An Introduction to Reactive Programming

Guido Salvaneschi
Software Technology Group

Outline

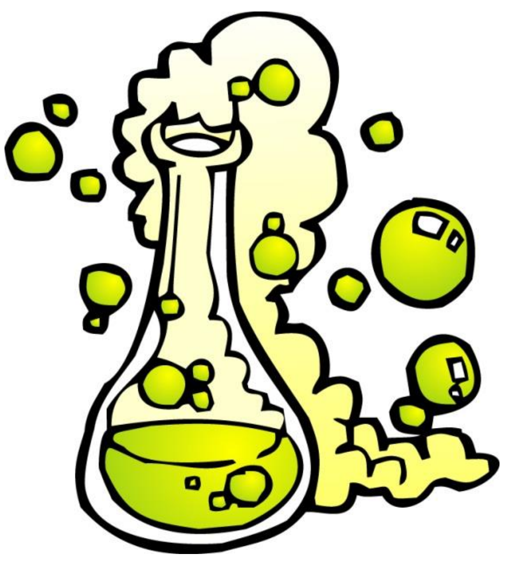
- Intro to reactive applications
- The Observer pattern
- Event-based languages
- Reactive languages

INTRO TO REACTIVE APPLICATIONS

Reactive Applications

- External/internal events trigger a reaction
  - User input
  - Network packet
  - Interrupt
  - Data from sensors

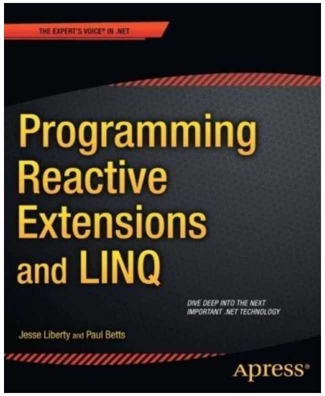
- Classic example:
  - Data change in MVC



Getting Widespread…

- Reactive programming in JavaScript
  - Bacon.js, Reactive.js, React.js, …

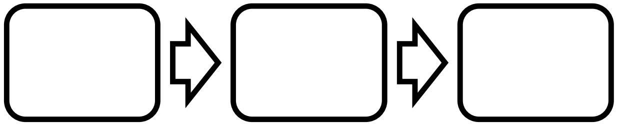
- Microsoft reactive extensions (Rx)



- Principles of Reactive Programming

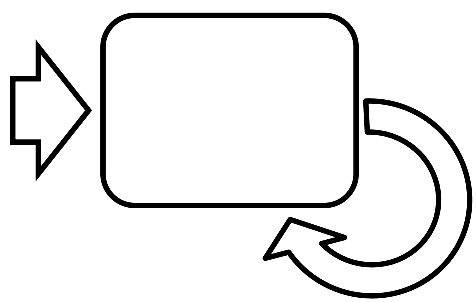
Software Taxonomy

- A transformational system:
  - Accepts some input, performs computation on it, produces output, and then terminates.
  - Independent of time, ideally instantaneous
  - Compilers, shell tools, scientific/engineering computations



Software Taxonomy

- A reactive system:
  - Continuously interacts with its environment.
  - Changing in time, reflects the environment
  - Editors, Web applications, embedded software, simulations



Reactive Programming

Now…

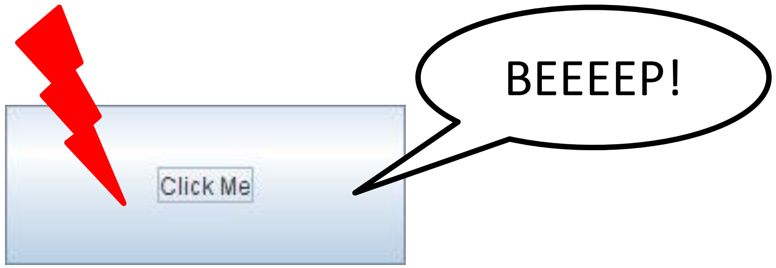
- The problem is extremely common
- Can we design new language features to specifically address this issue ?

