ISCTE IUL
Instituto Universitário de Lisboa

Nested/Inner Classes
More on SWING

Programação Concorrente e Distribuída
Parallel and Distributed Programming

2013-2014

To read before class

Understanding Instance and Class Members:

http://download.oracle.com/javase/tutorial/java/javaOO/classvars.html

After this class you will be able to…

- Understand all Java inner classes.
- Use inner classes (ex. implement listeners).
- See the basic mechanisms behind the JComponent.
- Draw directly on the JComponent.
- Understand and use mouse events.
- **Implement a basic “Paint” application.**

NESTED CLASSES

Nested Classes

Modern class-based object-oriented languages (e.g., C++, Java) support class nesting as a way of structuring code.

Nested Classes

- Nested classes are classes defined inside other classes
- They can be static or non-static (inner)
- They can be private, public, protected or package (by default)

Inner Classes

- Inner classes are associated with the external instance.

- They are member classes of the external class and share a trust relationship so they have access to all its attributes and methods.

Static Nested Classes

- Static nested classes act as top-level classes. They can always be instantiated.

- Static nested classes cannot access dynamic attributes or methods of the external class.

Example of nested classes

```
class OuterClass {

    ...

    static class StaticNestedClass {
        ...
    }

    class InnerClass {
        ...
    }
}
```

Why do we need them?

- Allows for a logically grouping classes that are only used in one place (example: the use of an ActionListener)
- It increases encapsulation (access to private information by classes that may still be private allowing to hide the implementation)
- Nested classes can lead to more readable and maintainable code.

Why do we need them?

“For example, a tree class may have a method and many helper methods that perform a search or walk of the tree. From an object-oriented point of view, the tree is a tree, not a search algorithm. However, you need intimate knowledge of the tree's data structures to accomplish a search.

An inner class allows us to remove that logic and place it into its own class. So from an object-oriented point of view, we've taken functionality out of where it doesn't belong and have put it into its own class. “

“Inner classes. So what are inner classes good for anyway?”
By Tony Sintes, JavaWorld.com, 07/27/01

Example: Subscription of an action listener

```
public class Botoes {
     public Botoes() {
          ...
          private SentinelaParaAcçoes sentinela =
                   new SentinelaParaAcçoes();

          // Regista sentinelas:
          botaoOK.addActionListener(sentinela);
          botaoCancel.addActionListener(sentinela);
     }
     private class SentinelaParaAcçoes
                   implements ActionListener {
          public void actionPerformed(ActionEvent e) {
               JButton botao=(JButton)e.getSource();
               if (botao==botaoOK ) {  ...}
               else {...}
          }
     }
     ...
}
```

Static nested classes

- This type of class is used mainly to increase encapsulation and modularity.
- Example of a list node:

```
public class ListaInteger {
  ...
  public static class No {
        Integer i;
        No proximo;
        No anterior;
  }
}
```

- As for static methods or variables a static nested class cannot directly access the dynamic elements (methods and methods) of the external class. It always needs to receive a reference to an object to call it’s methods and access it’s variables.

Using Static nested classes

- Their name has the external class name as a prefix

  ex: `OuterClass.StaticNestedClass`

- To instantiate a static nested class object, we must do the following:

```
OuterClass.StaticNestedClass nestedObject =
             new OuterClass.StaticNestedClass();
```

Inner Classes

Can only be instantiated within an instance of the external class

```
class OuterClass {
   ...
   class InnerClass { ... }
}
```

Inner Classes

To instantiate the inner class we need to do the following:

```
// create an instance of the external class
OuterClass outerObject = new OuterClass();

// create an instance of the inner class
OuterClass.InnerClass innerObject =
       outerObject.new InnerClass();
```

Inner Classes

Besides the general inner type class there are two additional specific inner type classes:

- <u>Local classes</u>: defined inside a method.

- <u>Anonymous classes:</u> defined, without a name, inside a method.

