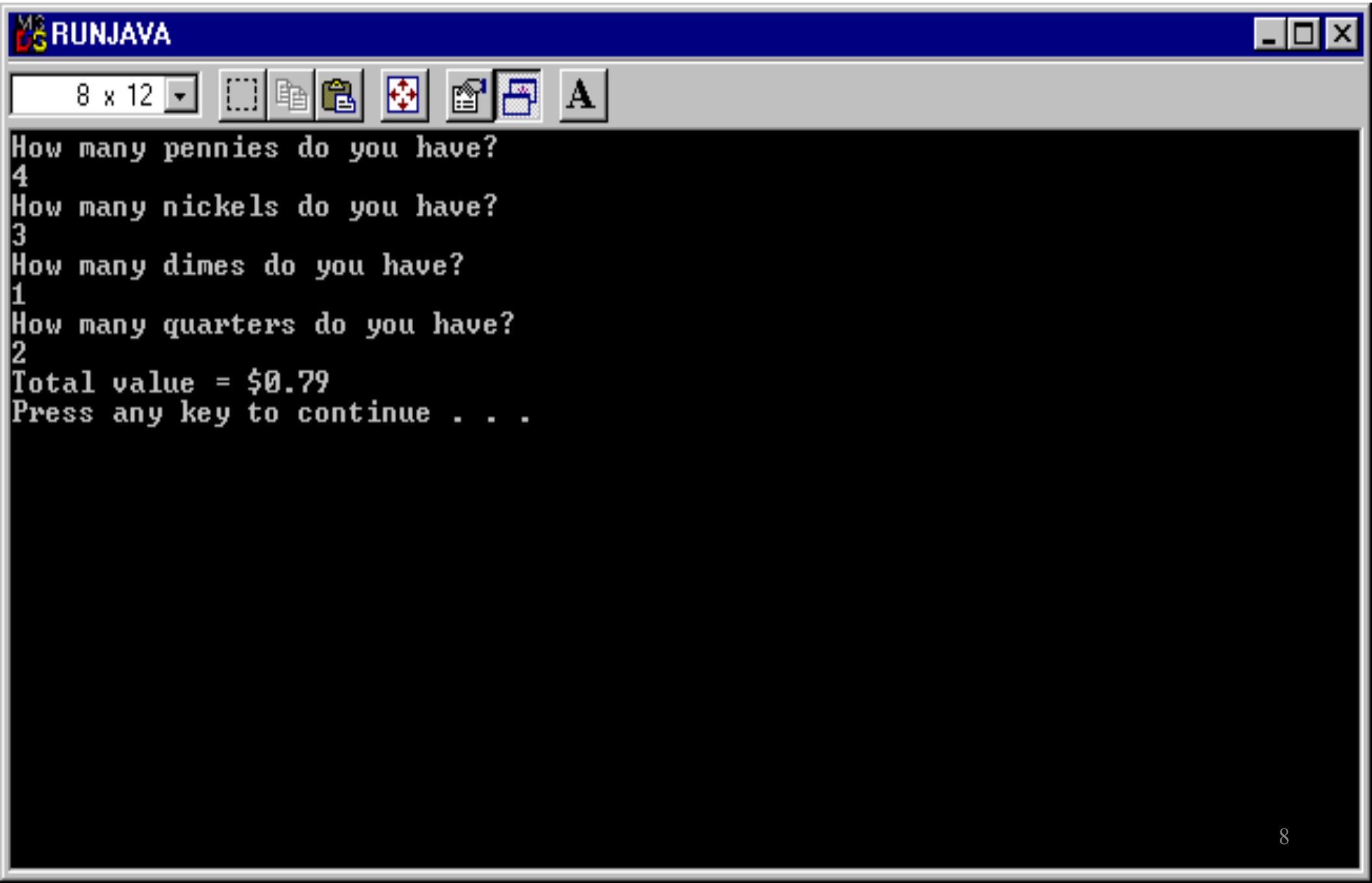
- Identify differences between console and *graphical application*
- Be able to configure and use *basic GUI components*
- Understand and be able to utilize commonly use *graphical classes*
- Understand and be able to implement the *event-handling model*
- Understand and be able to implement and use *inner classes*
- Implement *listener-based applications*
- Identify differences between *extends* and *implements*
- Identify the benefits of using *interfaces/adapters*
- Identify differences between *interfaces* and *adapters*

CONSOLE APPLICATIONS VS GRAPHICAL APPLICATIONS

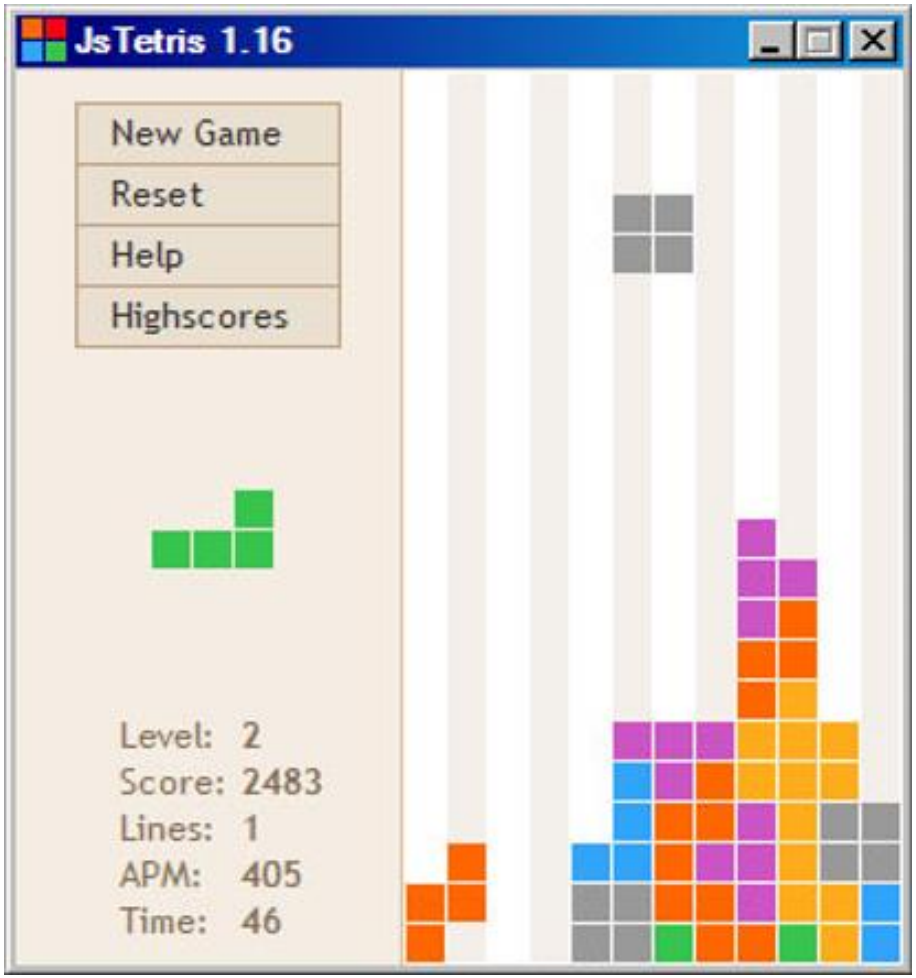
Applications

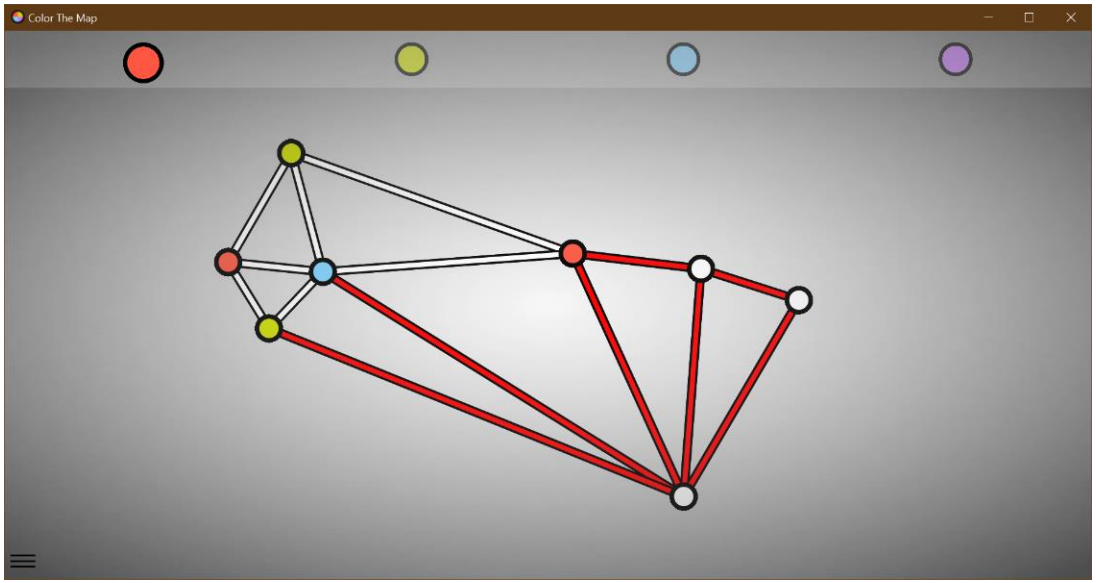
- **Applications are stand-alone programs that can be launched from the command line.**
  - Each application has a main method;
  - Each application has no restrictions on what it can do;
  - There exist two types of applications:
    - Console applications: run in a single terminal window;
    - Graphical Applications: use one or more windows filled with graphical user interface (GUI) components such as buttons, text fields, and menus.

A console application



A graphical application





GRAPHICAL APPLICATIONS

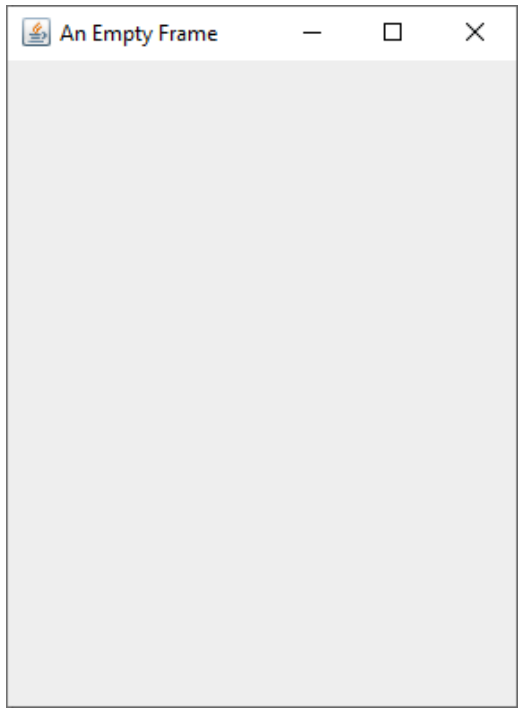
Graphical applications

- A graphical application shows information in a frame window.

There exist three main graphics packages:

- **Abstract Windowing Toolkit (AWT)**
- **Swing**
- JavaFX (Lecture 5)

Graphical applications



Showing a frame in `swing`:

1. Construct an object of the JFrame class:
```
JFrame frame = new JFrame();
```

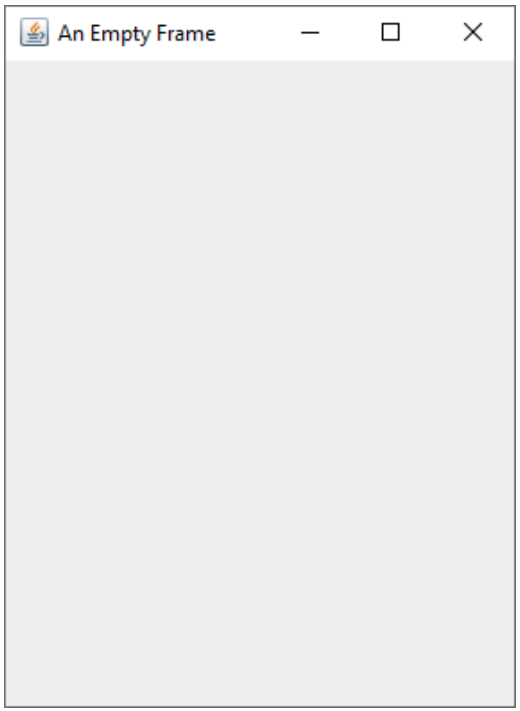
2. Set the size of the frame:
```
frame.setSize(300, 400);
```

