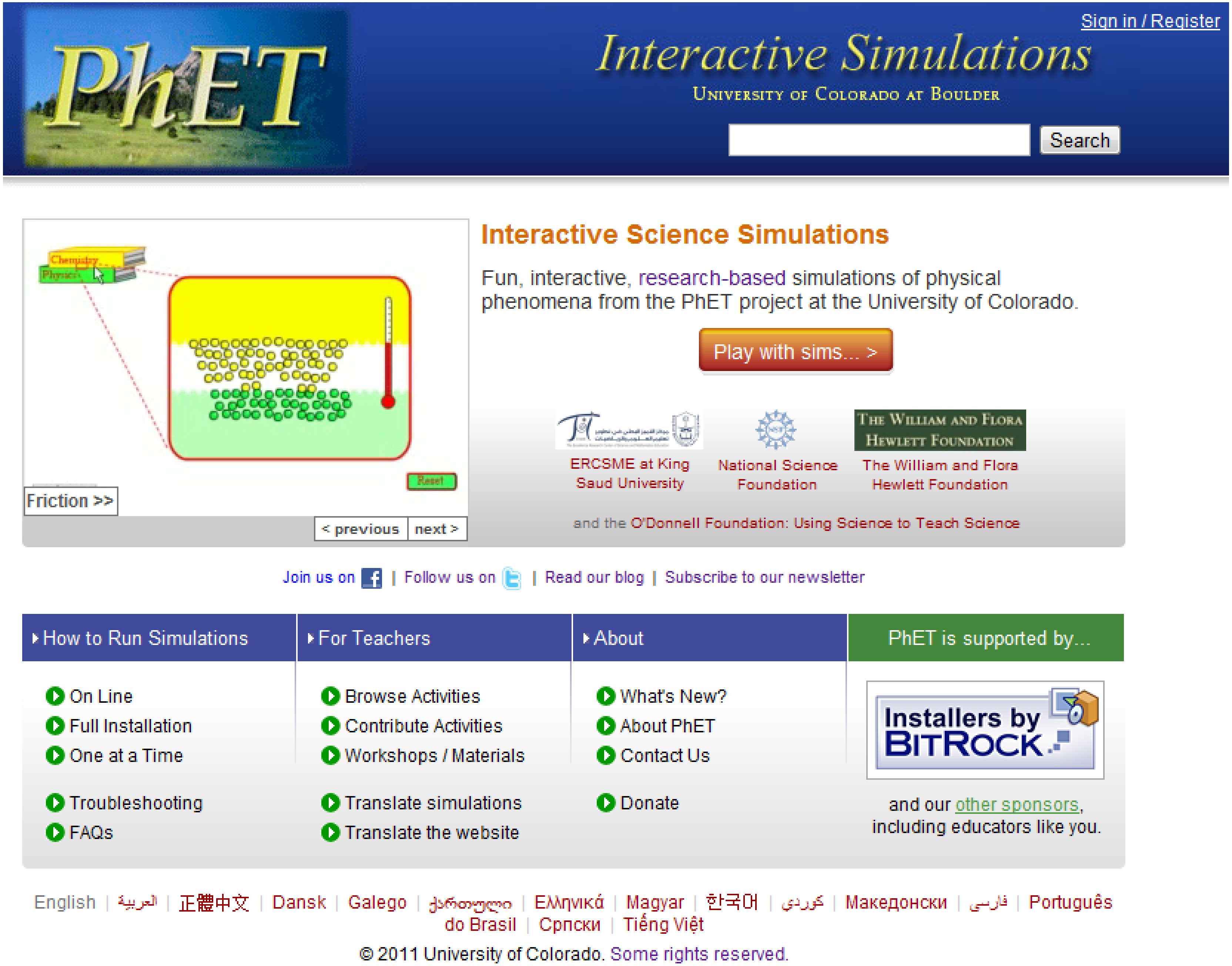
How Scala Improved Our Java

Sam Reid
PhET Interactive Simulations
University of Colorado

http://spot.colorado.edu/~reids/

PhET Interactive Simulations

- Provides free, open source educational science simulations at http://phet.colorado.edu/
- Used in middle school, high school, college
- Translated into 58 languages
- Launched over 2 million times per month, and growing
- Simulations written in Java, Flash, Flex & Scala



Interactive Simulations

University of Colorado at Boulder

Interactive Science Simulations

Fun, interactive, research-based simulations of physical phenomena from the PhET project at the University of Colorado.