Local Classes

```
interface Destination {
      String readLabel();
}
public class Parcel {
      public Destination dest(String s) {
            class PDestination implements Destination {
                  private String label;
                  private PDestination(String whereTo) {
                        label = whereTo;
                  }
                  public String readLabel() {
                        return label;
                  }
            }
            return new PDestination(s);
      }

      public static void main(String[] args) {
            Parcel p = new Parcel();
            Destination d = p.dest("Tanzania");
      }
}
```

Anonymous Classes

```
interface Contents {
     int value();
}

public class Parcel {
     public Contents cont() {
          return new Contents() {
               private int i = 11;
               public int value() { return i; }
          }; // ";” it is an instruction!
     }

     public static void main(String[] args) {
          Parcel p = new Parcel();
          Contents c = p.cont();
     }
}
```

Why anonymous classes?

Imagine an action listener for multiple buttons:

```
public class SomeGUI extends JFrame implements ActionListener
  {
    private JButton button1;
    private JButton button2;
    ...
    public void actionPerformed(ActionEvent e) {
     if(e.getSource()==button1) {
          // do something
     } else
          if(e.getSource()==button2) {
          ... you get the picture ...
     }
  }
```

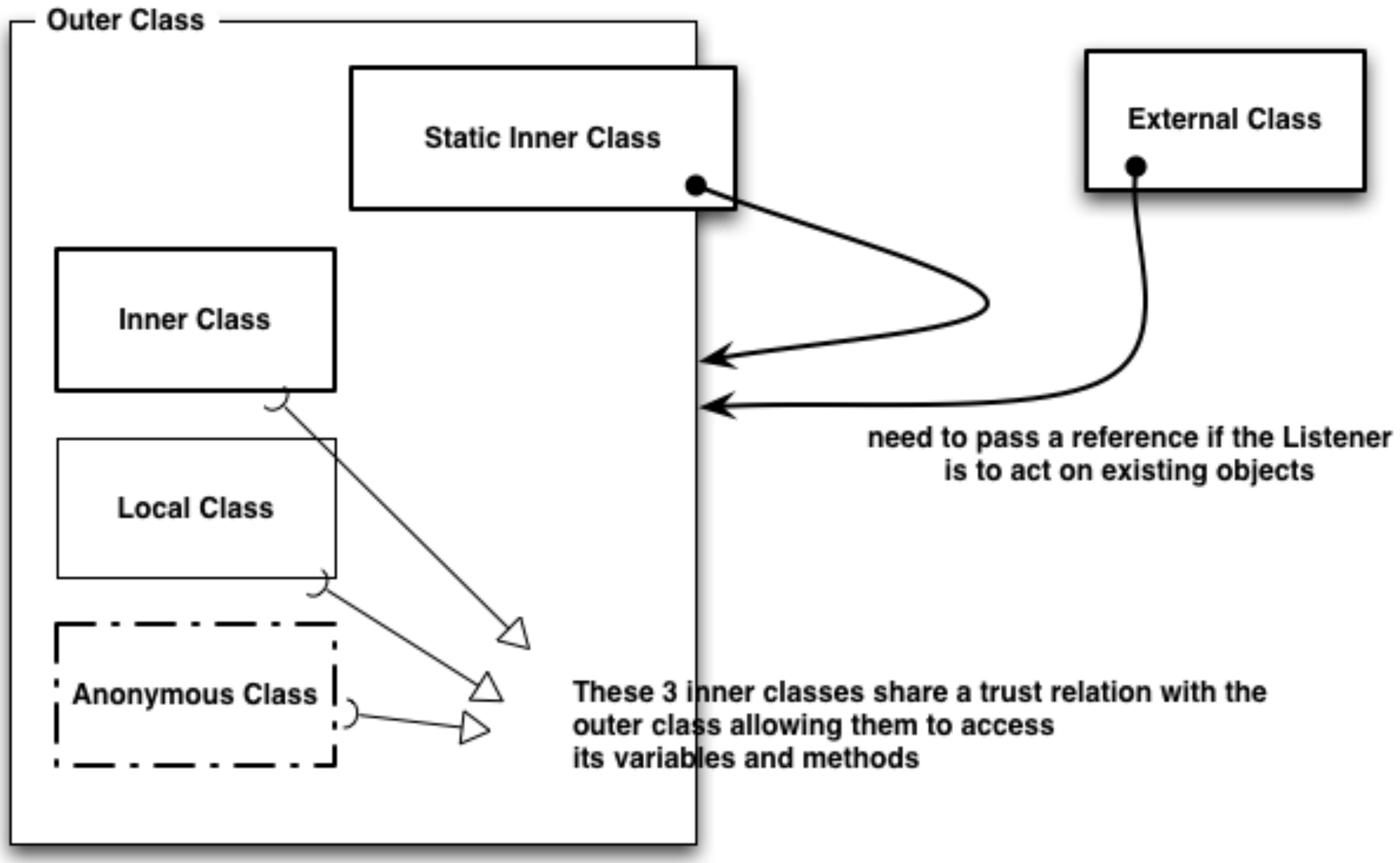
Why anonymous classes?