3. Set the title of the frame:
```
frame.setTitle("An Empty Frame");
```

4. Set the default operation:
```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

5. Make the frame visible:
```
frame.setVisible(true);
```

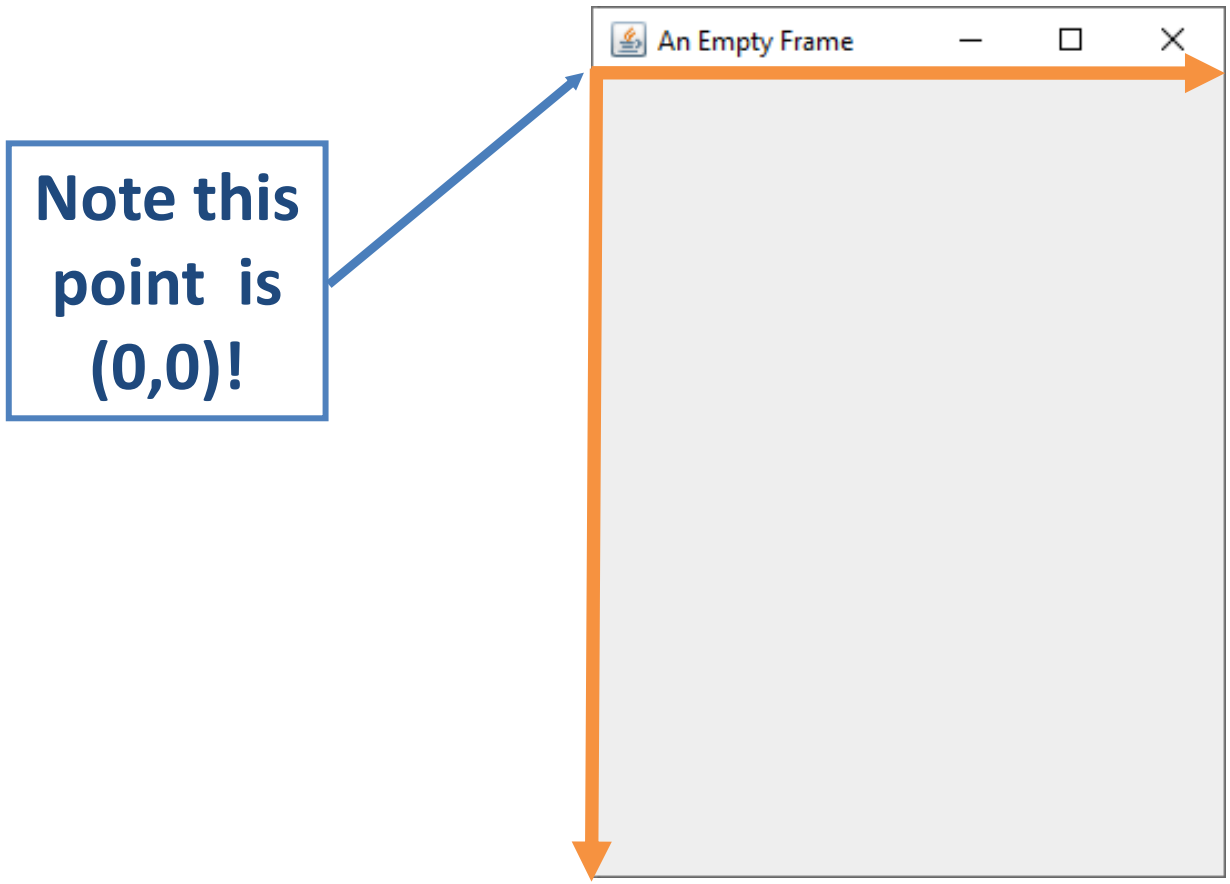
Graphical applications




```
import javax.swing.*;

public class EmptyFrameViewer
{
	public static void main(String[] args)
	{
		JFrame frame = new JFrame();
		final int FRAME_WIDTH = 300;
		final int FRAME_HEIGHT = 400;
		frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
		frame.setTitle("An Empty Frame");
		frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		frame.setVisible(true);
	}
}
```

Note that to use the `JFrame` class you need to import `swing` package!

Graphical applications



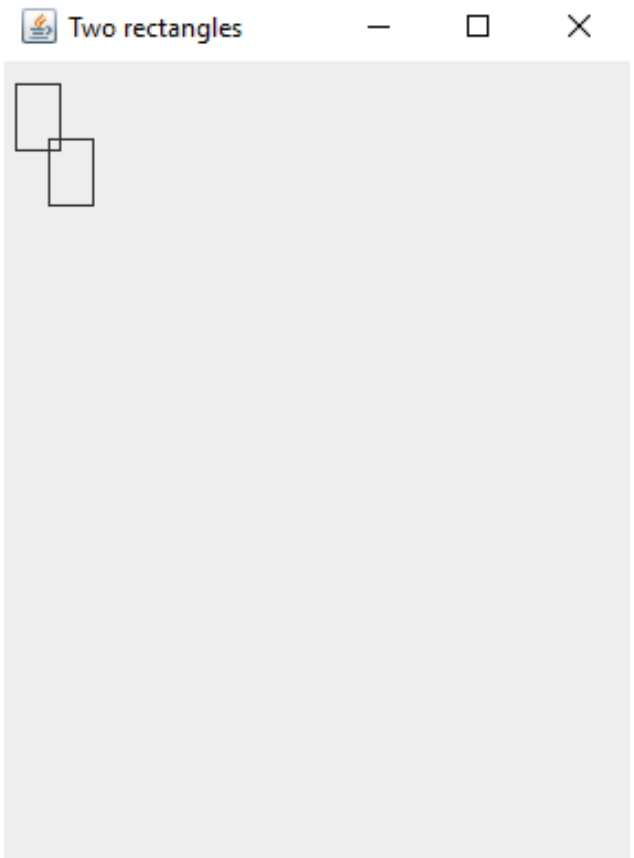
Graphical applications

- JFrame is a “container” UI element
  - Other elements “live in” JFrames
- By “live in” is meant
  - The coordinates of all elements in the JFrame are relative to the upper-left-hand corner of the JFrame
  - When the JFrame is moved, all the elements in the JFrame move with it
- In order to live in a container an object must extend the java.awt.Component abstract class
- Let us look at creating a Frame with a couple of rectangles in it

Graphical applications

- First we will look at creating the Component that will contain our Rectangle objects
- Then we will create a JFrame to hold it and display everything
- In the end it will look like the following

Graphical applications: drawing shapes



Graphical applications: drawing shapes

- To display a drawing, define a class (e.g. `RectangleComponent`) that extends the `JComponent` class
- The class `JComponent` is a standard Java class that represents a blank component
- To show anything in a frame, we override the method `paintComponent` in the `RectangleComponent` class, and then we construct a component object and add it to the frame.

```
public class RectangleComponent extends JComponent
{
      public void paintComponent(Graphics g)
      {
            // Recover Graphics2D
            Graphics2D g2 = (Graphics2D) g;
            // Put here your drawing operators
      }
}
```

`paintComponent:` called whenever the component needs to be repainted!

[Interlude: *graphics* and *graphics2D*]

```
public class RectangleComponent extends JComponent
{
      public void paintComponent(Graphics g)
      {
            // Recover Graphics2D
            Graphics2D g2 = (Graphics2D) g;
            // Put here your drawing operators
      }
}
```

- The drawing components inherit from the JComponent class
- Graphics is an abstract class (we cannot create an instance)
- Graphics was not intended as an object-oriented component
  - The 2D version was created to address the new programming paradigm
  - Graphics is still the input parameter of *paintComponent* to allow compatibility with previous versions)

Graphical applications: drawing shapes

```
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Rectangle;
import javax.swing.JComponent;

/**
A component that draws two rectangles.
*/
public class RectangleComponent extends JComponent
{
   public void paintComponent(Graphics g)
   {
      // Recover Graphics2D
      Graphics2D g2 = (Graphics2D) g;

      // Construct a rectangle and draw it
      Rectangle box = new Rectangle(5, 10, 20, 30);
      g2.draw(box);

      //Move rectangle 15 units to the right and 25 units down
      box.translate(15, 25);
      g2.draw(box);
   }
}
```

Abstract Windowing Toolkit

Graphical applications: drawing shapes

```
import javax.swing.JFrame;

public class RectangleViewer
{
   public static void main(String[] args)
   {
      JFrame frame = new JFrame();

      final int FRAME_WIDTH = 300;
      final int FRAME_HEIGHT = 400;

      frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
      frame.setTitle("Two rectangles");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      RectangleComponent component = new RectangleComponent();
      frame.add(component);
      frame.setVisible(true);
   }
}
```

Summary: Steps to draw in a Frame

1. Write your component class `MyComponent` inheriting from the class `JComponent`. Specify your graphics by providing code in the method `paintComponent(Graphics g)`.

2. Construct a frame:

```
JFrame frame = new JFrame();
```

3. Construct an object of your component class:

```
MyComponent component = new MyComponent();
```

4. Add the `component` to the frame:

```
frame.add(component);
```

5. Make the frame visible:

```
frame.setVisible(true);
```

Graphical applications

GRAPHICAL CLASSES

Graphical Classes

- There are many classes of graphical objects in the java.awt.geom package
  - Ellipses, lines, points, etc.
  - Any class that implements the java.awt.Shape interface will work
- We will not cover them all here
  - You can find them all at https://docs.oracle.com/javase/8/docs/api/index.html?java/awt/Shape.html
- It is worth noting that the graphical objects use `double` rather than `int` for measurement
  - For example, pixel location or width

Graphical Classes

- This is to allow the programmer to do general-purpose calculations on geometric objects
- The point of geometric objects is to model the world
- Objects in the world do not always come in nice integer sizes
- Java will handle translating what an x-coordinate of 12.35 means in terms of pixels

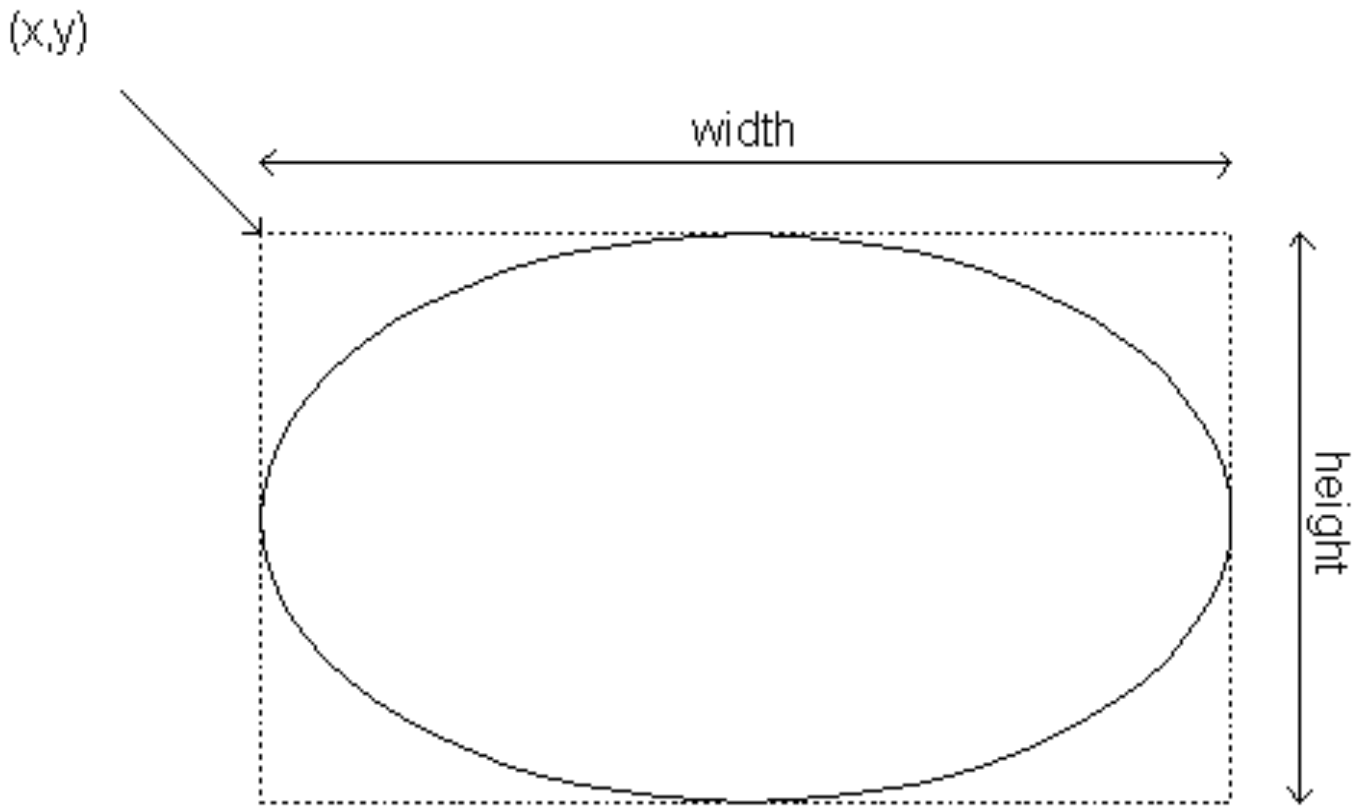
Graphical classes: `Double` version

- Class `Point2D.Double` with constructor:
  - `Point2D.Double(double x, double y)`
- Class `Line2D.Double` with constructors:
  - `Line2D.Double(double X1, double Y1, double X2, double Y2)`
  - `Line2D.Double(Point2D p1, Point2D p2)`

Notice that we use Double for the pixels instead of integers

Graphical classes: `Double` version

- Class **`Ellipse2D.Double`** with constructor:
  - **`Ellipse2D.Double(double x, double y, double width, double height)`**



Graphical classes

- More about two-dimensional geometric objects:
  - https://docs.oracle.com/javase/8/docs/api/index.html?java/awt/geom/package-summary.html
- The official Javadocs page is the single most important URL you will see in your Java education

Graphical Applications

CLASSES COLOR, BASICSTROKE AND FONT

Class `Color`

- By default, all shapes are drawn in black;
- To change the color, apply the method **`setColor`** on **`g2`** parameter with the actual parameter an object of the class **`Color`**;
- The constructor of the class **Color** is:
  - **`Color(float red, float green, float blue)`** where **`red`** is the intensity of the red color, **`green`** is the intensity of the green color, and **`blue`** is the intensity of the blue color. **`red`**, **`green`**, and **`blue`** are **`float`** in the range **[0.0, 1.0]**.
- There are predefined colors: **`Color.black`**, **`Color.GREEN`**, etc.
- Example: **`g2.setColor(Color.red);`**

Class `BasicStroke`

- To draw thicker lines, supply a different stroke object to `Graphics2D g2` parameter:
  - `g2.setStroke(new BasicStroke(4.0F));`

Drawing Strings

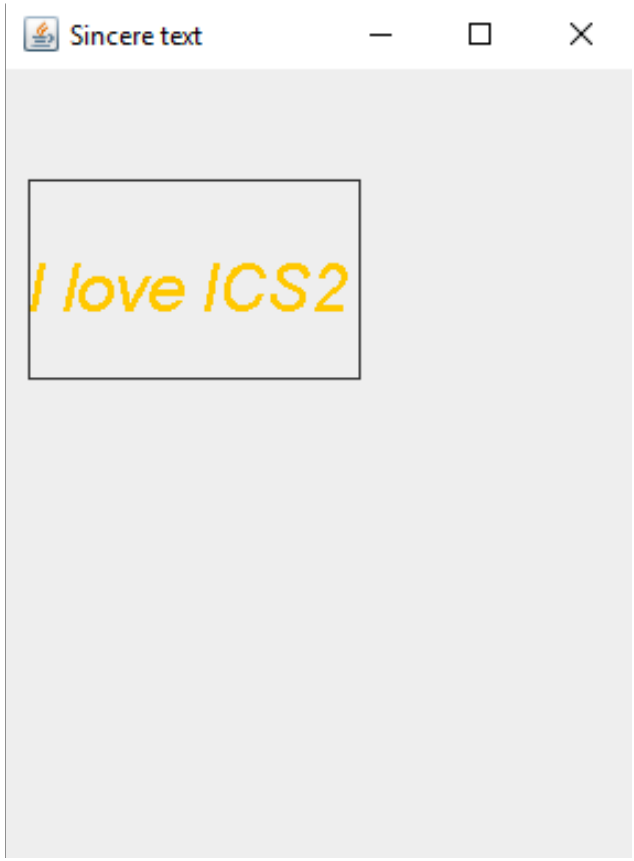
- To draw a string use method **`drawString`** of the class **`Graphics2D`**:

```
g2.drawString(String s, float x, float y);
```

  where **`s`** is the string to be drawn, and **`x`** and **`y`** are the **`x`** and **`y`** coordinates of the base point.

- Example: `g2.drawString("Applet", 50, 100);`

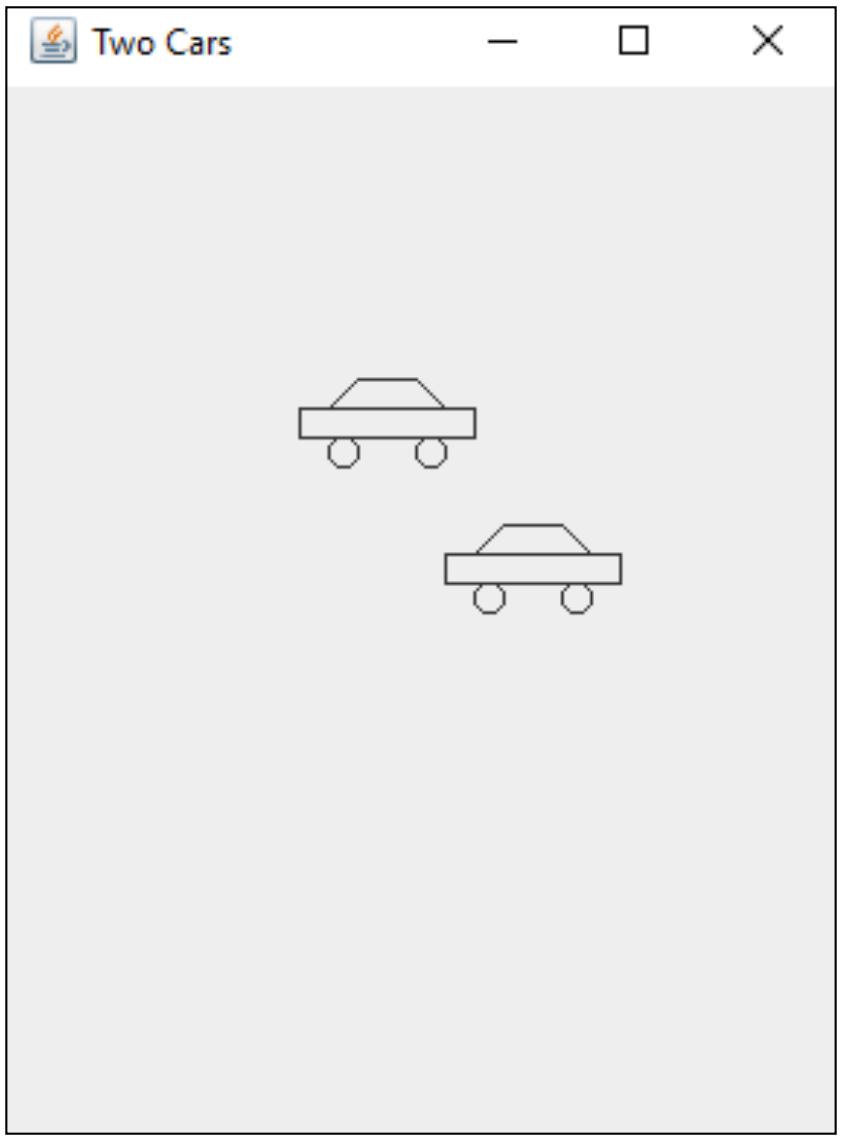
```java
import java.awt.Graphics;
import java.awt.Graphics2D;
import javax.swing.JComponent;
import java.awt.Rectangle;
import javax.swing.JFrame;
import java.awt.Font;
import java.awt.Color;

public class SincereText extends JComponent
{
	public void paintComponent(Graphics g)
	{
		Graphics2D g2 = (Graphics2D) g;

		final int posSquareX = 10;
		final int posSquareY = 50;

		// Construct a square and draw it
		Rectangle box = new Rectangle(posSquareX, posSquareY, 150, 90);
		g2.draw(box);

		// Construct a String and draw it
		Font myFont = new Font("Arial", Font.ITALIC, 30);
		g2.setFont(myFont);
		g2.setColor(Color.ORANGE);
		g2.drawString("I love ICS2", posSquareX, posSquareY+60);

	}

	public static void main(String[] args)
	{
		JFrame frame = new JFrame();

		final int FRAME_WIDTH = 300;
		final int FRAME_HEIGHT = 400;

		frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
		frame.setTitle("Sincere text");
		frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

		SincereText component = new SincereText();

		frame.add(component);
		frame.setVisible(true);
	}
}
```



Graphical Applications

CAR EXAMPLE

Frame with two cars



Important

- To let Visual Studio Code know where to find the required files (we will be using three classes for this example):
  - Create a *package.* We need to add the following statement at the beginning of each class involved (being the package name the same as the folder containing the files)

```
package Car;

public class CarComponent extends Jcomponent { … }
```

  - **Open the folder directly from Visual Studio Code**

CarViewer Class

```
import javax.swing.JFrame;

public class CarViewer {
public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setSize(300, 400);
        frame.setTitle("Two Cars");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        CarComponent component = new CarComponent();
        frame.add(component);
        frame.setVisible(true);
}
}
```

CarComponent Class

```
import javax.swing.JComponent;
import java.awt.Graphics;
import java.awt.Graphics2D;

public class CarComponent extends JComponent {
public void paintComponent(Graphics g) {
        Graphics2D g2 = (Graphics2D) g;
        Car car1 = new Car(100, 100);
        Car car2 = new Car(150, 150);
        car1.drawMe(g2);
        car2.drawMe(g2);
}
}
```

Car Class

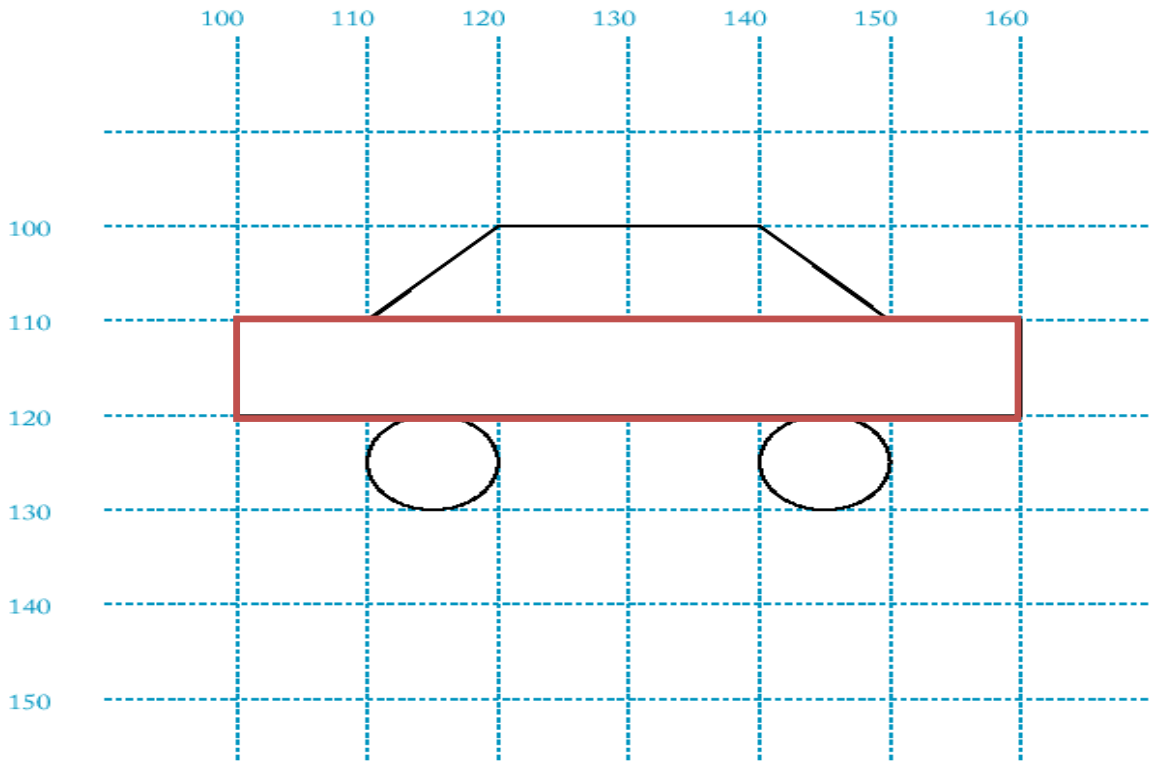
```
public class Car {
public Car(double x, double y) {
        xLeft = x;
        yTop = y;
}
public void drawMe(Graphics2D g2) {
        Rectangle2D.Double body = new Rectangle2D.Double(xLeft, yTop + 10, 60, 10);
        Ellipse2D.Double frontTire = new Ellipse2D.Double(xLeft + 10, yTop + 20, 10, 10);
        Ellipse2D.Double rearTire = new Ellipse2D.Double(xLeft + 40, yTop + 20, 10, 10);
        Point2D.Double r1 = new Point2D.Double(xLeft + 10, yTop + 10);
        Point2D.Double r2 = new Point2D.Double(xLeft+20, yTop);
        Point2D.Double r3 = new Point2D.Double(xLeft + 40, yTop);
        Point2D.Double r4 = new Point2D.Double(xLeft+ 50, yTop + 10);
        Line2D.Double frontWindshield = new Line2D.Double(r1, r2);
        Line2D.Double roofTop = new Line2D.Double(r2, r3);
        Line2D.Double rearWindshield = new Line2D.Double(r3, r4);
        g2.draw(body);
        g2.draw(frontTire);
        g2.draw(rearTire);
        g2.draw(frontWindshield);
        g2.draw(roofTop);
        g2.draw(rearWindshield);
}
private double xLeft;
private double yTop; }
```

`Car` Class

- Note that the drawMe method does not draw the parts at absolute locations
  - It uses offsets from the base x and y coordinates
  - This is a small example of how working with graphics requires a large amount of algebra
- The drawMe method is large
- We will take a look at what the parts do step-by-step in the next few slides

Car Class

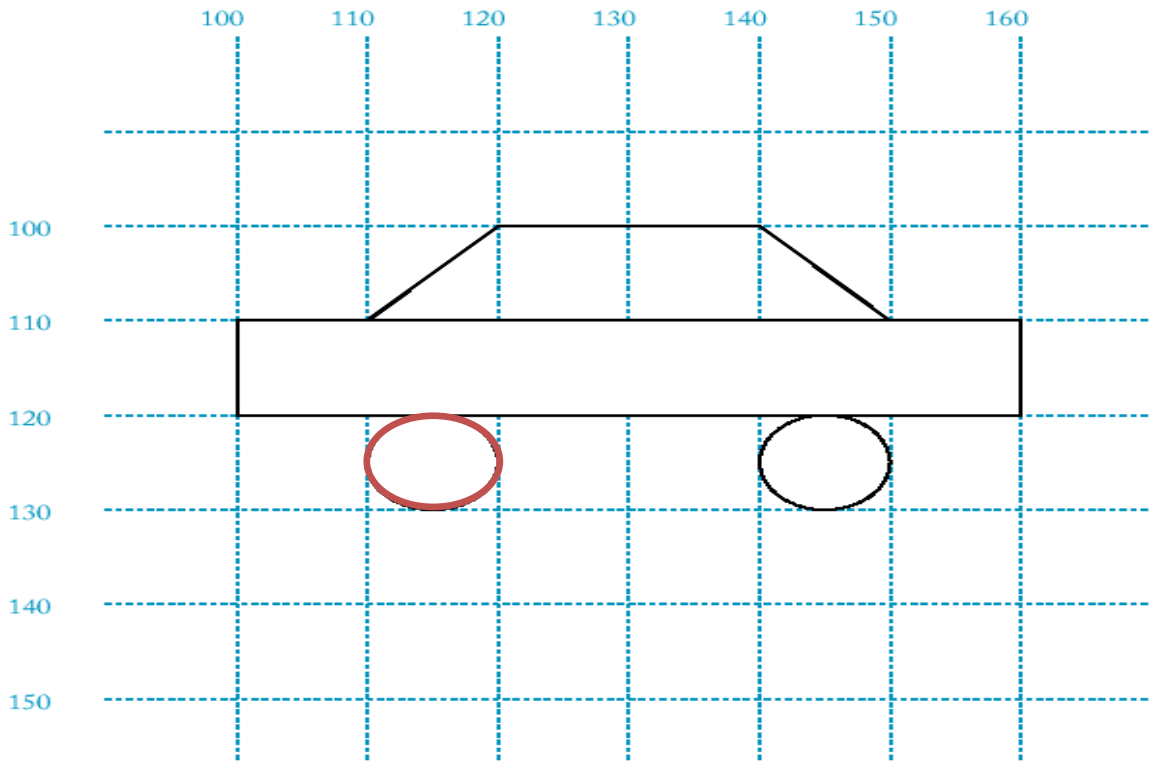
```
xLeft = 100;
yTop = 100;
```