Play with sims... >

ERCSME at King Saud University | National Science Foundation | The William and Flora Hewlett Foundation

and the O'Donnell Foundation: Using Science to Teach Science

Join us on | Follow us on | Read our blog | Subscribe to our newsletter

How to Run Simulations

- On Line
- Full Installation
- One at a Time
- Troubleshooting
- FAQs

For Teachers

- Browse Activities
- Contribute Activities
- Workshops / Materials
- Translate simulations
- Translate the website

About

- What's New?
- About PhET
- Contact Us
- Donate

PhET is supported by...

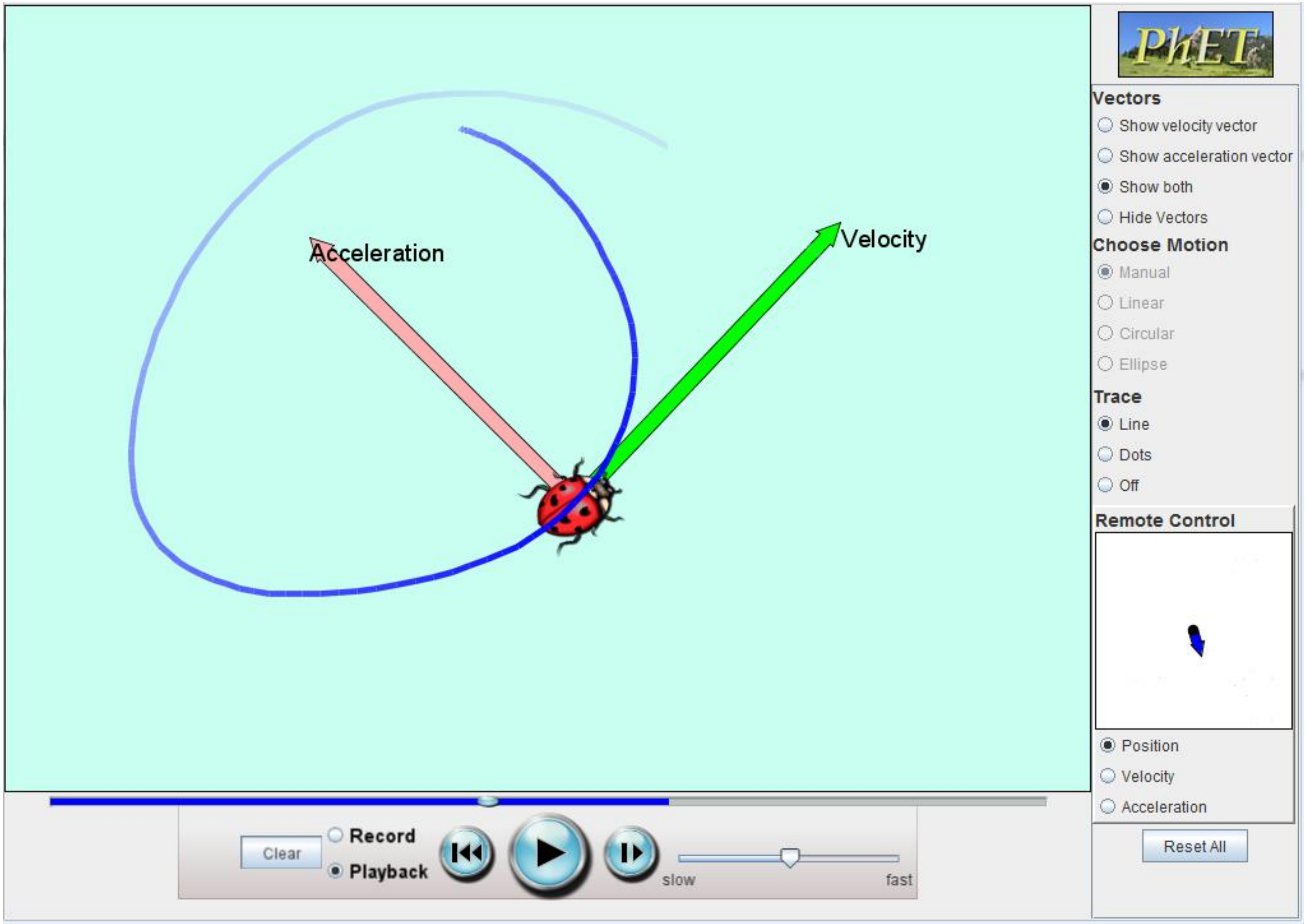
Installers by BITROCK

and our other sponsors, including educators like you.