- Huge if/else blocks, unclear code.
- Changes can be complex and deep.
- Would it be beneficial/easier to define a different action listener for each button?

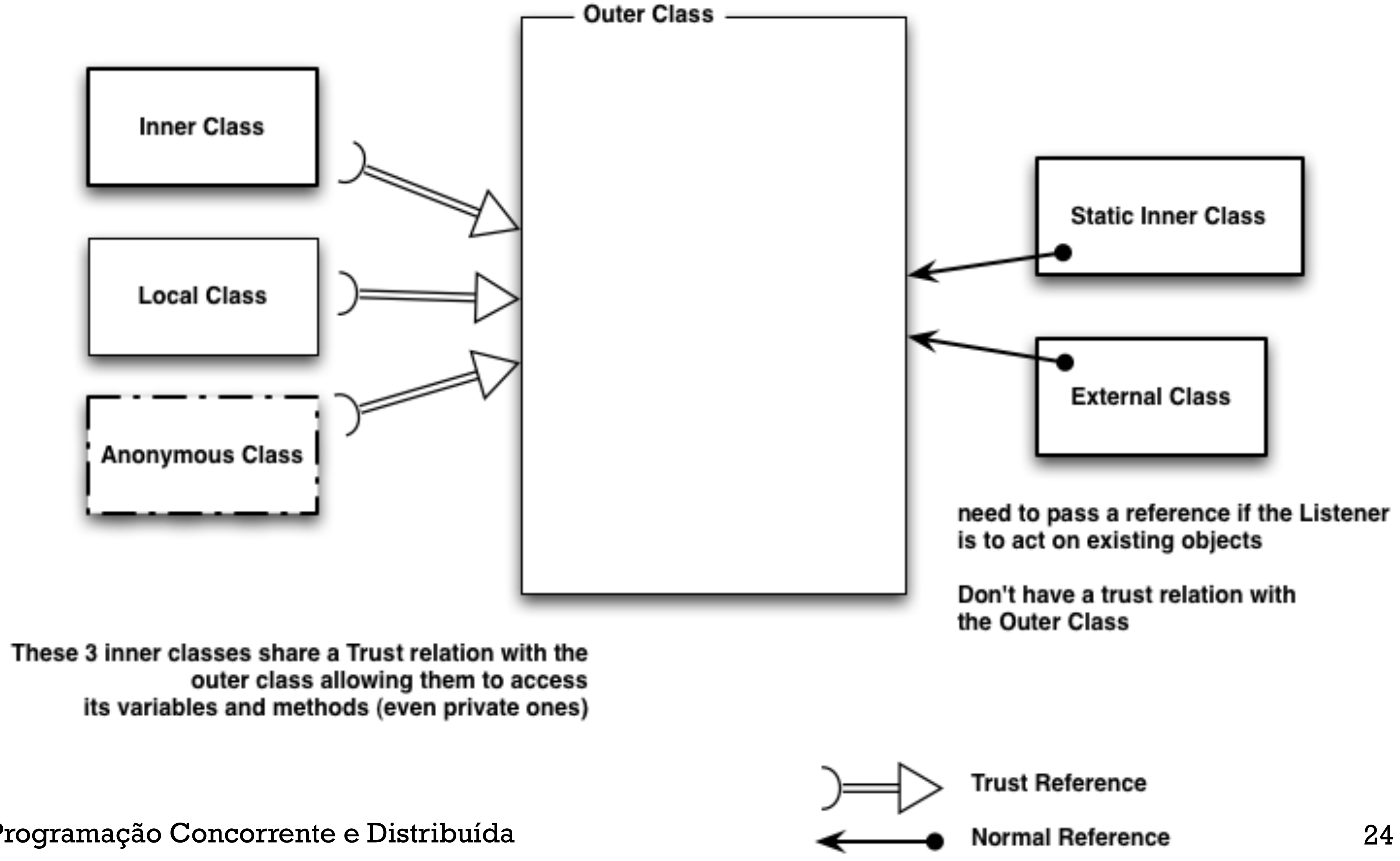
Anonymous classes?

```
public class SomeGUI extends JFrame {
     // ... button member declarations ...
     protected void buildGUI()
     {
        button1 = new JButton();
        button2 = new JButton();
        ...
        button1.addActionListener(
                      new ActionListener() {
             public void actionPerformed(
                                    ActionEvent e){
                // do something
             }
        });
        // .. repeat for each button
```