```
public void drawMe(Graphics2D g2) {
    Rectangle2D.Double body = new Rectangle2D.Double(xLeft, yTop + 10, 60, 10);
    …
}
```

Car Class

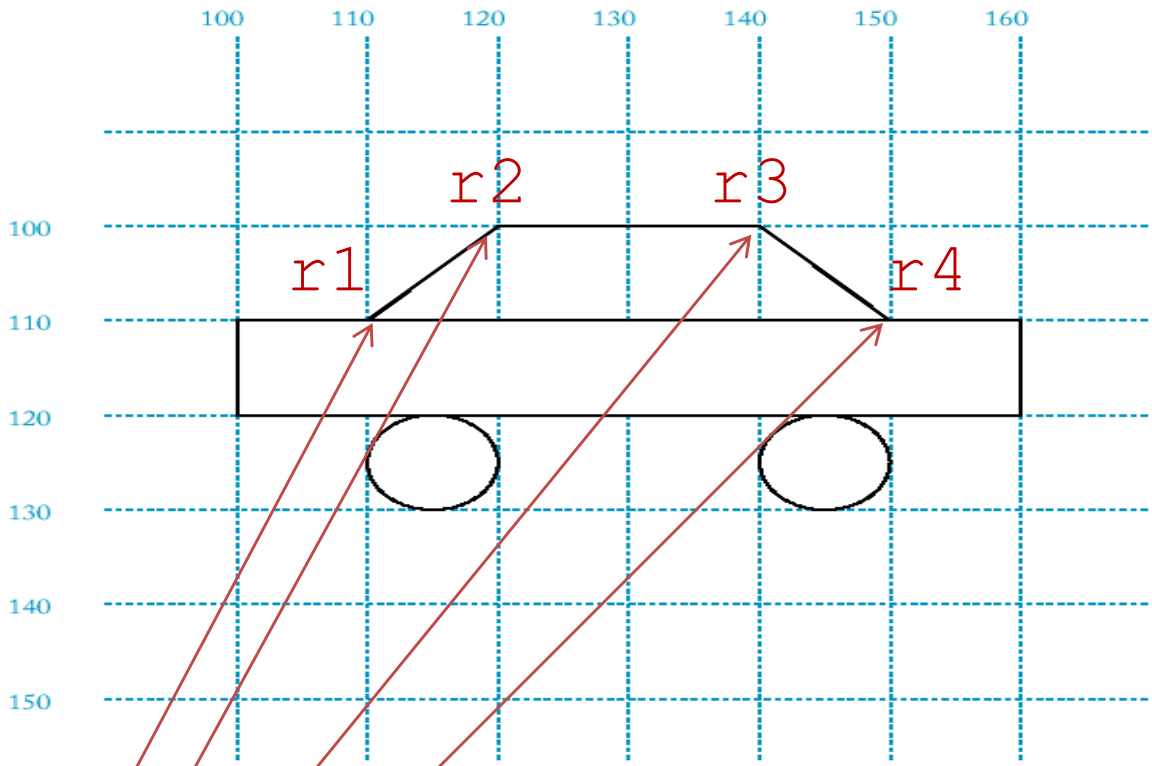
```
xLeft = 100;
yTop = 100;
```




```
public void drawMe(Graphics2D g2) { …
    Ellipse2D.Double frontTire = new Ellipse2D.Double(xLeft + 10, yTop + 20, 10, 10);
    …
}
```

Car Class

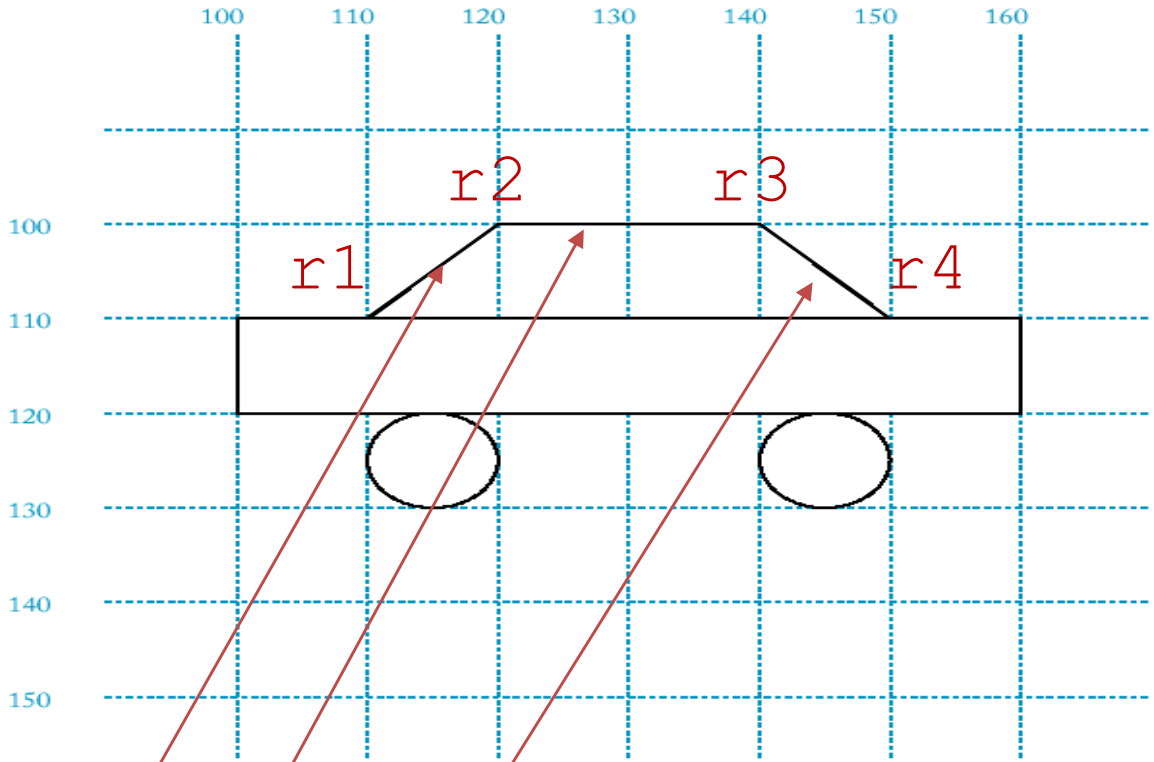
```
xLeft = 100;
yTop = 100;
```




```
public void drawMe(Graphics2D g2) { …
    Point2D.Double r1 = new Point2D.Double(xLeft + 10, yTop + 10);
    Point2D.Double r2 = new Point2D.Double(xLeft+20, yTop);
    Point2D.Double r3 = new Point2D.Double(xLeft + 40, yTop);
    Point2D.Double r4 = new Point2D.Double(xLeft+ 50, yTop + 10); … }
```

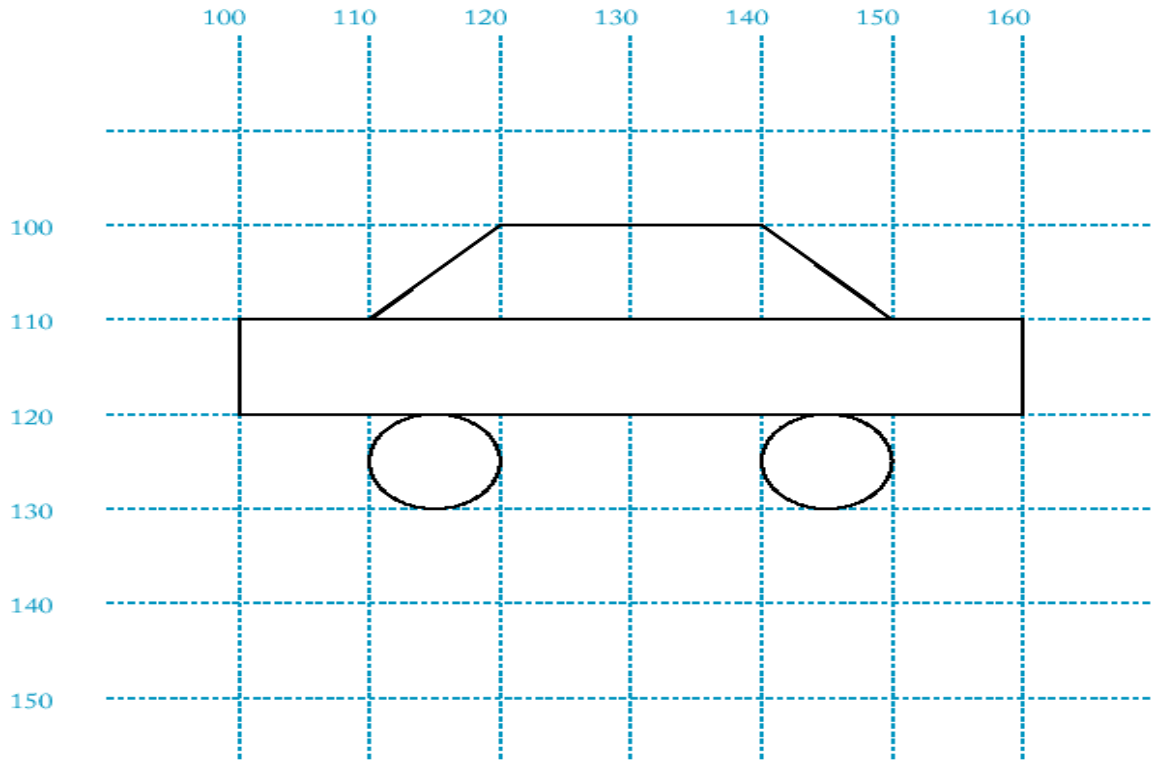
Car Class

```
xLeft = 100,
yTop = 100;
```




```
public void drawMe(Graphics2D g2) { …
    Line2D.Double frontWindshield = new Line2D.Double(r1, r2);
    Line2D.Double roofTop = new Line2D.Double(r2, r3);
    Line2D.Double rearWindshield = new Line2D.Double(r3, r4);
… }
```

Car Class




```
xLeft = 100;
yTop = 100;
```

```
public void drawMe(Graphics2D g2) { …
    g2.draw(body);
    g2.draw(frontTire);
    g2.draw(rearTire);
    g2.draw(frontWindshield);
    g2.draw(roofTop);
    g2.draw(rearWindshield);                    … }
```

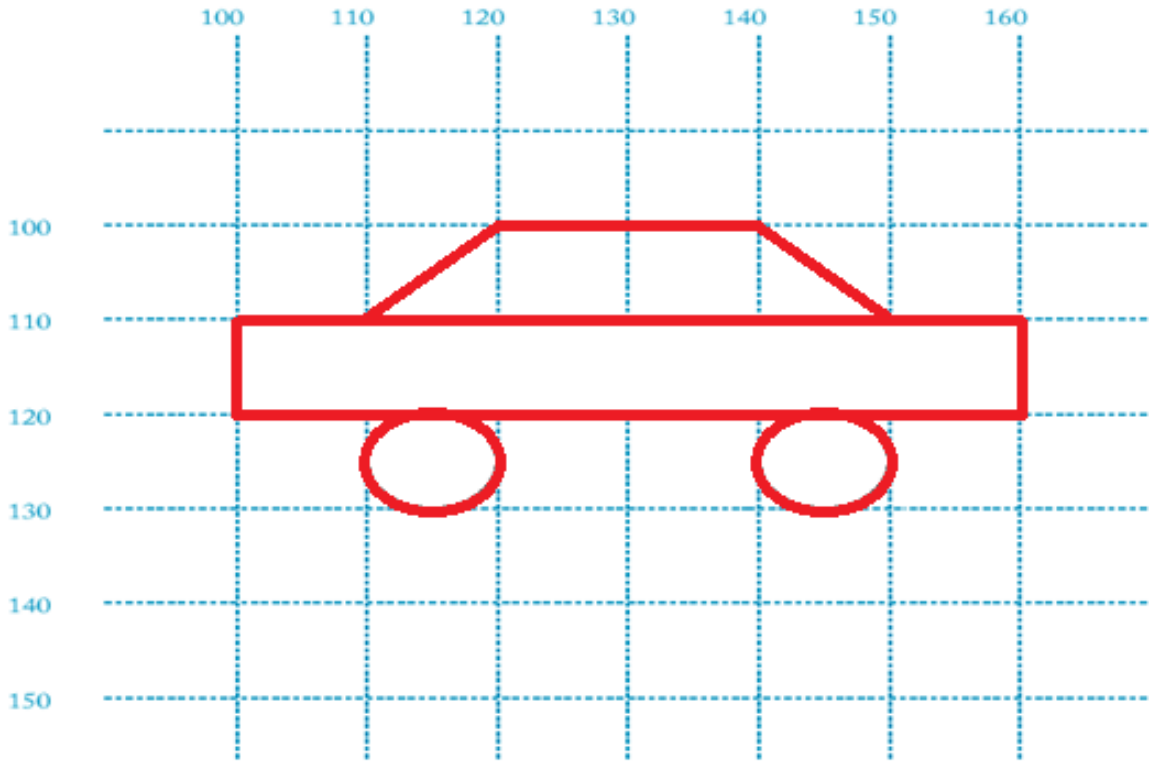
CarComponent Class