English | العربية | 正體中文 | Dansk | Galego | ქართული | Ελληνικά | Magyar | 한국어 | کوردی | Македонски | فارسی | Português do Brasil | Српски | Tiếng Việt

PhET’s Scala Simulations

  - Gravity Force Lab
  - Ladybug Motion 2D
  - Forces and Motion
  - Ramp: Forces and Motion
- Launched over 50,000 times in the last month
- Translated into over 25 languages



Vectors
○ Show velocity vector
○ Show acceleration vector
● Show both
○ Hide Vectors

Choose Motion
● Manual
○ Linear
○ Circular
○ Ellipse

Trace
● Line
○ Dots
○ Off

Remote Control
● Position
○ Velocity
○ Acceleration

Reset All

Clear
○ Record
● Playback
slow
fast

Advantages of Scala over Java

- Function literals
  - useful in GUI callbacks and observer pattern
- Operator infix notation
  - useful in 2D vector arithmetic
- Case classes and immutability
  - simplified record/playback
- For comprehensions
- If/else more readable than Java’s ternary
- Many more!

Better Java

1. Improved object initialization
2. Improved management of boilerplate
3. Reduced code duplication
4. Improved code organization

1. Idiomatic Java for Object Initialization

```
JButton button = new JButton("Click Me");
JLabel label = new JLabel("No clicks yet");

JPanel contentPane = new JPanel();
contentPane.add( button );
contentPane.add( label );

JFrame frame = new JFrame("Reactive Swing App");
frame.setContentPane( contentPane );
```

Idiomatic Scala

```
def top = new MainFrame {
   title = "Reactive Swing App"
   val button = new Button {
      text = "Click me“
   }
   val label = new Label {
      text = "No clicks yet“
   }
   contents = new BoxPanel(Orientation.Vertical) {
     contents += button
     contents += label
   }
}
```

Idiomatic Scala + Inline Declarations

```
def top = new MainFrame {
   title = "Reactive Swing App"
   contents = new BoxPanel(Orientation.Vertical) {
     contents += new Button {text = "Click me“}
     contents += new Label {text = "No clicks yet}
   }
}
```

Advantages of Scala initialization

- Structural instead of procedural
  - More natural: ”a frame with a button”
  - Intended scope clear
  - Top down instead of bottom-up
- Object correct upon instantiation, instead of mutating through a series of incorrect states
- No need to build a mental model of many-to-many map; code is a tree like the object graph

Double Brace Initialization

```
JFrame frame = new JFrame("Reactive Swing App") {{
  setContentPane( new JPanel() {{
    add(new JButton( "Click Me" ) );
    add(new JLabel( "No clicks yet" ) );
  }} );
}};
```

DBI Issues

- Like Scala, creates an anonymous subclass
  - About 454 bytes each
  - 100 DBI's increases a 2MB JAR by about 2%
- Need to be cautious about equals, hashcode, serialization
- Unfamiliar style to idiomatic Java programmers?

When to use anonymous subclass?

- Describing object state in an object tree
- Constructor is insufficient

When not to use?

- Too many nested levels can be confusing
  - Especially when nesting objects of similar type
- No name for intermediate ‘this’

Better Java

1. Improved object initialization
2. **Improved management of boilerplate**
3. Reduced code duplication
4. Improved code organization

2. Closure Folding: Before and After

```
//Idiomatic Java
model.addListener( new Listener() {
   public void update() {
       trace( "hello" );
   }
});

//Idiomatic Scala
m.addListener( ()=>{trace("hello"})

//Java code with closure folding
m.addListener(Listener(){trace( "hello" );}});
```

Benefits of Closure Folding

- Easier to read and understand
- Helps you focus on the tricky parts
- See more code at once
  - In production code, 49 lines of code reduced to 9 in one file
- Closure folding accentuates advantages in
  - Listener callbacks
  - Inner functions

Better Java

1. Improved object initialization
2. Improved management of boilerplate
3. **Reduced code duplication**
4. Improved code organization