Nested classes


Outer Class
Static Inner Class
External Class
Inner Class
Local Class
Anonymous Class
need to pass a reference if the Listener is to act on existing objects
These 3 inner classes share a trust relation with the outer class allowing them to access its variables and methods

Inner/Outer trust relation


Outer Class
Inner Class
Local Class
Anonymous Class
Static Inner Class
External Class
need to pass a reference if the Listener is to act on existing objects
Don't have a trust relation with the Outer Class
These 3 inner classes share a Trust relation with the outer class allowing them to access its variables and methods (even private ones)
Trust Reference
Normal Reference

Bibliography

- The Java Tutorial, *Nested Classes*

http://download.oracle.com/javase/tutorial/java/javaOO/nested.html

- "The Java™ Programming Language, Fourth Edition", by Ken Arnold, James Gosling, and David Holmes

(Chapter 5)

Bibliografia auxiliar

- The Java World

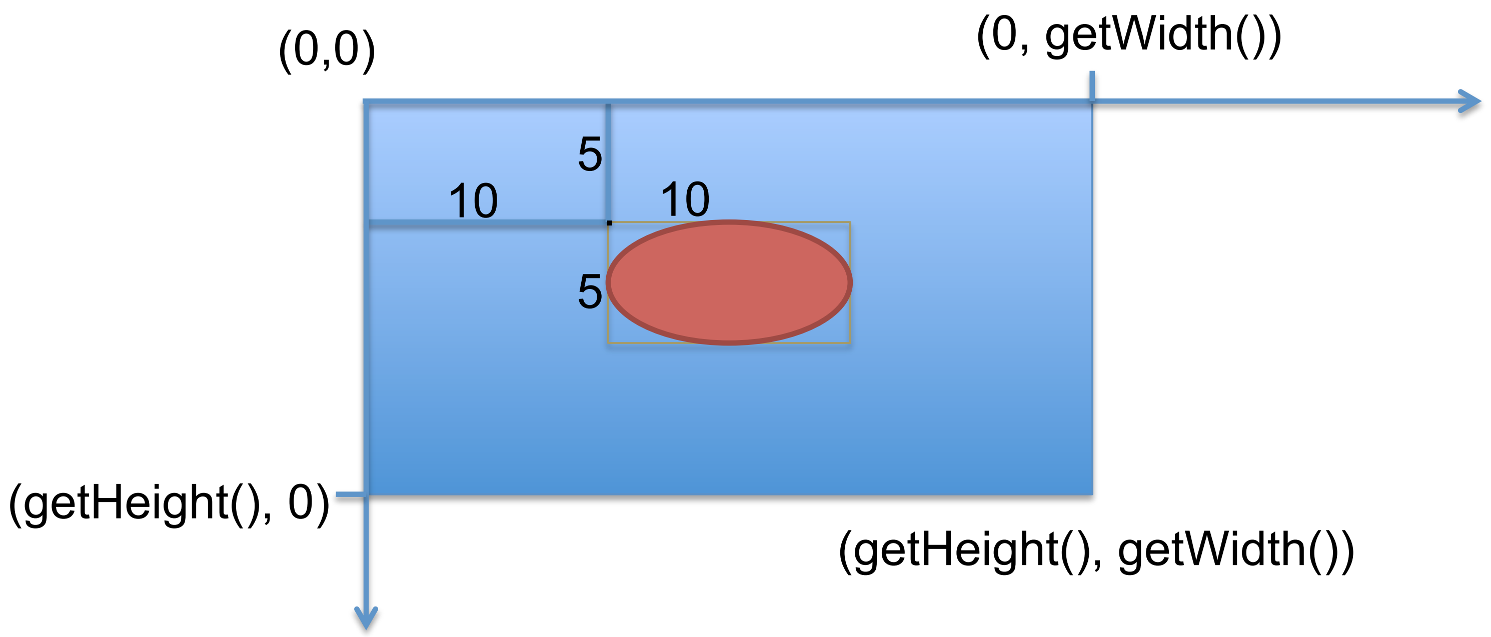
http://www.javaworld.com/javaworld/javaqa/2000-03/02-qa-innerclass.html