```
import javax.swing.JComponent;
import java.awt.Graphics;
import java.awt.Graphics2D;

public class CarComponent extends JComponent {
public void paintComponent(Graphics g) {
        Graphics2D g2 = (Graphics2D) g;
        Car car1 = new Car(100, 100);
        Car car2 = new Car(150, 150);
        car1.drawMe(g2);
        car2.drawMe(g2);
}
}
```

Exercise to practise: implement a method "translate" (as in the example of the two rectangles) to draw two cars instead of create them

“Tuning” your car

```
xLeft = 100;
yTop = 100;
```



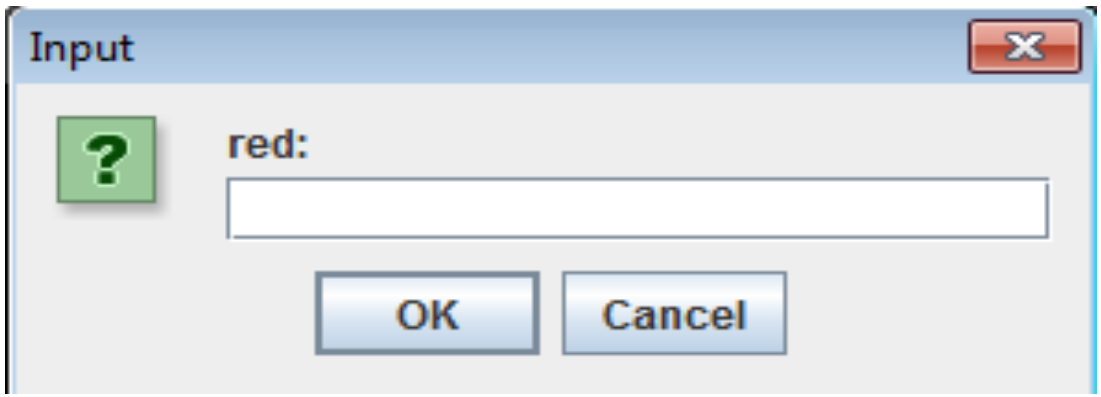

```
public void drawMe(Graphics2D g2) { …
    g2.setStroke(new BasicStroke(2));
    g2.setColor(Color.RED);
    g2.draw(body);
    g2.draw(frontTire);
    …
}
```

https://docs.oracle.com/javase/7/docs/api/java/awt/Graphics2D.html

READING INPUT

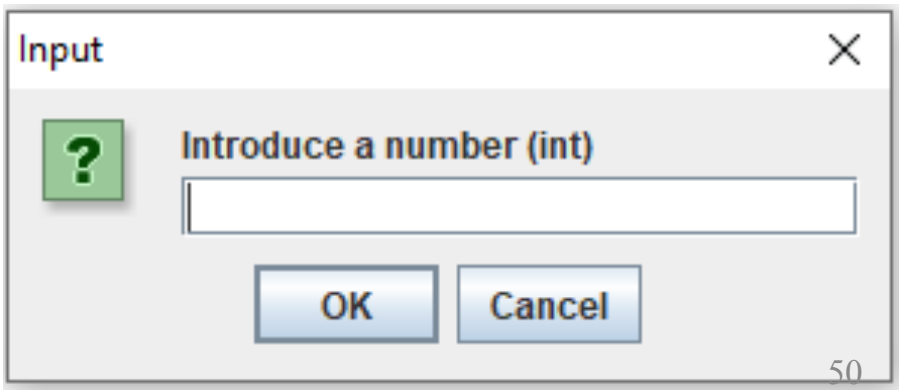
Reading input in windows

- `String input = JOptionPane.showInputDialog (prompt)`
- Convert strings to numbers if necessary:
  `int count = Integer.parseInt(input);`
- Conversion throws an exception if user doesn't supply a number
- Add `import java.swing.JOptionPane;`



Reading input in windows

```
import javax.swing.JOptionPane;

