PIC 20A
GUI with swing

David Hyde
UCLA Mathematics

Last edited: April 29, 2020

Hello swing

Let's create a JFrame.

```java
import javax.swing.*;

public class Test {
  public static void main(String[] args) {
    //JFrame is javax.swing.JFrame
    JFrame frame = new JFrame("Hello Swing");
    //intput is the frame's title
  }
}
```

Nothing happened since JFrame isn't visible.

Hello swing

Since JFrame is an object, we change its state with a member function.

```
import javax.swing.*;

public class Test {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Hello Swing");
    frame.setVisible(true);
  }
}
```

We now see a window.

Hello swing

Roughly speaking, the face of the window is called the *content pane*.

```
import javax.swing.*;

public class Test {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Hello Swing");
    frame.setVisible(true);
    java.awt.Container contPane
                        = frame.getContentPane();
  }
}
```

The content pane is an object of type `java.awt.Container`.

Hello swing

Let's add a JLabel to the content pane.

```java
import javax.swing.*;

public class Test {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Hello Swing");
    frame.setVisible(true);
    java.awt.Container contPane
                         = frame.getContentPane();
    JLabel label = new JLabel("Hello World");
    contPane.add(label);
  }
}
```

The add method, inherited from java.awt.Container, takes in java.awt.Component as its input.
JLabel inherits Component.

Hello swing

You don't have to keep references to the content pane and the `JLabel`.

```
import javax.swing.*;

public class Test {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Hello Swing");
    frame.setVisible(true);
    frame.getContentPane().add(
                    new JLabel("Hello World") );
  }
}
```

Hello swing

It's good preactice to not display a GUI until you're done building it.

```
import javax.swing.*;

public class Test {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Hello Swing");
    frame.getContentPane().add(
                      new JLabel("Hello World") );
    frame.setVisible(true);
  }
}
```

If the time it takes to build the GUI is perceivable, the user can see the GUI components popping into existance.

Hello swing

Use setSize to set window size.

```
import javax.swing.*;

public class Test {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Hello Swing");
    frame.getContentPane().add(
                      new JLabel("Hello World") );
    frame.setSize(300, 200);
    frame.setVisible(true);
  }
}
```

JFrame inherits setSize from its parent class java.awt.Component.

Hello swing

Make the program exit when you close the JFrame.

```java
import javax.swing.*;

public class Test {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Hello Swing");
    frame.setDefaultCloseOperation(
                       JFrame.DISPOSE_ON_CLOSE);
    //DISPOSE_ON_CLOSE is a constant of JFrame
    frame.getContentPane().add(
                    new JLabel("Hello World") );
    frame.setSize(300, 200);
    frame.setVisible(true);
  }
}
```

We'll talk about setDefaultCloseOperation more later.

Hello swing

Instead of adding a JLabel, we can add a JButton.

```
import javax.swing.*;

public class Test {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Hello Swing");
    frame.setDefaultCloseOperation(
                         JFrame.DISPOSE_ON_CLOSE);
    frame.getContentPane().add(
                       new JButton("Do nothing") );
    frame.setSize(300, 200);
    frame.setVisible(true);
  }
}
```

Again, this works because the add method requires a Component as its input, and JButton is a (inherits) Component.

Hello swing

Let's add a `JLabel` and a `JButton`.

```java
import javax.swing.*;

public class Test {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Hello Swing");
    frame.setDefaultCloseOperation(
                          JFrame.DISPOSE_ON_CLOSE);
    frame.getContentPane().add(
                      new JLabel("Label name") );
    frame.getContentPane().add(
                      new JButton("Do nothing") );
    frame.setSize(300, 200);
    frame.setVisible(true);
  }
}
```

This doesn't work because the `JButton` is overlaid on top of the `JLabel`.

Hello swing

We should place Components in different locations.

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

public class Test {
  public static void main(String[] args) {
    ...
    frame.getContentPane().add(
                    new JLabel("Label name"),
                    BorderLayout.CENTER);
    frame.getContentPane().add(
                    new JButton("Do nothing"),
                    BorderLayout.SOUTH);
    ...
  }
}
```

We'll talk about `java.awt.BorderLayout` later.

Hello swing

Let's make the JButton do something.