- Think about exceptions, visibility modifiers, inheritance, …

THE OBSERVER PATTERN

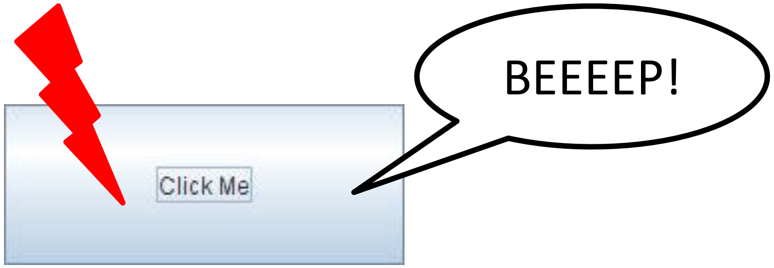
The Observer Pattern

- What about Java Swing ?
  - javax.swing

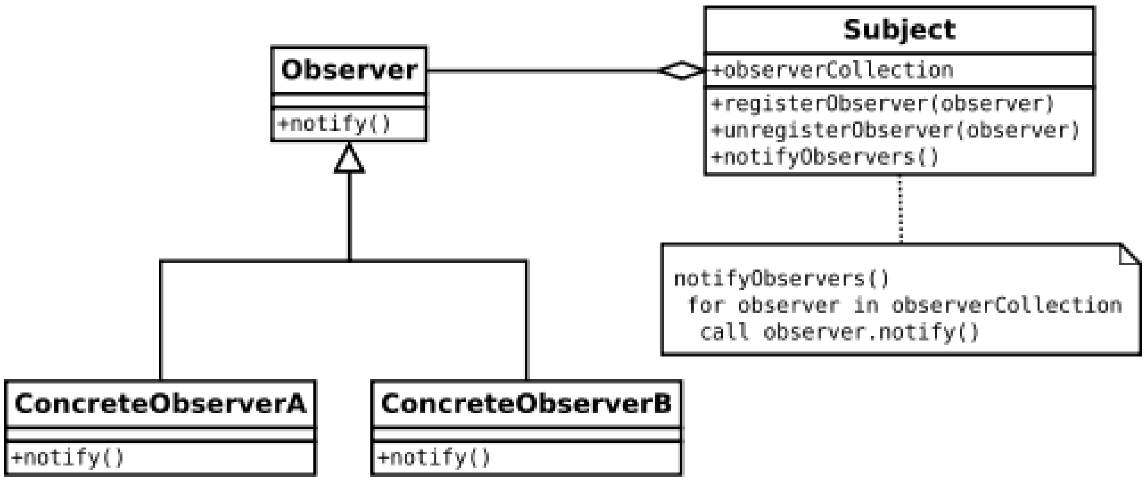


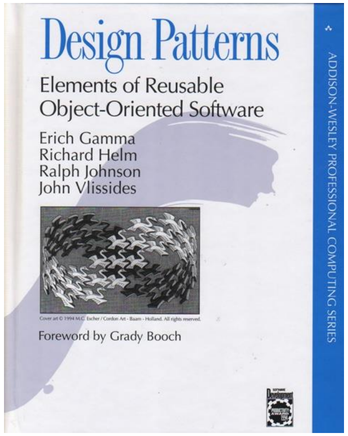
```java
public class Beeper extends JPanel  implements ActionListener {
  JButton button;

  public Beeper() {
    super(new BorderLayout());
    button = new JButton("Click Me");
    button.setPreferredSize(new Dimension(200, 80));
    add(button, BorderLayout.CENTER);
    button.addActionListener(this);
  }
  public void actionPerformed(ActionEvent e) {
    Toolkit.getDefaultToolkit().beep();
  }
  private static void createAndShowGUI() { // Create the GUI and show it.
    JFrame frame = new JFrame("Beeper");        //Create and set up the window.
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JComponent newContentPane = new Beeper();  //Create and set up the content pane.
    newContentPane.setOpaque(true);
    frame.setContentPane(newContentPane);
    frame.pack();        //Display the window.
    frame.setVisible(true);
  }
  public static void main(String[] args) {
    javax.swing.SwingUtilities.invokeLater( new Runnable() { public void run() {createAndShowGUI();}});
  }
}
```



The (*good*? old) Observer Pattern





EVENT-BASED LANGUAGES

Event-based Languages

- Events as objects attributes
  - Describe changes of object's state
  - Part of the interface

- Event-based languages are *better*!
  - Language-level support for events
  - C#, Ptolemy, EScala, EventJava, …

Example in C#

```
public class Drawing {
   Collection<Figure> figures;
   public event NoArgs Changed();
   public virtual void Add(Figure figure) {
      figures.Add(figure);
      figure.Changed += OnChanged;
      OnChanged();
   }
   public virtual void Remove(Figure figure) {
      figures.Remove(figure);
      figure.Changed -= OnChanged;
      OnChanged();
   }
   protected virtual void OnChanged() {
      if (Changed != null) { Changed(); }
   }
  ...
}
```

EVENTS IN SCALA

REScala

- Supports:
  - An advanced event-based system
  - Abstractions for time-changing values
  - Bridging between them

- **Philosophy**: foster a more declarative and functional style without sacrificing the power of OO design
- Pure Scala

Adding Events to Scala

- C# events are recognized by the compiler
  - Scala does not support events by itself, but…

- Can we introduce events using the powerful Scala support for DSLs?
- Can we do even better than C# ?
  - E.g. event composition ?