class ReadingInput {
    public static void main(String[] args) {
        String input = JOptionPane.showInputDialog("Introduce a number (int)");
        int count = Integer.parseInt(input);
        […]
        System.out.print("The number introduced is " + count);
    }
}
```



Reading input in windows

- This is a pretty simple way to interact with the computer
- The user will

1. See the window pop up
2. Type in an integer
3. Hit return

- The computer will

1. Display the window
2. Wait for the user to type something in and hit return
3. Store that value in a variable

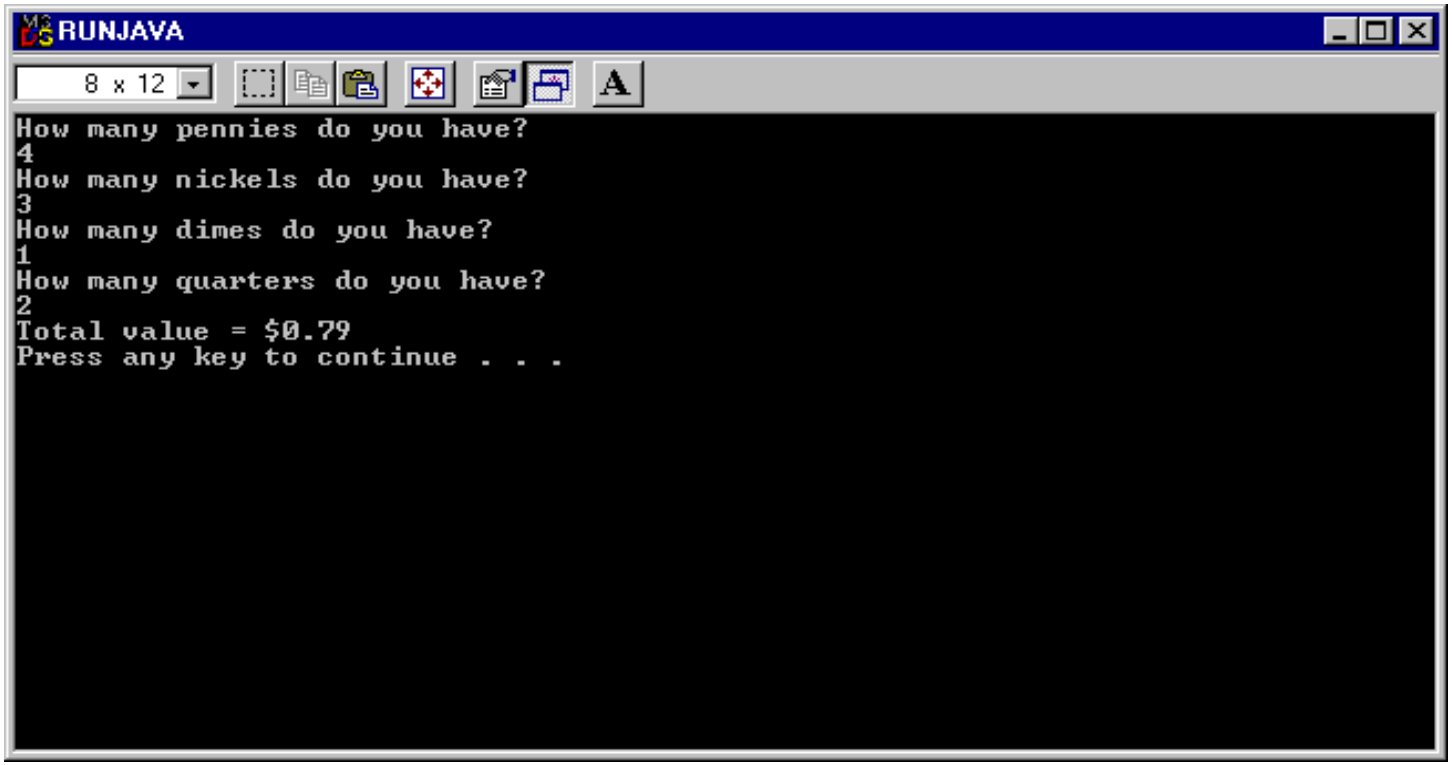
Reading input in windows

- Most current GUIs have many buttons, text fields, and so on, any of which the user could interact with at any time
- In order to do something like this first we have to look at Java's event-handling model

EVENT-HANDLING MODEL

Program control

- In console-based applications user input is under the control of the program: i.e., the program will ask the user for input in a specific order.

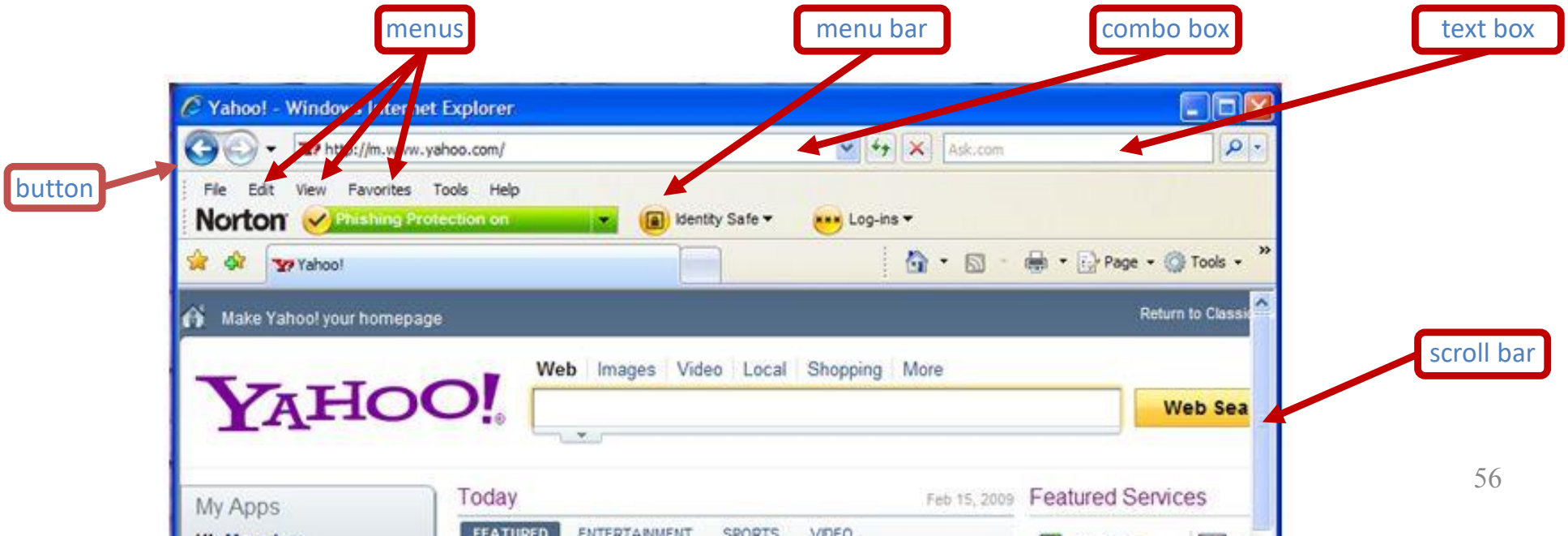


Program control

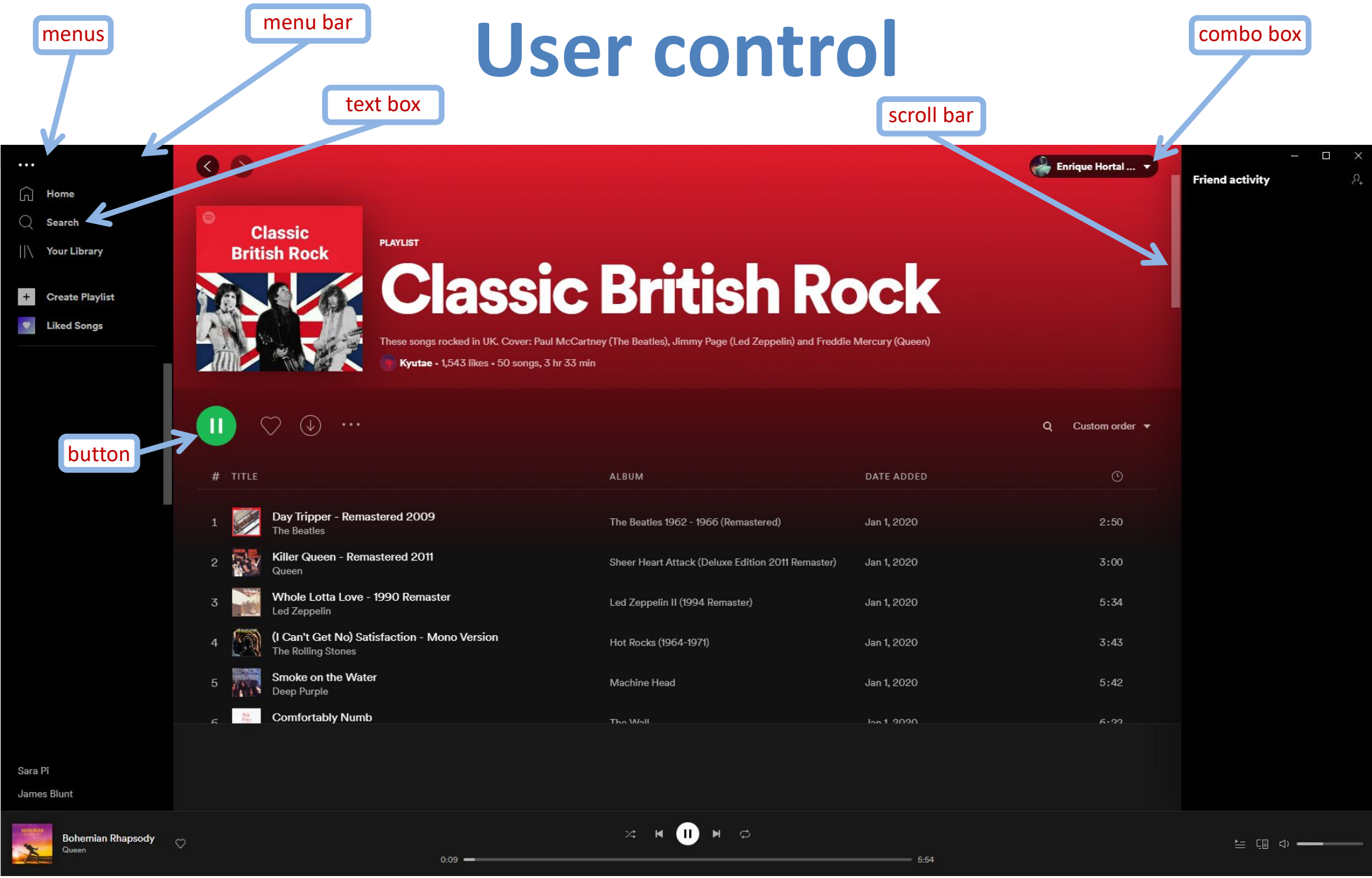
- In a console application, when input is requested from the user, the entire program stops and waits for the user to type something
  - Known as synchronous input
- The user can type whatever they want, but there is no way for the user to choose what to do
  - For example, the user cannot skip a prompt and go on the the next one, or go back to a previous one

User control

- In programs with a modern graphical user interface (GUI) the user is in control:
  - The user can use both the mouse and keyboard
  - The user can manipulate many parts of the GUI in any desired order (click buttons, pull down menus, scroll bars *etc.*).

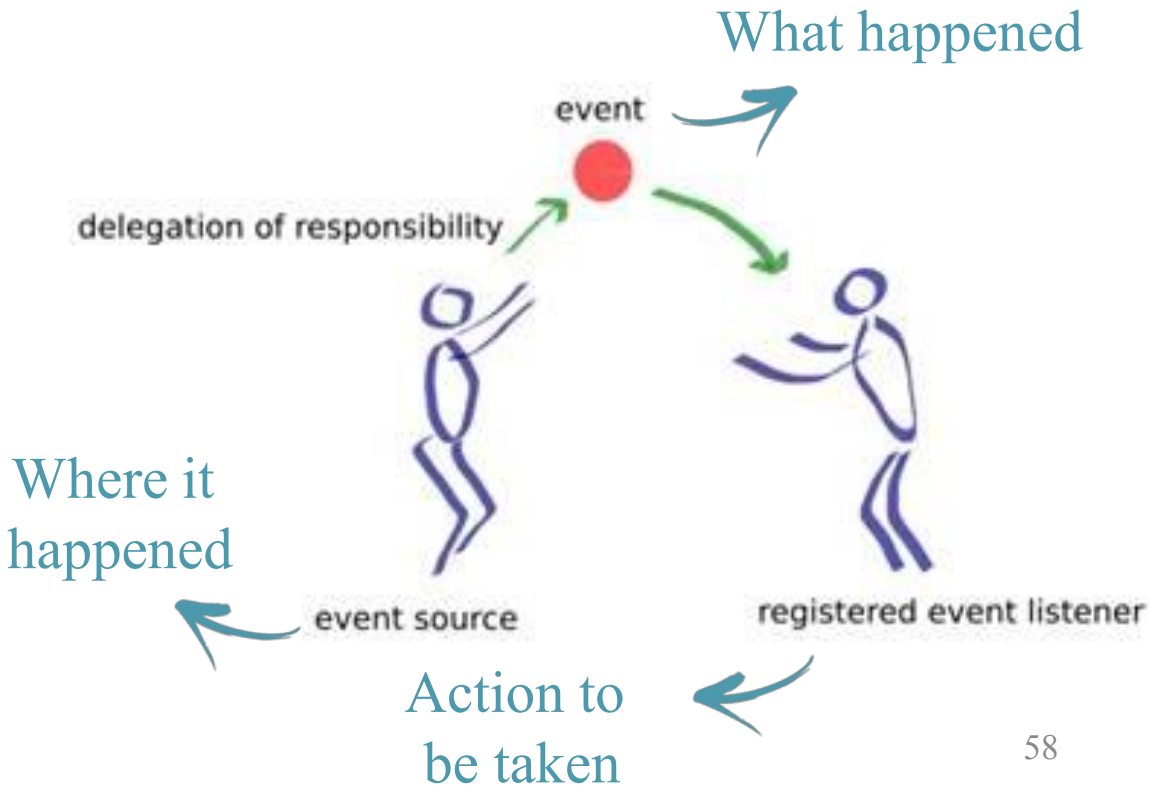


User control



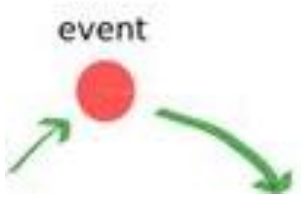
Event-handling model

- The Graphical User Interface (GUI) in Java is based on the event-handling model;
- The event-handling model is based on:
  - Events;
  - Event sources;
  - Event listeners.



Events

- The Java window manager sends a program an event notification when:
  - the user types characters;
  - the user uses the mouse inside one of the program's windows;
- The window manager generates a huge number of events:



  - *e.g.* whenever the mouse moves a tiny interval over a window a *mouse move* event is generated;
  - Most programs have no interest in many of these events.

Event sources

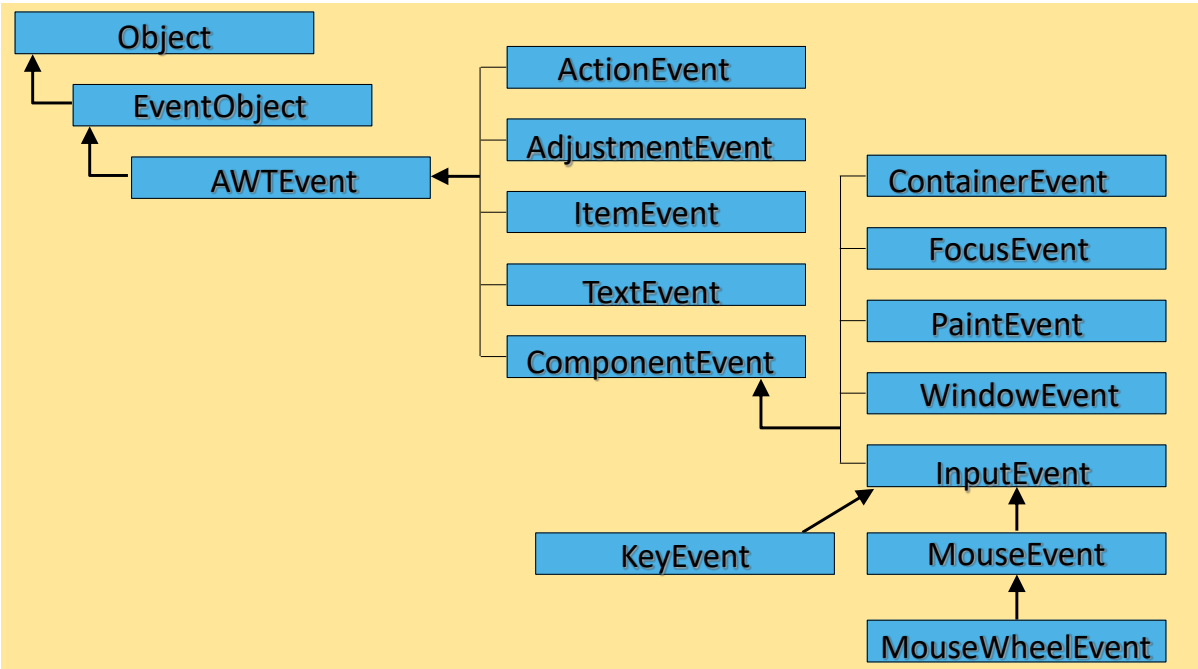
- The event source is the GUI component that generates a particular event:
  - button -> click events
  - menu item -> menu selection event
  - scrollbar -> scrollbar adjustment event
- Once you have determined the event source, you attach one or more listeners to it.

Event listeners

- To avoid unnecessary events, a program indicates events it likes to receive;
- This is done by installing event listeners;
- Different listeners are used to listen for and respond to different kinds of events.

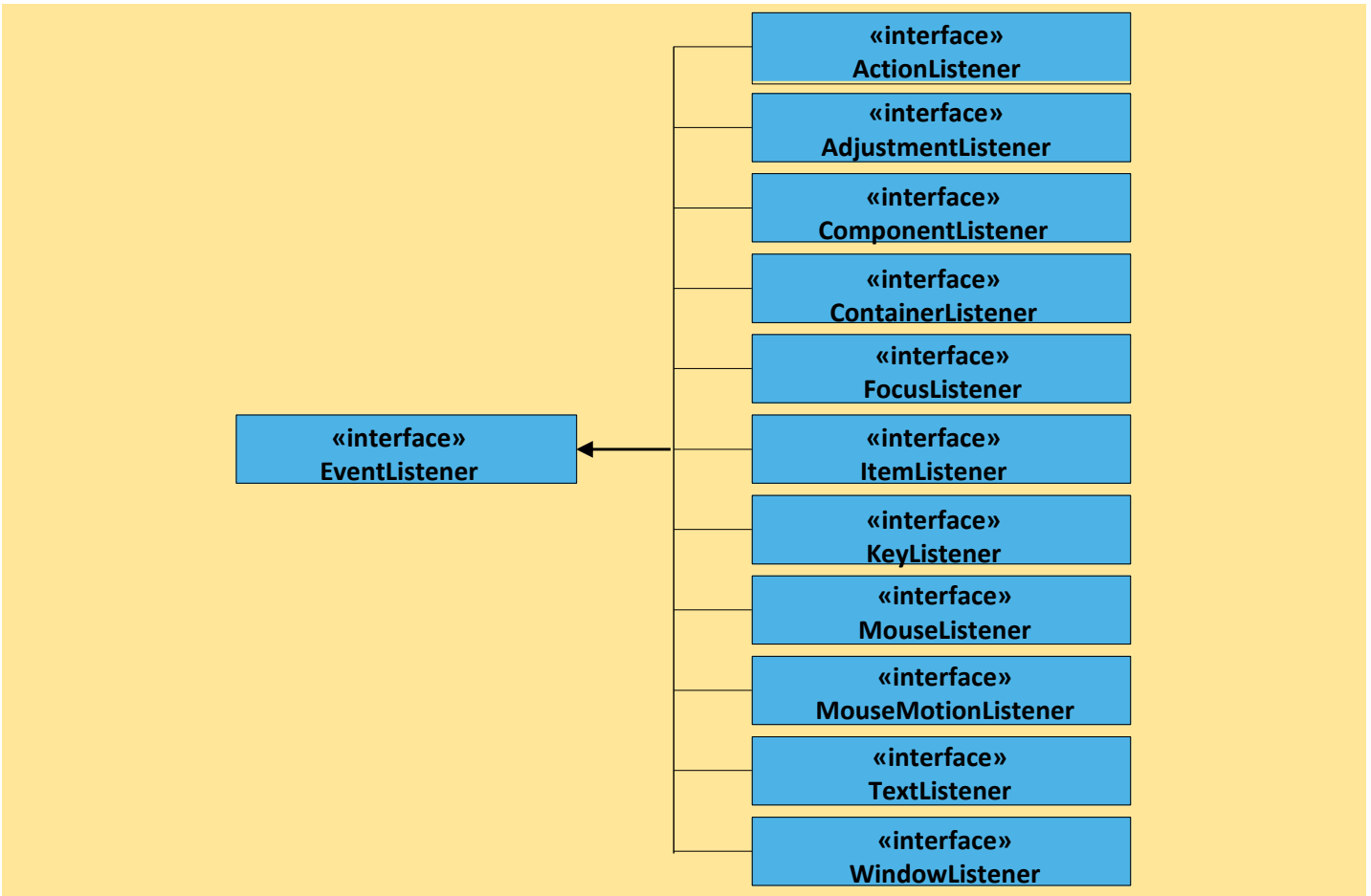
Events

- Events are represented as standard classes in Java;
- Each event object has type, name, source, and additional relevant information;
- Event objects are provided by Java window manager;



Event listeners

- Event listeners are implemented in Java as classes;
- An event-listener class indicates the type **T** of events it listens by implementing a listener interface **T**;



Event sources & event-handling model

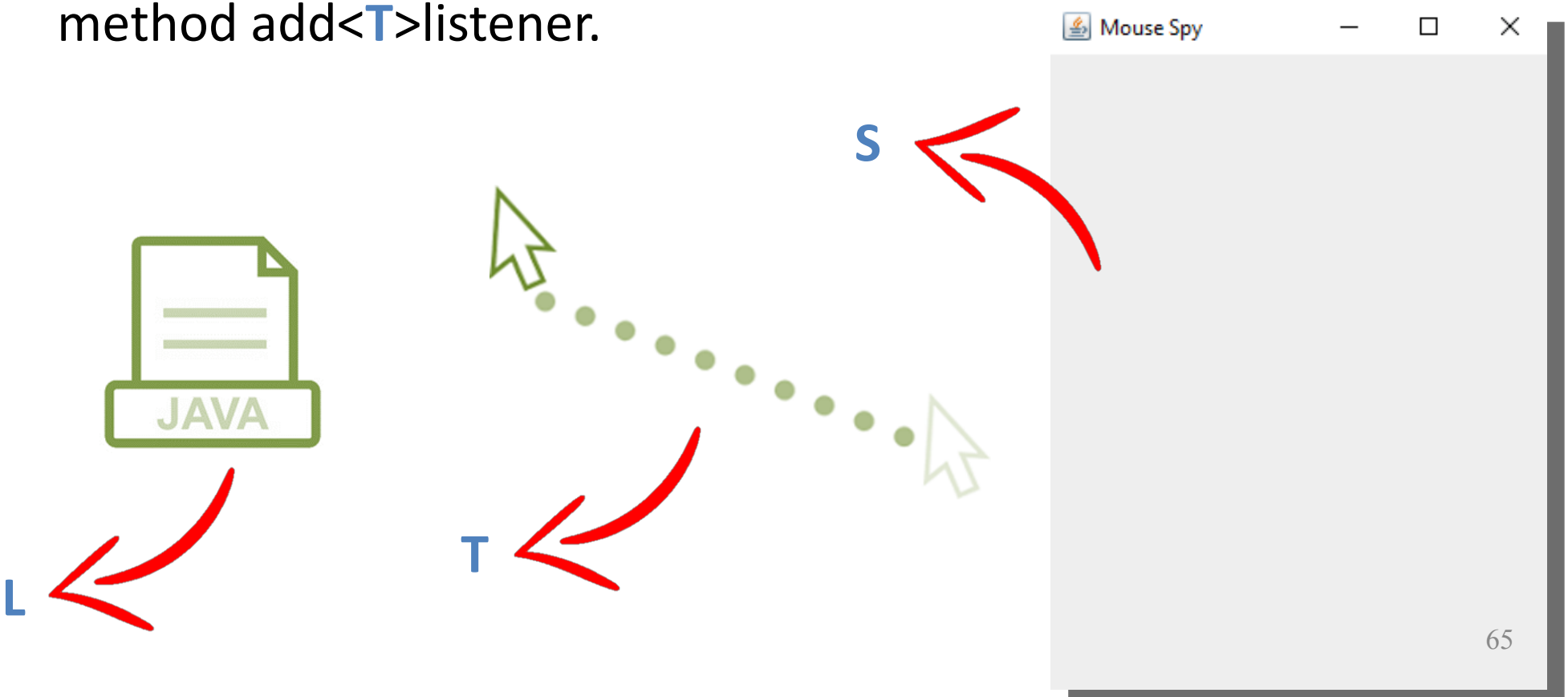
- Assume that we have:
  - an event-source object **S** that generates events of type **T**;
  - an event listener **L** for events of type **T**;
- Then, we add **L** to listen events from **T** generated in **S** using a method add<**T**>listener.
- In this case the Java window manager uses the loop:

while (true)

- » **wait for next event;**
- » **determine event type T, and generate an object E with type event T ;**
- » **determine GUI source object S where event occurred;**
- » **call appropriate method of the listener L added to S with factual parameter the object E.**

Event sources & event-handling model

- Assume that we have:
  - an event-source object **S** that generates events of type **T**;
  - an event listener **L** for events of type **T**;
- Then, we add **L** to listen events from **T** generated in **S** using a method add<**T**>listener.



Example: `MouseSpyViewer`

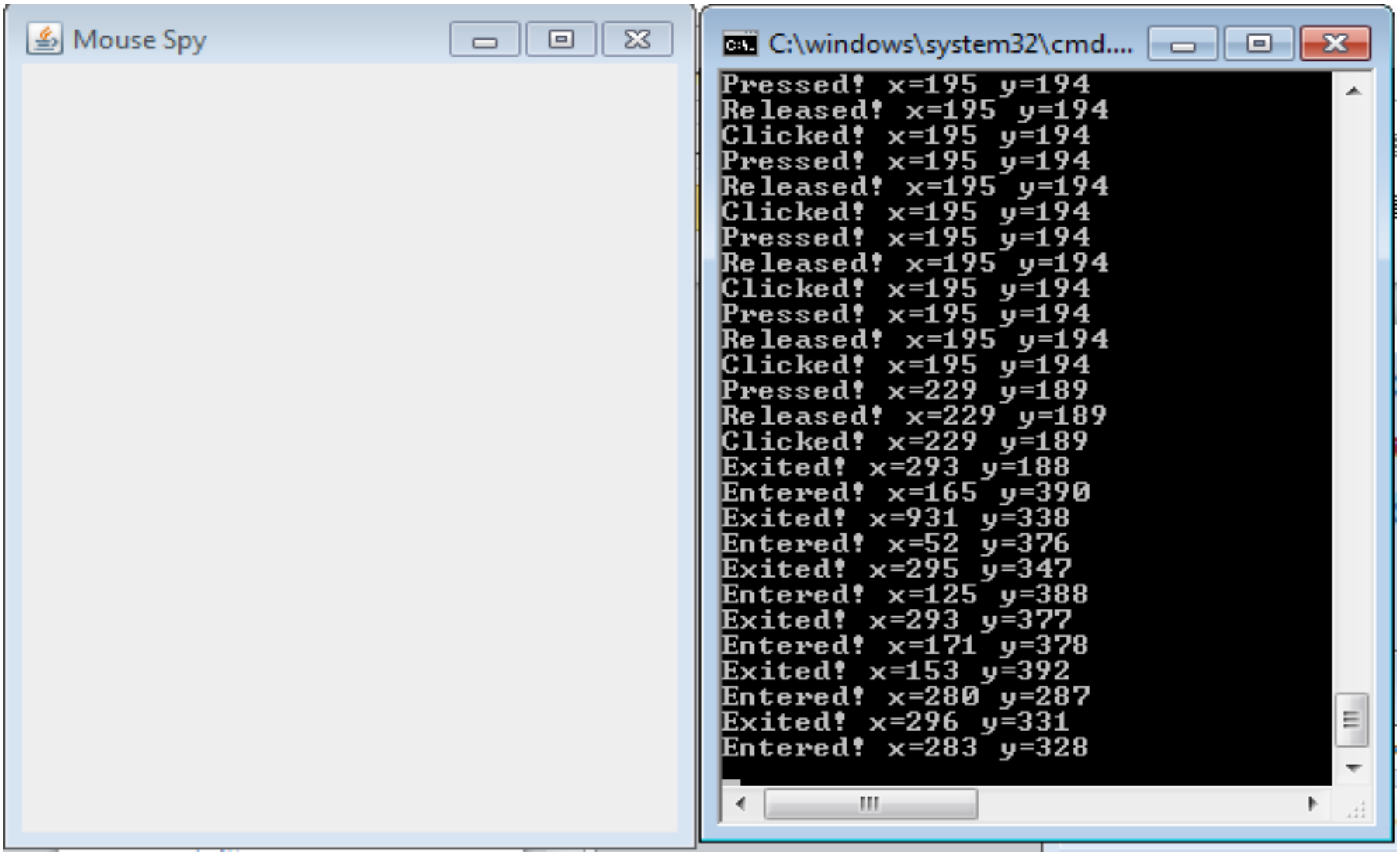
- The mouse is a source of events
- There are too many to mention, but we will look at
  - when a mouse button has been clicked
  - when a mouse button has been pressed on a component
  - when a mouse button has been released on a component
  - when the mouse enters a component
  - when the mouse exits a component
- First we will need a component for the mouse to interact with
- Then we will write a listener that will receive the events so they can be processed

Example: `MouseSpyViewer`

Write an application that prints in the console window the `X` and `Y` coordinates of the dot pointed at by the computer mouse. In this case:

- The event class is the standard **`MouseEvent`** class;
- The event source is an object of some frame class (**`MouseSpyViewer`**);
- The listener class (**`MouseSpy`**) implements the standard **`MouseListener`** interface.

Example: MouseSpyViewer



The MouseSpy Class

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

class MouseSpy implements MouseListener
{   public void mousePressed(MouseEvent event)
    { System.out.println("Pressed! x=" + event.getX() + " y=" + event.getY()); }

    public void mouseReleased(MouseEvent event)
    { System.out.println("Released! x=" + event.getX() + " y=" + event.getY()); }

    public void mouseClicked(MouseEvent event)
    { System.out.println("Clicked! x=" + event.getX() + " y=" + event.getY()); }

    public void mouseEntered(MouseEvent event)
    {  System.out.println("Entered! x=" + event.getX() + " y=" + event.getY()); }

    public void mouseExited(MouseEvent event)
    { System.out.println("Exited! x=" + event.getX() + " y=" + event.getY()); }
}
```