```
import java.awt.event.*;
...
public class Test {
  public static void main(String[] args) {
    ...
    JButton button = new JButton("Click me");
    frame.getContentPane().add(button,
                    BorderLayout.SOUTH);
    button.addActionListener(new ActionClass());
    ...
  }
}
```

Hello swing

Let's make the `JButton` do something.

```
...
class ActionClass implements ActionListener {
  public void actionPerformed(ActionEvent event){
    System.out.println("Button clicked");
  }
}
```

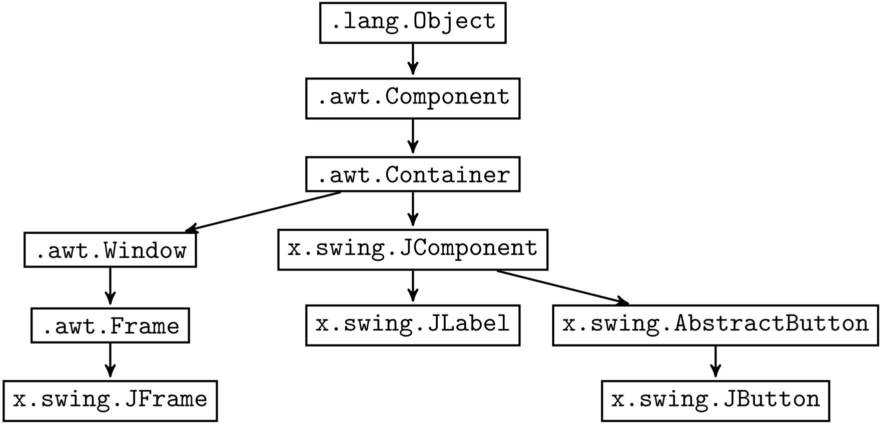
We'll talk about the interface `java.awt.event.ActionListener` later.

swing inheritance hierarchy

The `swing` and `awt` libraries are organized around inheritance.

Getting used to this inheritance hierarchy is key.

swing inheritance hierarchy



(. means `java.` and `x.` means `javax.`)

How to write a GUI

To build a graphical user interface (GUI), you need to

- create components,
- lay out the components, and
- make the components do useful things.

These lectures will proceed in this rough order.

Outline

Basic JComponents

Event listeners

Layout manager

JLabel

Use JLabel to display a text and/or an image. The text and image are unselectable.

```
public static void main(String[] args) {
  ...
  JLabel label = new JLabel("UCLA logo");
  Icon icon = new ImageIcon("ucla.jpg");
  label.setIcon(icon);
  label.setVerticalTextPosition(JLabel.BOTTOM);
  label.setHorizontalTextPosition(JLabel.CENTER);
  frame.getContentPane().add(
                label, BorderLayout.CENTER);
  ...
}
```

JLabel

setIcon takes in Icon, which ImageIcon implements.

```
  Icon icon = new ImageIcon("ucla.jpg");
  label.setIcon(icon);
```

These set the text position with respect to the icon

```
label.setVerticalTextPosition(JLabel.BOTTOM);
label.setHorizontalTextPosition(JLabel.CENTER);
```

JLabel inherits the static fields from interface SwingConstants.

JLabel

The constructors and “setter” methods tells us a lot about what we can do with the `class`. Use the “getter” methods to retrieve information.

Let’s look at

- the constructors,
- `setText`,
- `setVerticalAlignment`, and
- `setHorizontalAlignment`.

Icon

The iterface `javax.swing.Icon` represents a small fixed size picture, typically used to decorate components.

`ImageIcon` is the most (and probably the only) useful `class` that implements `Icon` within the Java API.

ImageIcon

`class javax.swing.ImageIcon` represents icons from images.

`ImageIcon`'s constructors support a few ways of loading an image.

- What is the "description"?
- What does `equals` do? It's inherited but not overridden.
- Can you add an `ImageIcon` directly to a content pane?

ImageIcon

`class javax.swing.ImageIcon` represents icons from images.

`ImageIcon`'s constructors support a few ways of loading an image.

- What is the "description"?
- What does `equals` do? It's inherited but not overridden.
- Can you add an `ImageIcon` directly to a content pane?
  No, because `ImageIcon` doesn't inherit from `Component`.

Custom Icon

There's only 1 useful `class` that implements `Icon`, but that doesn't mean the interface `Icon` is useless. We can create our own custon `Icon`s.

To implement `Icon` we must implement the (implicitly) `abstract` method

```
paintIcon(Component c, Graphics g, int x, int y)
```

We need to learn about `java.awt.Component` and `java.awt.Graphics`.

JPanel

`javax.swing.JPanel` is a general purpose lightweight container.