3. Listeners in Java

```
public class Player {
    private String name = …;//set in constructor
    private List<Listener> listeners = new ArrayList<Listener>();

    public void addListener( Listener listener ) {
        listeners.add( listener );
    }

    public void setName( String name ) {
        this.name = name;
        //Copy since list may be modified during traversal
        for ( Listener listener : new ArrayList(listeners) )
           listener.nameChanged( this );
    }

    public static interface Listener {
        void nameChanged( Player player );
        void ageChanged( Player player );
    }
}
```

Listener Client Code

```
public class ListenerClient {
  public static void main( String[] args ) {
    Player player = new Player(“Larry”);

    final JTextField tf = new JTextField( “Name is: " +
  player.getName() );

    player.addListener( new Listener() {
      public void nameChanged( Player player ) {
        tf.setText( "The name is: " + player.getName() );
      }

      public void ageChanged( Player player ) {}
    } );
  }
}
```

Problems with Java Idiom

- Duplicated code to implement observers
- Duplicated code in synchronizing a client with the observable
  - Potential mismatch between initial and subsequent values
- Duplicated code in wiring up to GUI controls

Control Structure in Scala

```
var text: String = null

val player = new Player("Larry")
invokeAndPass(player.addListener) {
  tf.setText( "The name is: " + player.getName() )
}
player.name = "Steve"

//Result
//The name is: Larry
//The name is: Steve
```

Reusable Check Box in Scala

```
class FunctionCheckBox(
   text: String,
   actionListener: Boolean => Unit,
   getter: () => Boolean,
   addListener: ( () => Unit ) => Unit
)  extends CheckBox

//How to encapsulate all these parts?
```

Property<T> and Binding

```
name = new Property<String>( “Larry” );
name.addObserver( n{ tf.setText(“The name is: “ + n );}});
name.set( “Steve” );

//Displayed in the text field:
// The name is: Larry
// The name is: Steve
```

- See also:
  - Maier’s “Signal[+A]”
    - *Deprecating the Observer Pattern, EPFL Report: Maier, Rompf, Odersky*
    - *Scala.React*
  - JavaFX Java API for properties and binding

ObservableProperty

```
class ObservableProperty<T>{
  List<VoidFunction1<T>> observers = …

  abstract T get();

  //Adds an observer that will be automatically
  notified with the current value, and when the
  value changes
  addObserver(VoidFunction1<T> observer){
      observers.add(observer);
      observer.apply(get());
  }
}
```

Property Composition DSL

```
public class DividedBy extends CompositeDoubleProperty{
 public DividedBy( final ObservableProperty<Double> num,
                   final ObservableProperty<Double> den){
  super(Function0(){return num.get()/den.get();},num,den);
 }
}

//Compute concentration of salt = moles/volume
ObservableProperty<Double> saltConcentration=
 salt.molesDissolved.dividedBy(solution.volume);

//Show “remove” button if any salt or sugar
ObservableProperty<Boolean> anySolutes =
 salt.greaterThan( 0 ).or(sugar.greaterThan( 0 ));
```

PropertyCheckBox in Java

```
public PropertyCheckBox( String text,
                         final SettableProperty<Boolean> p) {
 super( text );

 // update the model when the check box changes
 addChangeListener(ChangeListener(e) {p.set( isSelected() );}} );

 // update the check box when the model changes
 p.addObserver(SimpleObserver() {setSelected( p.get() );});
}

//Sample usage
add(new PropertyCheckBox(“gravity on”, gravityEnabled));
add(new PropertyCheckBox(“gravity off”, not(gravityEnabled));
```

Pros and Cons of Property<T>

- Pro
  - Uniform API for dealing with mutable state
  - Good for binding/synchronization
  - Useful in removing duplicated code
  - Blocks against redundant messages
- Con
  - Impedance mismatch when interfacing with legacy code
  - Sometimes harder to debug than primitives
  - Most suitable for 1-way flow of information

Better Java

1. Improved object initialization
2. Improved management of boilerplate
3. Reduced code duplication
4. **Improved code organization**

4. Private Methods in Java

```
public PrismGraphic(){
    //Initialization code
    color = createColor(wavelength);
    neighborColor = createColor(wavelength+10);
    //Many more lines
    //of complex initialization code
    //…
}

//Other methods…
private Color createColor(double wavelength){
    //code to return the Color for a wavelength
}
```

Vs. Inner Functions in Scala

```
public PrismGraphic(){
      def createColor = (d:Double)=>{…}
      color = createColor(wavelength);
      neighborColor=createColor (wavelength+10);
}
```