- “Thinking in JAVA”, Bruce Eckel

MORE ON SWING

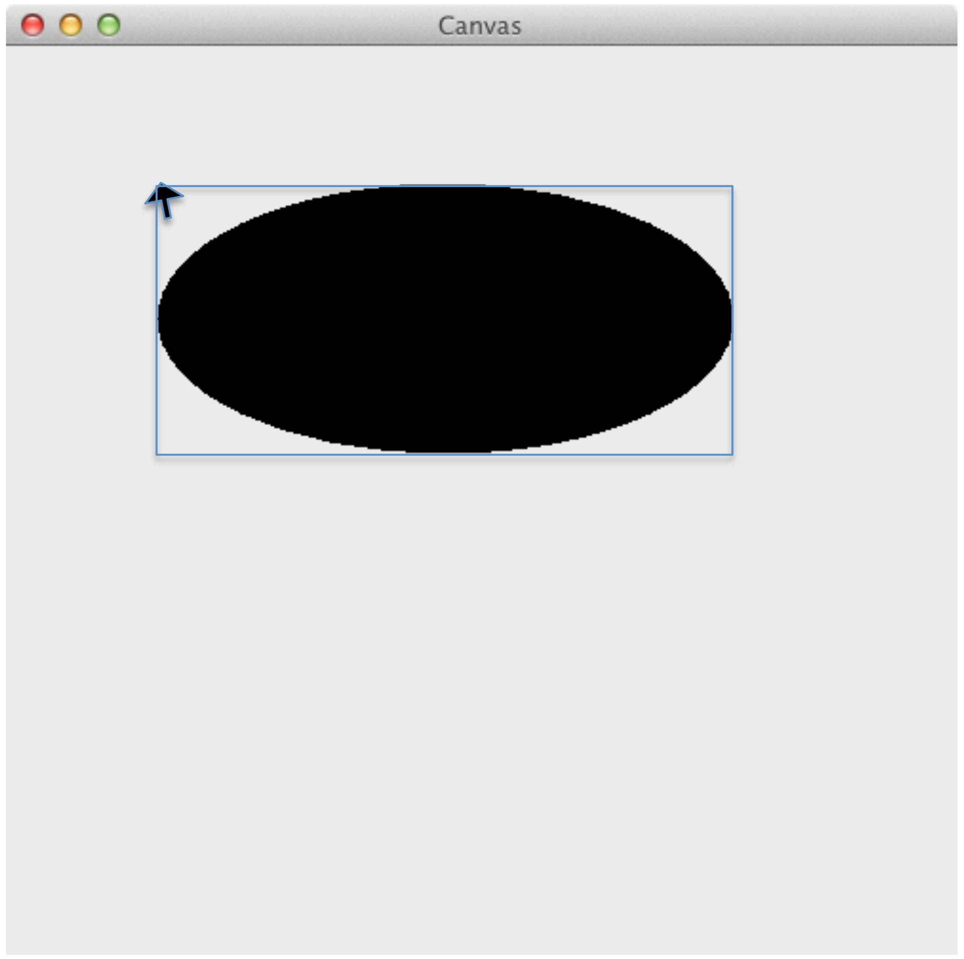
JComponent

[java.lang.Object]()
    [java.awt.Component]()
        [java.awt.Container]()
            javax.swing.JComponent