```
JPanel panel1 = new JPanel();
panel1.add(new JLabel("Label in panel1"));
frame.getContentPane().add(panel1,
                           BorderLayout.CENTER);

JPanel panel2 = new JPanel();
panel2.add(new JLabel("Label in panel2"));
frame.getContentPane().add(panel2,
                           BorderLayout.SOUTH);
```

JPanel

`JPanel` inherits `javax.swing.JComponent` and, in particular, inherits `setBackground`.

```
JPanel panel1 = new JPanel();
//pass in a Color object with RGB code
panel1.setBackground(new Color(50,132,191));

JPanel panel2 = new JPanel();
//Color.YELLOW is a constant of class Color
panel2.setBackground(Color.YELLOW);
```

We'll skip the discussion of `java.awt.Color`.

Nested JPanels

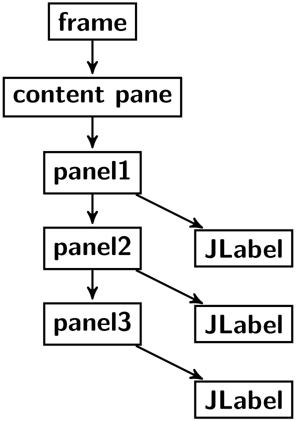
JPanels can be nested.

```
JPanel panel1 = new JPanel();
JPanel panel2 = new JPanel();
JPanel panel3 = new JPanel();

panel1.setBackground(Color.BLUE);
panel2.setBackground(Color.YELLOW);
panel3.setBackground(Color.RED);

panel1.add(new JLabel("Label in panel1"),
                                BorderLayout.SOUTH);
panel2.add(new JLabel("Label in panel2"),
                                BorderLayout.SOUTH);
panel3.add(new JLabel("Label in panel3"));

...
```

Nested JPanels

```
...

frame.getContentPane().add(panel1);
panel1.add(panel2, BorderLayout.CENTER);
panel2.add(panel3, BorderLayout.CENTER);
```

Nested JPanels



GUI's naturally have a nested and hierarchical structure. This makes inheritance and polymorphism the right tool for this job.

Custom JPanel

From `JComponent`, `JPanel` inherits `paintComponent`.

```
protected void paintComponent(Graphics g);
```

It's protected and not final. It's meant to be overriden.

Java calls `paintComponent` when it renders the GUI on the screen.

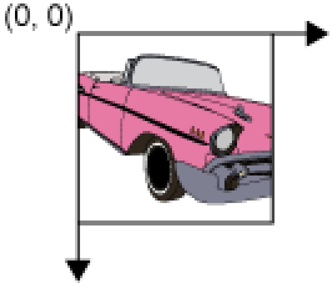
Custom JPanel

```java
class MyPanel extends JPanel {
  public void paintComponent(Graphics g) {
    g.setColor(Color.ORANGE);
    g.fillRect(20,50,100,100);

    g.setColor(Color.RED);
    g.drawOval(30,30,10,10);
  }
}
```

Graphics

Use `java.awt.Graphics` to draw onto components.

- Set the color and the properties of the draw, and then draw.
- "Draw" draws the border. "Fill" fills the interior of the shape.
- The x-coordinate goes from left to right. The y-coordinate goes from top to bottom.



Custom Icon

```java
public class MissingIcon implements Icon {
  private int w = 32, h = 32;

  @Override
  public void paintIcon(Component c, Graphics g,
                         int x, int y) {
    g.setColor(Color.BLACK);
    g.drawRect(x, y, w-1, h-1);
    g.setColor(Color.WHITE);
    g.fillRect(x+1, y+1, w-2, h-2);
    g.setColor(Color.RED);
    g.drawLine(x+10, y+10, x+w-10, y+h-10);
    g.drawLine(x+10, y+h-10, x+w-10, y+10);
  }
  ...
}
```

Graphics2D

java.awt.Graphics2D is a subclass of java.awt.Graphics with additional functionality.