https://docs.oracle.com/javase/7/docs/api/java/awt/event/MouseEvent.html#method_summary

The MouseSpyViewer Class

```
import javax.swing.*;

public class MouseSpyViewer extends JFrame {
        public MouseSpyViewer() {
                MouseSpy listener = new MouseSpy();
                 addMouseListener(listener);
         }

        public static void main(String[] args) {
                MouseSpyViewer frame = new MouseSpyViewer();
                frame.setSize(300, 400);
                frame.setTitle("Mouse Spy");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
        }
}
```

INNER CLASSES IN LISTENERS

Inner classes

[…] This is a class that is declared within another class. - Java: A Beginner's Guide, Eighth Edition

But WHY???!!!

- Java does not permit multiple inheritances
- Useful when using adapters (more later)

Interesting link:
http://www.fredosaurus.com/notes-java/GUI/events/inner_class_listener.html

The use of inner classes in listeners

- Class inside another class
- Inner class can access variables and methods of the outer class

```
public class ButtonFrame extends JFrame
{
   private JButton button;
   private JLabel label;

   class ClickListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         // We can modify label from the inner class
         label.setText("Button clicked");
      }
   }
}
```

The use of inner classes in listeners

```
class MyClass {
    public MyClass() {
    // begin inner class
        class MyListener implements ListenerInterface {
            public void eventOccured(EventClass event)
            {
             // event actions go here!
            }
        } // end inner class
        MyListener listener = new MyListener();
        anEventSource.addListener(listener);
    } // Outer class methods and variables
} // end outer class
```

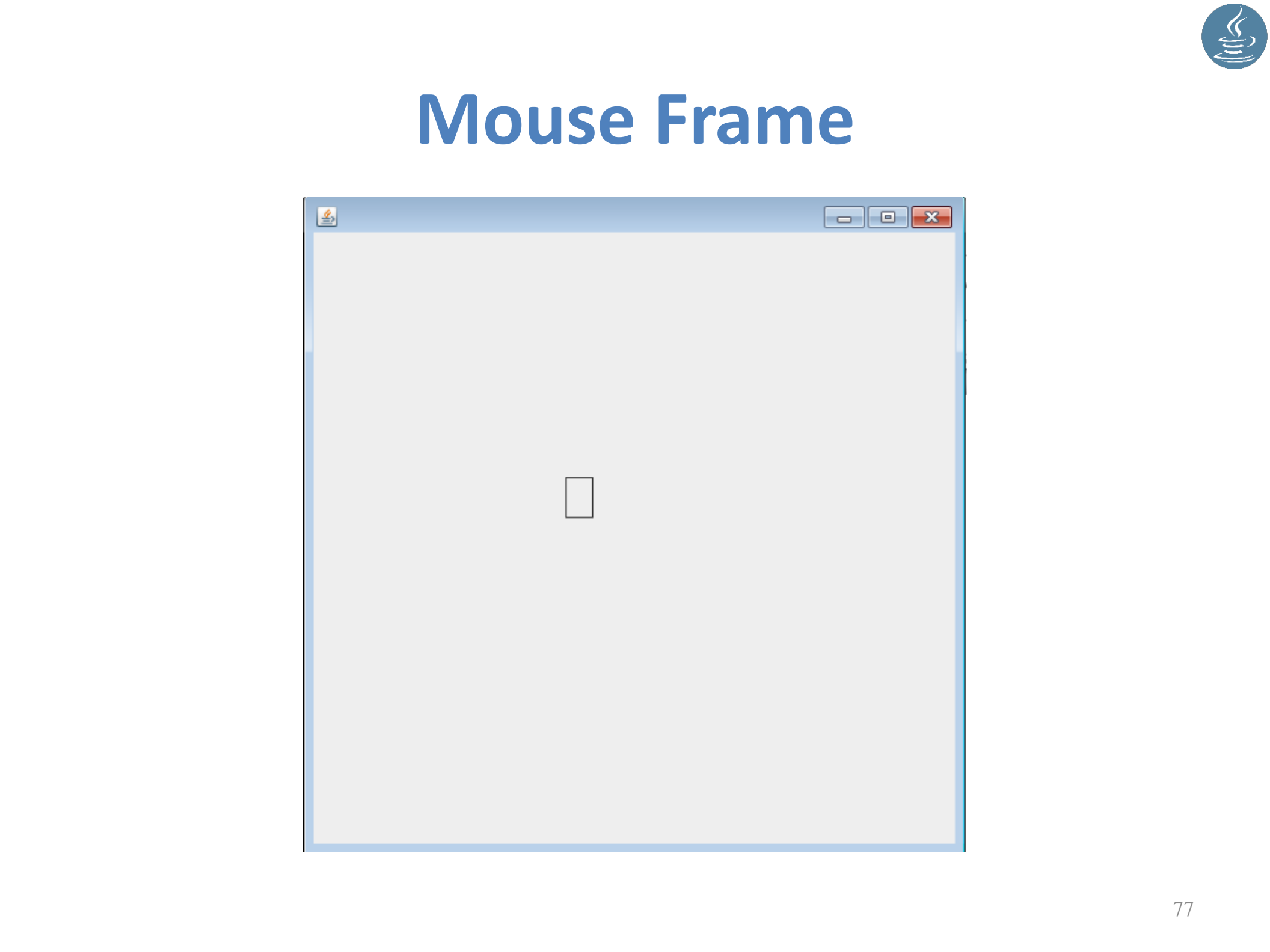
[Interlude: *extends vs implements*]

extends	implements
A keyword available in Java programming language that allows a class to use the features of an already existing class	A keyword available in Java programming language that allows a class to provide definitions to the abstract methods of an interface
A class can extend one superclass	Class can implement one or more interfaces
An interface can extend one or more interfaces	An interface cannot implement another interface
Associated with inheritance	Associated with abstraction

Example: Mouse Frame

Write an application that moves a rectangle to the mouse press position. In this case:

- The event class is the standard **`MouseEvent`** class;
- The event source is an object of some component class (**MouseComponent**);
- The listener class (an inner class **MousePressListener**) implements the standard **MouseListener** interface.