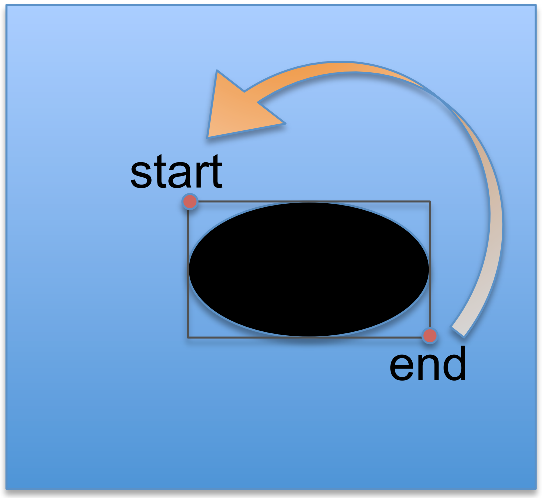
```
public void paintComponent(Graphics g){
    super.paintComponent(g);
    g.fillOval(10, 5, 10, 5);
}
public void repaint()
```

Example: MyCanvas



MyCanvas

```
public class MyCanvas extends JLabel {

    private Point start;
    private Point end;

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        if (start != null && end != null) {
            int xi = Math.min(start.x, end.x);
            int yi = Math.min(start.y, end.y);
            int dx = Math.abs(start.x - end.x);
            int dy = Math.abs(start.y - end.y);

            g.fillOval(xi, yi, dx, dy);
        }
    }
}
```



Example (Canvas)

```
public class CanvasDemo{

   private JFrame frame = new JFrame("Canvas");
   private MyCanvas canvas = new MyCanvas();

   public CanvasDemo() {
      frame.getContentPane().add(canvas);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setSize(500, 500);
      canvas.addMouseListener(…);
   }
   public void init() {
      frame.setVisible(true);
   }
   public static void main(String[] args) {
      new CanvasDemo().init();
   }
}
```

Example (Canvas)

```
canvas.addMouseListener(new MouseListener() {
      @Override
      public void mouseReleased(MouseEvent event) {}

      @Override
      public void mousePressed(MouseEvent event) {}

      @Override
      public void mouseExited(MouseEvent arg0) {}

      @Override
      public void mouseEntered(MouseEvent arg0) {}

      @Override
      public void mouseClicked(MouseEvent arg0) {}

});
```

Example (Canvas)

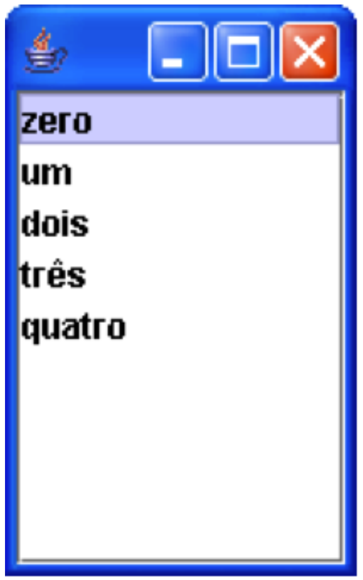
```
…

@Override
public void mouseReleased(MouseEvent event) {
   canvas.setEnd(event.getPoint());
   canvas.repaint();
}

@Override
public void mousePressed(MouseEvent event) {
   canvas.setStart(event.getPoint());
}

…
```

Example (Canvas)

```
canvas.addMouseMotionListener(new
                        MouseMotionListener() {

      @Override
      public void mouseMoved(MouseEvent arg0) {}

      @Override
      public void mouseDragged(MouseEvent event) {
         canvas.setEnd( event.getPoint());
         canvas.repaint();

      }
});
```

An example with lists...