Inner Functions

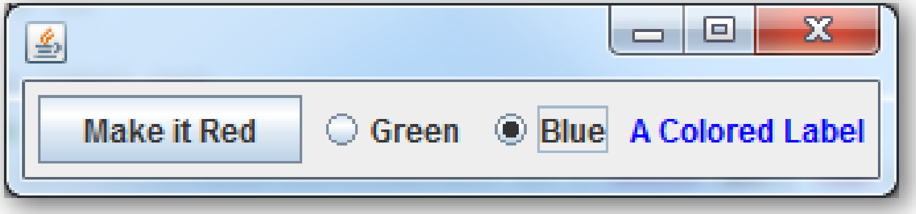
- Pro
  - Local scope makes it obvious to maintainer where it is supposed to be used
  - Reduces clutter in class namespace
- Con
  - Sometimes good to move implementation away from usage point to improve readability

Inner Function in Java

```
public PrismGraphic(){
 Function1<Double,Color> createColor =
  Function1(Double wavelength) { return … };

  color = createColor.apply(wavelength);
  neighborColor=createColor.apply(wavelength+10);
}
```

Larger Example/Idiomatic Java



```
public class TestWithoutDBI {
    private Color color = Color.red;
    private ArrayList<ColorChangeListener> listeners = new …

    public interface ColorChangeListener {
        void colorChanged( Color color );
    }

    public void addColorChangeListener( ColorChangeListener cl ) {
        listeners.add( cl );
    }

    private void start() {
        JFrame frame = new JFrame();
        final JPanel contentPane = new JPanel();
```

```
//Button that makes the label red
final JButton redButton = new JButton( "Make it Red" );
redButton.addActionListener( new ActionListener() {
   public void actionPerformed( ActionEvent e ) {
        setColor( Color.red );
   }
} );
contentPane.add( redButton );

//Radio boxes to change the color
final JRadioButton greenRadioButton = new JRadioButton( "Green" );
greenRadioButton.addActionListener( new ActionListener() {
   public void actionPerformed( ActionEvent e ) {
        setColor( Color.green );
   }
} );
contentPane.add( greenRadioButton );

final JRadioButton blueRadioButton = new JRadioButton( "Blue" );
```

```
blueRadioButton.addActionListener( new ActionListener() {
   public void actionPerformed( ActionEvent e ) {
        setColor( Color.blue );
   }
} );
contentPane.add( blueRadioButton );

ButtonGroup buttonGroup = new ButtonGroup();
buttonGroup.add( greenRadioButton );
buttonGroup.add( blueRadioButton );
buttonGroup.add( redButton );

//A label whose color is set based on the property
final JLabel label = new JLabel( "A Colored Label" );
addColorChangeListener( new ColorChangeListener() {
   public void colorChanged( Color color ) {
        label.setForeground( color );
   }
} );
label.setForeground( color );
```

```
   contentPane.add( label );
   frame.setContentPane( contentPane );
   frame.pack();
   frame.setVisible( true );
}

private void setColor( Color color ) {
   boolean changed = !this.color.equals( color );
   this.color = color;
   if ( changed ) {
        for ( ColorChangeListener listener : listeners ) {
                listener.colorChanged( color );
            }
   }
}
```

Same Example: Folding/DBI/Properties

```
new JFrame() {{
    final Property<Color> color = new Property<Color>( red );

    //Set the content pane, which contains controls for changing the red
        property and a label that is shown in the selected color
    setContentPane( new JPanel() {{

        //Button that makes the label red
        add( new JButton( "Make it Red" ) {{
            addActionListener( event { color.set( red );}} );
        }} );

        //Radio boxes to change the color
        add( new PropertyRadioButton<Color>( “Green", color, green ) );
        add( new PropertyRadioButton<Color>( “Blue", color, blue ) );

        //A label whose color is set based on the property
        add( new JLabel( "A Colored Label" ) {{
            color.addObserver( color { setForeground( color );}} );
        }} );
    }} );
    pack();
}}.setVisible( true );
```

Conclusion

- Goal: code easier to write/read/maintain
- Improved object initialization
  - Double-brace pattern
- Improved management of boilerplate
  - Closure folding
- Reduced code duplication
  - Variable binding
- Improved code organization
  - Local function objects

Questions?