REScala events: Summary

- Different types of events: Imperative, declarative, …

- Events carry a value
  - Bound to the event when the event is fired
  - Received by all the handlers

- Events are parametric types.
  - Event[T], ImperativeEvent[T]

- All events are subtype of Event[T]

Imperative Events

- Valid event declarations

```
val e1 = new ImperativeEvent[Unit]()
val e2 = new ImperativeEvent[Int]()
val e3 = new ImperativeEvent[String]()
val e4 = new ImperativeEvent[Boolean]()
val e5: ImperativeEvent[Int] = new ImperativeEvent[Int]()
class Foo
val e6 = new ImperativeEvent[Foo]()
```

Imperative Events

- Multiple values for the same event are expressed by tuples

```
val e1 = new ImperativeEvent[(Int,Int)]()
val e2 = new ImperativeEvent[(String,String)]()
val e3 = new ImperativeEvent[(String,Int)]()
val e4 = new ImperativeEvent[(Boolean,String,Int)]()
val e5: ImperativeEvent[(Int,Int)] = new ImperativeEvent[(Int,Int)]()
```

Handlers

- Handlers are executed when the event is fired
  - The += operator registers the handler.
- The handler is a first class function
  - The attached value is the function parameter.

```
var state = 0
val e = new ImperativeEvent[Int]()
e += { println(_) }
e += (x => println(x))
e += ((x: Int) => println(x))
e += (x => {  // Multiple statements in the handler
  state = x
  println(x)
})
```

Handlers

- The signature of the handler must conform the event
  - E.g., Event[(Int,Int)] requires (Int,Int) =>Unit
  - The handler receives the attached value and performs side effects.

```
val e = new ImperativeEvent[(Int,String)]()
e += (x => {
  println(x._1)
  println(x._2)
})
e += (x: (Int,String) => {
  println(x)
})
```

Handlers

- Events without arguments still need a Unit argument in the handler.

```
val e = new ImperativeEvent[Int]()
e += { x => println() }
e += { (x: Unit) => println() }
```

Methods as Handlers

- Methods can be used as handlers.
  - Partially applied function syntax

```
def m1(x: Int) = {
  val y = x + 1
  println(y)
}
val e = new ImperativeEvent[Int]
e += m1 _
e(10)
```

Firing Events

- Method call syntax
- The value is bound to the event occurrence

```
val e1 = new ImperativeEvent[Int]()
val e2 = new ImperativeEvent[Boolean]()
val e3 = new ImperativeEvent[(Int,String)]()

e1(10)
e2(false)
e3((10,"Hallo"))
```

Firing Events

- Registered handlers are executed every time the event is fired.
  - The actual parameter is provided to the handler

```
val e = new ImperativeEvent[Int]()
e += { x => println(x) }
e(10)
e(10)
-- output ----
10
10
```

Firing Events

- All registered handlers are executed
  - The execution order is non deterministic

```
val e = new ImperativeEvent[Int]()
e += { x => println(x) }
e += { x => println("n: " + x)}
e(10)
e(10)
-- output ----
10
n: 10
10
n: 10
```

Firing Events

- The -= operator unregisters a handler

```
val e = new ImperativeEvent[Int]()
val handler1 = { x: Int => println(x)
val handler2 = { x: Int => println("n: " + x) }

e += handler1
e += handler2
e(10)
e -= handler2
e(10)
e -= handler1
e(10)

-- output ----
10
n: 10
10
```

Imperative Events

Simple but important…

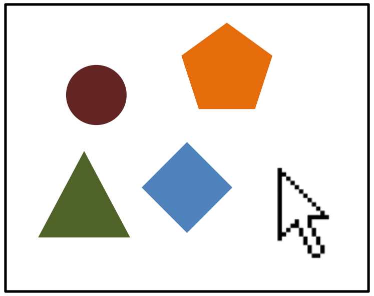
- Events can be referred to generically

```
val e1: Event[Int] = new ImperativeEvent[Int]()
```

DECLARATIVE EVENTS

The Problem

- Imperative events are fired by the programmer
- Conceptually, certain events depend on other events



- Examples:
  - mouseClickE -> museClickOnShape
  - mouseClose, keyboardClose -> closeWindow
- Can we solve this problem enhancing the language?

Declarative Events

- Declarative events are defined by a combination of other events.