```java
public class MissingIcon implements Icon {
  ...
  @Override
  public void paintIcon(Component c, Graphics g,
                        int x, int y) {
    ...
    g.setColor(Color.RED);
    ((Graphics2D) g).setStroke(
                           new BasicStroke(4));
    g.drawLine(x+10, y+10, x+w-10, y+h-10);
    g.drawLine(x+10, y+h-10, x+w-10, y+10);
  }
  ...
}
```

Graphics2D

Since Java 1.2, all `Graphics` objects provided by the Java API are actually `Graphics2D` objects.

`java.awt.Graphics` maintained for backward compatibility.

Using MissingIcon

Here's how you might use this custom `Icon`.

```
public class MissingIcon implements Icon {
  ...
  public static Icon iconFact(String dir) {
    ...
    if (fileExists)
      return new ImageIcon(dir);
    else
      return new MissingIcon();
  }
  ...
}
```

JTextArea

Use javax.swing.JTextArea to display multiple lines of text that the user can edit.

JTextArea inherits javax.swing.text.JTextComponent and is one of the several components that can hold text.

```
JTextArea textArea = new JTextArea();
textArea.setBackground(Color.YELLOW);
frame.getContentPane().add(textArea);
```

Outline

Basic JComponents

Event listeners

Layout manager

Events

Various GUI components fire *events*. By default these events are ignored, but you can choose to listen to them.

When you attach a `EventListener` to an event, the `EventListener` is run when the event is fired.

There are several types of events. (It's possible to create and fire custom events, but we won't.)

When a `Component` has the method

```
public void addXListener(XListener l)
```

you can tell that it fire an `XEvent`.

ActionEvent and ActionListener

Buttons and clickable components can fire `java.awt.event.ActionEvents`.
A `java.awt.event.ActionListener` listens to an `ActionEvent`.

The `interface ActionListener` has one method

```
public void actionPerformed(ActionEvent e)
```

which is called when the `ActionEvent` the `ActionListener` is listening to is fired.

ActionEvent and ActionListener

A `JButton` (and any `AbstractButton`) can fire an `ActionEvent`. Use

```
addActionListener(ActionListener l)
```

to attach an `ActionListener` to the `JButton`'s `ActionEvent`.

ActionEvent and ActionListener

Let's separate the `main` function from the GUI.

```
public class Test {
  public static void main(String[] args) {
    MyGUI gui = new MyGUI();
  }
}
...
```

ActionEvent and ActionListener

```java
...
class MyGUI implements ActionListener {
  public MyGUI() {
    ...
    JButton button = new JButton("Click me");
    button.addActionListener(this);
    ...
  }
  @Override
  public void actionPerformed(ActionEvent e) {
    System.out.println("Say something");
  }
}
```

ActionEvent and ActionListener

We can make `actionPerformed` access other GUI components.

```
...
class MyGUI implements ActionListener {
  private JTextArea text;
  public MyGUI() {
    ...
    JButton button = new JButton("Click me");
    button.addActionListener(this);
    text = new JTextArea();
    ...
  }
  @Override
  public void actionPerformed(ActionEvent e) {
    System.out.println(text.getText());
  }
}
```

Inner classes as EventListeners

If you think about it, it's weird that the `class MyGUI` is an `ActionListener`.

What listens to `Jbutton`'s `ActionEvent` should belong to the GUI. It should not be the GUI itself.

Inner classes as EventListeners

If you have 2 `JComponents` that fire `ActionEvents`, what do you do? Here's a not-so-nice ad-hoc solution.

```
class MyGUI implements ActionListener {
  private final JButton button1, button2;
  ...
  {
    ...
    button1 = new JButton("button1");
    button2 = new JButton("button2");
    button1.addActionListener(this);
    button2.addActionListener(this);
    ...
  }
  ...
```

Inner classes as EventListeners

If you have 2 JComponents that fire ActionEvents, what do you do? Here's a not-so-nice ad-hoc solution.