- `javax.swing.JList`
- `javax.swing.JScrollPane`
- `javax.swing.ListSelectionModel`
- `javax.swing.event.ListSelectionEvent`
- `javax.swing.event.ListSelectionListener`


zero
um
dois
três
quatro

List, items and the listener

```
public class Listador {

      private static final String[] nomesDoItens = {
            "zero",
            "um",
            "dois",
            "três",
            "quatro"
      };


      private int índiceDoItemSeleccionado = 0;
      private JFrame janela = new JFrame("Listas");

      private JList lista = new JList(nomesDoItens);
      private SentilenaParaALista sentinela =
                  new SentilenaParaALista();
```

(continues)

Adding the list and setting up the window

```
public Listador() {
        janela.getContentPane().add(new JScrollPane(lista));
        lista.setSelectionMode(
                ListSelectionModel.SINGLE_SELECTION);
        lista.setSelectedIndex(0);
        lista.addListSelectionListener(sentinela);


  janela.setSize(100, 200);
  janela.setLocation(200, 100);
  janela.setDefaultCloseOperation(
                              JFrame.EXIT_ON_CLOSE);
}


public void executa() {
        janela.setVisible(true);
}
```

(continues)

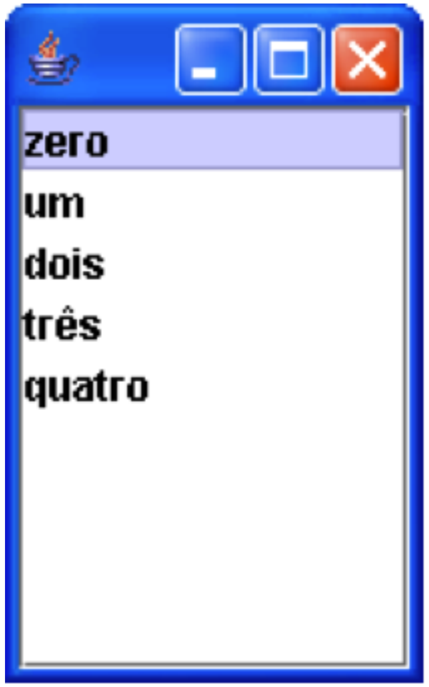
Handling events

(continued)

```
    private class SentilenaParaALista implements
                            ListSelectionListener {

        public void valueChanged(ListSelectionEvent e) {
            if(índiceDoItemSeleccionado !=
                          lista.getSelectedIndex()) {

                System.out.println(lista.getSelectedIndex()
                                   + " --> "
                                   + lista.getSelectedValue());

                índiceDoItemSeleccionado =
                               lista.getSelectedIndex();
            }
        }
    }
```

(continues)

```
(continuation)

    public static void main(String[] args) {
        Listador l = new Listador();
        l.executa();
    }
}
```



```
zero
um
dois
três
quatro
```

SWING

(Modified) Model View Controller Design Pattern

MVC pattern divides an application into three parts: a model, a view and a controller.

- **The model** represents the data in the application.
- **The view** is the visual representation of the data.
- **The controller** processes and responds to events, typically user actions, and may invoke changes on the model.