Mouse Frame

Mouse Component

```
public class MouseComponent extends JComponent {
    private Rectangle box;
    public MouseComponent() {
        box = new Rectangle(100, 100, 20, 30);
        class MousePressListener implements MouseListener {
            public void mousePressed(MouseEvent event) {
                int x = event.getX();
                int y = event.getY();
                box.setLocation(x, y);
                repaint(); // repaints the component
            }
            public void mouseReleased(MouseEvent event) {}
            public void mouseClicked(MouseEvent event) {}
            public void mouseEntered(MouseEvent event) {}
            public void mouseExited(MouseEvent event) {}
        }
        MousePressListener listener = new MousePressListener();
        addMouseListener(listener);
    }
    public void paintComponent(Graphics g) {
        Graphics2D g2 = (Graphics2D) g;
        g2.draw(box);
    }
}
```

Mouse Frame

```
import javax.swing.JFrame;

public class MouseFrame {
    public static void main(String[] args) {
        MouseComponent comp = new MouseComponent();
        JFrame frame = new JFrame();
        frame.add(comp);
        frame.setSize(400, 400);
        frame.setVisible(true);
    }
}
```

MOUSE ADAPTER CLASS

Adapter Classes

- Adapter classes provide the default implementation of listener interfaces
- If you inherit the adapter class, you will not be forced to provide the implementation of all the methods of listener interfaces
- So it saves time/code

➢ *Revise interfaces in Lecture 2!*

https://developer.classpath.org/doc/java/awt/event/MouseAdapter-source.html

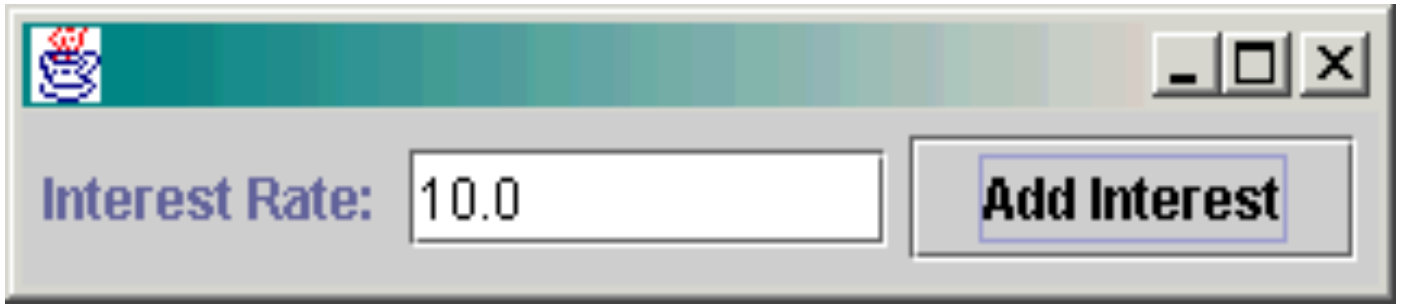
The MouseAdapter Class

- To avoid writing empty methods in your listeners, use the standard **MouseAdapter** class;
- This class implements the **MouseListener** interface, but all the methods are empty;
- Thus, when you extend your listener with the **MouseAdapter** class, you write only nonempty methods:

```
class MousePressListener extends MouseAdapter {
        public void mousePressed(MouseEvent event) {
                int x = event.getX();
                int y = event.getY();
                box.setLocation(x, y);
                repaint();// repaints the applet
        }
}
```

GUI COMPONENTS EXAMPLE

Graphical applications: an example

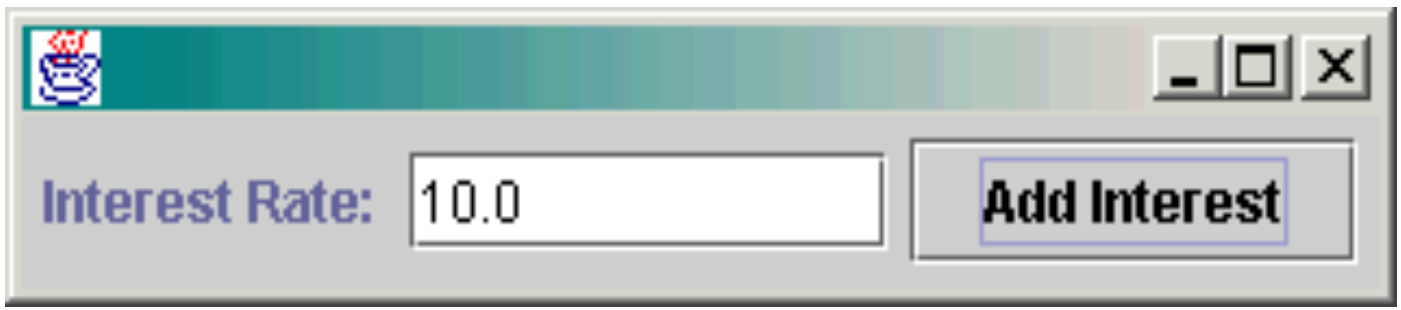



```
textArea.append(String.format("%.2f\n", account.getBalance()));
```

Classes for GUI components

- Use package `javax.swing`
- Class names for components start with **J**

```
JTextField rateField = new JTextField(10);
JLabel xLabel = new JLabel("Interest Rate:
");
JButton calButton = new
JButton("AddInterest");
```



JPanel

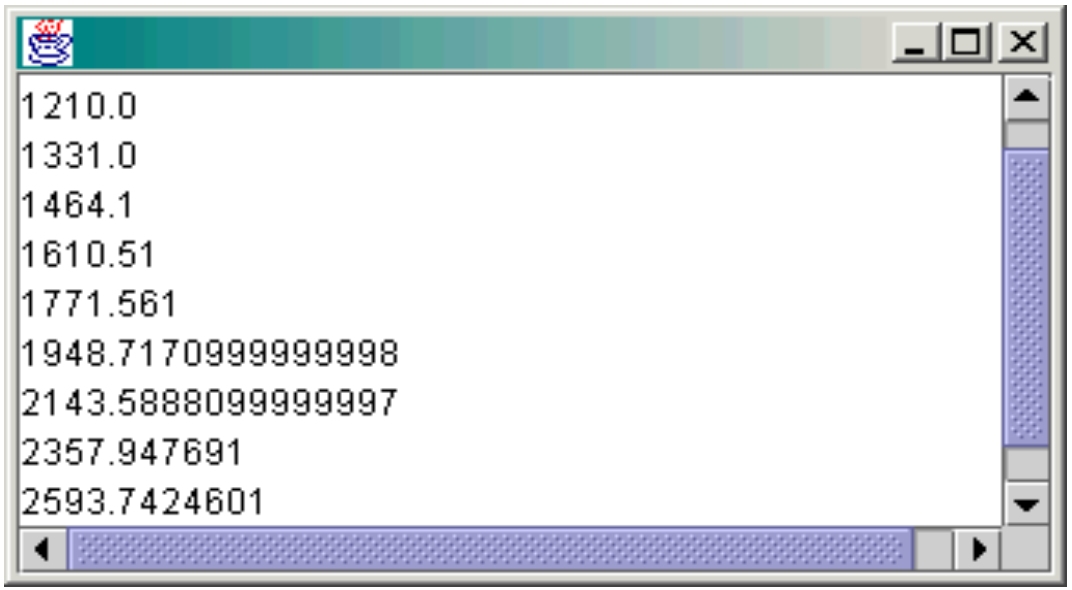
- Use package `javax.swing`
- JFrame: heavyweight container used as the top-level window
- JPanel: lightweight container generally used to organize/group Graphic user interface components
- JPanels are added on top of JFrame, whereas graphical user interface components are added on one or more JPanels.

```
JPanel controlPanel = new JPanel();
JFrame controlFrame = new JFrame();
controlFrame.add(controlPanel);
```

https://www.educba.com/jpanel-vs-jframe/

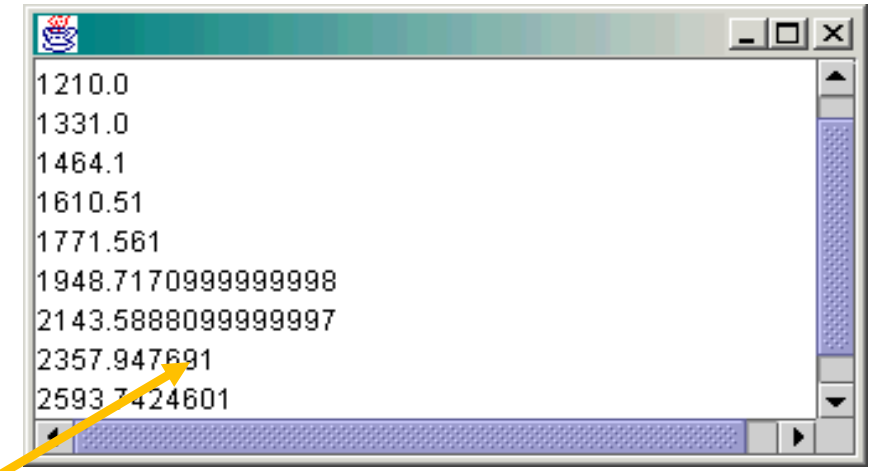
Text components

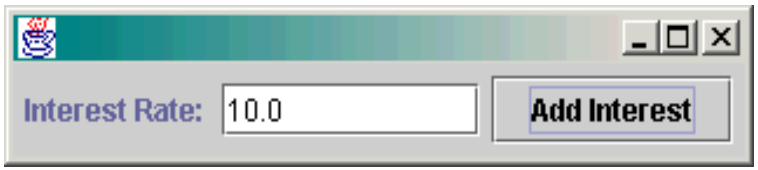
- To construct a text area use the **`JTextArea`** class ;
- To construct a text area with `n` rows and `m` columns use the constructor: **`JTextArea (int n, int m)`**
  - To append a string to a text area use the method **`append(String str)`** that appends the given text to the end of the text area.
- To (dis)allow the user to edit a text area use the methods: **`setEditable(boolean flag)`**
- To set the font of a text area use: **`setFont(Font f)`**
- To add scroll bars create a **`JScrollPane`** object with constructor parameter the reference of the **`JTextArea`** object.



Text area example

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

public class TextAreaTest {
  public static void main(String[] args) {
    BankAccount account = new BankAccount(1000);
    JTextArea textArea = new JTextArea(10, 30);
    JScrollPane scrollPane = new JScrollPane(textArea);
    JFrame frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.add(scrollPane);
    frame.setSize(200, 100);
    frame.setVisible(true);
    JLabel rateLabel = new JLabel("Interest Rate: ");
    JTextField rateField = new JTextField(10);
    JButton calButton = new JButton("Add Interest");
    class CalculateListener implements ActionListener {
      public void actionPerformed(ActionEvent event) {
        double rate = Double.parseDouble(rateField.getText());
        account.deposit((account.getBalance()*rate/100));
        textArea.append(account.getBalance() + "\n");
      }
    }
    ActionListener listener = new CalculateListener();
    calButton.addActionListener(listener);
    JPanel controlPanel = new JPanel();
    controlPanel.add(rateLabel);
    controlPanel.add(rateField);
    controlPanel.add(calButton);
    JFrame controlFrame = new JFrame();
    controlFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    controlFrame.add(controlPanel);
    controlFrame.setSize(220, 100);
    controlFrame.setVisible(true);
  }
}
```





Recommendation

- Get familiar with the use of Java documentation
  - Several constructors and (overloaded) methods are available per each component
  - You can learn a lot from it
  - It saves time!
- Practice: some examples (you may need Lecture 5)
  - https://personales.unican.es/corcuerp/java/Labs/LAB_18.htm
  - https://www3.ntu.edu.sg/home/ehchua/programming/java/GraphicsExercises.html

Learning goals

- Identify differences between console and *graphical application*
- Be able to configure and use *basic GUI components*
- Understand and be able to utilize commonly use *graphical classes*
- Understand and be able to implement the *event-handling model*
- Understand and be able to implement and use *inner classes*
- Implement *listener-based applications*
- Identify differences between *extends* and *implements*
- Identify the benefits of using *interfaces/adapters*
- Identify differences between *interfaces* and *adapters*