```
  ...
  @Override
  public void actionPerformed(ActionEvent e) {
    if (e.getSource() == button1)
      System.out.println("Button1 clicked");
    else if (e.getSource() == button2)
      System.out.println("Button2 clicked");
  }
}
```

Using ActionEvent is fine. That actionPerformed becomes complicated with many buttons and components is the problem.

Inner classes as EventListeners

It's better to use inner `classes` as `EventListeners`.

```
class MyGUI {
  private JTextArea text;
  private final JButton button1, button2;

  private class B1L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      System.out.println("button1 clicked");
    }
  }
  private class B2L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      System.out.println(text.getText());
    }
  }
  ...
```

Inner classes as EventListeners

Use inner classes

```
  ...
  {
    ...
    button1 = new JButton("button1");
    button2 = new JButton("button2");
    button1.addActionListener(new B1L());
    button2.addActionListener(new B2L());
    ...
  }
}
```

Inner classes as EventListeners

Note that if B2L were a top-level class, it couldn't access MyGUI's private member text.

```
class MyGUI {
  private JTextArea text;
  private final JButton button1, button2;

  private class B2L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      //only inner classes can do this
      System.out.println(text.getText());
    }
  }
  ...
```

You want to use inner classes as EventListeners, so that they have access to the GUI components.

Mouse Listener

A `java.awt.Component` can fire a `java.awt.event.MouseEvent`.
A `java.awt.event.MouseListener` listens to a `MouseEvent`.

```
class MyGUI {
  {
    JPanel panel = new JPanel();
    panel.addMouseListener(new ML());
    ...
  }
  ...
```

Mouse Listener

```
...
private class ML implements MouseListener {
  public void mouseClicked(MouseEvent e) {
    System.out.println("clicked");
    String xy = "("+e.getX()+","+e.getY()+")";
    System.out.println(xy);
  }
  ...
```

Mouse Listener

```
    public void mouseEntered(MouseEvent e) {
      System.out.println("entered");
    }
    public void mouseExited(MouseEvent e) {
      System.out.println("exited");
    }
    public void mousePressed(MouseEvent e) {
      System.out.println("pressed");
    }
    public void mouseReleased(MouseEvent e) {
      System.out.println("released");
    }
  }
}
```

Mouse Listener

Even if you don't use all 5 `MouseEvents`, you must provide an implementation of the (implicitly `abstract`) methods.

```
private class ML implements MouseListener {
  public void mouseClicked(MouseEvent e) {
    ...
  }
  public void mouseEntered(MouseEvent e) { }
  public void mouseExited(MouseEvent e) { }
  public void mousePressed(MouseEvent e) { }
  public void mouseReleased(MouseEvent e) { }
}
```

For the ones you don't use, a blank implementation is just fine.

setDefaultCloseOperation

When you close a JFrame, a WindowEvent is fired and, by default, the JFrame is hidden.

When a java.awt.Window (a superclass of JFrame) exists, Java does not exit, even if the end of main is reached.

You can add a WindowListener to the JFrame and have it dispose the JFrame when it's closed. You can do the same with

```
frame.setDefaultCloseOperation(
                               JFrame.DISPOSE_ON_CLOSE);
```

Once all JFrames are disposed, Java can exit the program.

setDefaultCloseOperation

You can forcefully exit the program upon closing the JFrame.

```
frame.setDefaultCloseOperation(
                              JFrame.EXIT_ON_CLOSE);
```

DISPOSE_ON_CLOSE and EXIT_ON_CLOSE are actually different when you have 2 or more JFrames.

Outline

Basic JComponents

Event listeners

Layout manager

Layout manager

`JPanels` use layout managers to lay out its components.

By default, `JPanels` use `FlowLayout`.
By default, content panes use `BorderLayout`.

Setting the layout manager

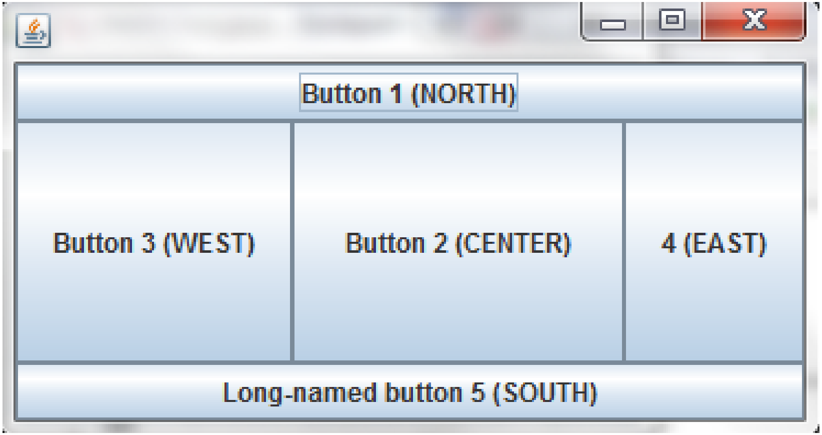
Set a `JPanel`'s layout manager in its constructor.