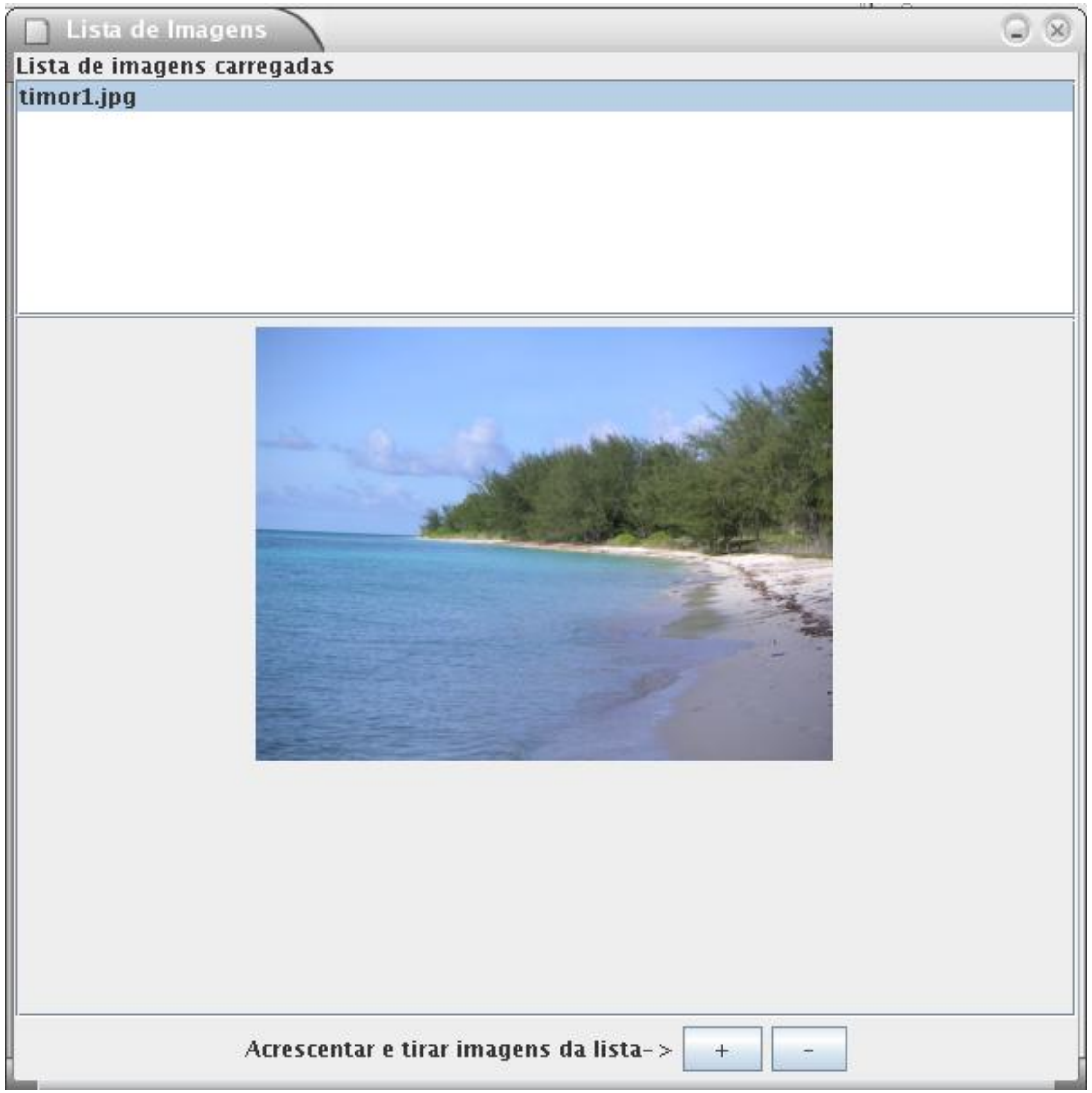
http://www.oracle.com/technetwork/java/architecture-142923.html

List Model

- **`ListModel`** – model interface
- **`AbstractListModel`** - abstract class with most list model functionalities implemented
- **`DefaultListModel`** - concrete class for a default list model

```
private DefaultListModel listModel= new DefaultListModel ();
public Listador() {
     …
     listModel = new DefaultListModel();
     listModel.addElement("zero");
     listModel.addElement(”um");
     listModel.addElement("dois");
     …
     lista = new JList(listModel);
     …
}
```

Exercício - Album de Fotografias


Lista de Imagens
Lista de imagens carregadas
timor1.jpg
Acrescentar e tirar imagens da lista- >
+
-

Summary

Nested Classes

- Static nested classes
- Inner classes
- Local and anonymous classes
- Examples

Swing

- Basic JComponent (`paintComponent()`)
- Mouse events
- MVC

Bibliography

- Nested Classes - The Java™ Tutorials:
  http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html
- So what are inner classes good for anyway?
  http://www.javaworld.com/javaworld/javaqa/2000-03/02-qa-innerclass.html

- A Swing Architecture Overview:
  http://www.oracle.com/technetwork/java/architecture-142923.html
- Model–view–controller:
  http://en.wikipedia.org/wiki/Model–view–controller