```
JPanel panel = new JPanel(new BorderLayout());
```

You can later set a `Container`'s layout manager using `setLayout`.

```
Container contentPane = frame.getContentPane();
contentPane.setLayout(new FlowLayout());
```

BorderLayout

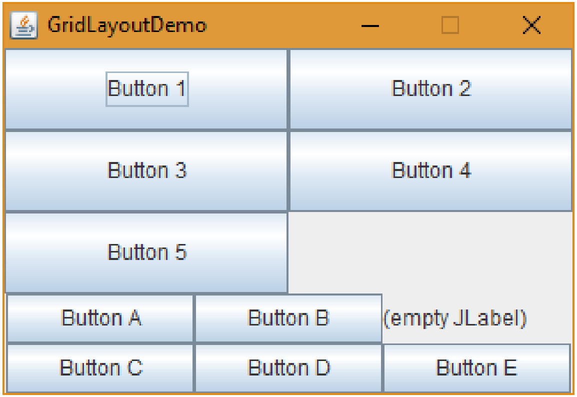


BorderLayout

`CENTER` gets as much of the available space as possible.
Other areas get the minimum space to fit their `Components`.

When no position is specified, `add` places a `Component` in `CENTER`.

GridLayout


GridLayoutDemo
Button 1
Button 2
Button 3
Button 4
Button 5
Button A
Button B
(empty JLabel)
Button C
Button D
Button E

GridLayout

`Components` are placed in a grid of cells. The cells have same size.

Specify the number of rows and columns of the grid in `GridLayout`'s constructor.

`0` rows means the number of rows is unspecified.
`0` columns means the number of columns is unspecified.

GridLayout

```
Container pane = frame.getContentPane();
JPanel panel1 = new JPanel();
JPanel panel2 = new JPanel();

pane.add(panel1, BorderLayout.NORTH);
pane.add(panel2, BorderLayout.SOUTH);
```

GridLayout

```
panel1.setLayout(new GridLayout(0,2));
panel1.add(new JButton("Button 1"));
panel1.add(new JButton("Button 2"));
panel1.add(new JButton("Button 3"));
panel1.add(new JButton("Button 4"));
panel1.add(new JButton("Button 5"));

panel2.setLayout(new GridLayout(2,3));
panel2.add(new JButton("Button A"));
panel2.add(new JButton("Button B"));
panel2.add(new JLabel("(empty JLabel)"));
panel2.add(new JButton("Button C"));
panel2.add(new JButton("Button D"));
panel2.add(new JButton("Button E"));
```

Putting space between Components

Here are 3 simple ways to insert empty space in your GUI:
layout manager, invisible components, and empty borders.

- Some layout managers give you some control over empty space.
- Empty `JPanels` or `JLabels` can take up space.
- You can add invisible borders around `JPanels` and `JLabels`.

Borders

JPanels and JLabels can have Borders.

```
JPanel panel1 = new JPanel();
panel1.setBorder(
   BorderFactory.createLineBorder(Color.BLACK));
JPanel panel2 = new JPanel();
panel2.setBorder(
   BorderFactory.createEmptyBorder(3,3,3,3));
```

Use the factory class `javax.swing.BorderFactory` to create a `javax.swing.border.Border` object.

Absolute positioning

You can manually specify the size and position of `Components` with absolute positioning. This is usually a bad idea.

A GUI using absolute positioning does not adjust well to resizing and to differences between systems.