- Some valid declarations:

```
val e1 = new ImperativeEvent[Int]()
val e2 = new ImperativeEvent[Int]()

val e3 = e1 || e2
val e4 = e1 && ((x: Int)=> x>10)
val e5 = e1 map ((x: Int)=> x.toString)
```

OR events

- The event e1 || e2 is fired upon the occurrence of one among e1 or e2.
  - The events in the event expression have the same parameter type

```
val e1 = new ImperativeEvent[Int]()
val e2 = new ImperativeEvent[Int]()
val e1_OR_e2 = e1 || e2
e1_OR_e2 += ((x: Int) => println(x))
e1(10)
e2(10)
-- output ----
10
10
```

Predicate Events

- The event e && p is fired if e occurs and the predicate p is satisfied.
  - The predicate is a function that accepts the event parameter as a formal and returns Boolean.
  - && filters events using a parameter and a predicate.

```
val e = new ImperativeEvent[Int]()
val e_AND: Event[Int] = e && ((x: Int) => x>10)
e_AND += ((x: Int) => println(x))
e(5)
e(15)
-- output ----
15
```

Map Events

- The event e map f is obtained by applying f to the value carried by e.
  - The map function takes the event parameter as a formal.
  - The return type of map is the type parameter of the resulting event.

```
val e = new ImperativeEvent[Int]()
val e_MAP: Event[String] = e map ((x: Int) => x.toString)
e_MAP += ((x: String) => println("Here: " + x))
e(5)
e(15)
-- output ----
Here: 5
Here: 15
```

DropParam

- The dropParam operator transforms an event into an event with Unit parameter.
  - E.g.: Event[Int] into Event[Unit]

```
val e = new ImperativeEvent[Int]()
val e_drop: Event[Unit] = e.dropParam
e_drop += (_ => println("*"))
e(10)
e(10)
-- output ----
*
*
```

DropParam

- Typical use case for the dropParam. Make events with different types compatible.

WRONG!

```
val e1 = new ImperativeEvent[Int]()
val e2 = new ImperativeEvent[Unit]()
val e1_OR_e2 = e1 || e2  // Compiler error
```

```
val e1 = new ImperativeEvent[Int]()
val e2 = new ImperativeEvent[Unit]()
val e1_OR_e2: Event[Unit] = e1.dropParam || e2
```

EXAMPLES OF RESCALA EVENTS

Example: Temperature Sensor

```
class TemperatureSensor {
  imperative evt tempChanged[Int]
 ...
  def run {
    var currentTemp = measureTemp()
    while(!stop) {
      val newTemp = measureTemp()
      if (newTemp != currentTemp) {
        tempChanged(newTemp)
        currentTemp = newTemp
      }
      sleep(100)
    }
  }
}
```

Example: Figures

```
abstract class Figure {
   val moved[Unit] = afterExecMoveBy
   val resized[Unit]
   val changed[Unit] = resized || moved || afterExecSetColor
   val invalidated[Rectangle] = changed.map( _ => getBounds() )
  ...
   val afterExecMoveBy = new ImpertiveEvent[Unit]
   val afterExecSetColor = new ImpertiveEvent[Unit]
  ...
   def moveBy(dx: Int, dy: Int) { position.move(dx, dy); afterExecMoveBy() }
   def resize(s: Size) { size = s }
   def setColor(col: Color) { color = col; afterExecSetColor() }
   def getBounds(): Rectangle
...
}
```

Example: Figures

```
class Connector(val start: Figure, val end: Figure) {
    start.changed += updateStart _
    end.changed += updateEnd _
  ...
    def updateStart() { ... }
    def updateEnd() { ... }
  ...
    def dispose {
        start.changed -= updateStart _
        end.changed -= updateEnd _
    }
}
```

Example: Figures

- Inherited events
  – May be overridden
  – Are late bound

```
abstract class Figure {
   val moved[Unit] = afterExecMoveBy
   val resized[Unit]
  ...
}
```

```
class RectangleFigure extends Figure {
   val resized[Unit] = afterExecResize || afterExecSetBounds
   override val moved[Unit] = super.moved || afterExecSetBounds)
  ...
   val afterExecResize = new ImpertiveEvent[Unit]
   val afterExecSetBounds = new ImpertiveEvent[Unit]
  ...
   def resize(s: Size) { ... ; afterExecResize() }
   def setBounds(x1: Int, y1: Int, x2: Int, y2: Int) { ... ; afterExecSetBounds }
  ...
}
```

REACTIVE LANGUAGES

Events and Functional Dependencies

- Events are often used for functional dependencies

```
val update = new ImperativeEvent[Unit]()
val a = 3
val b = 7
val c = a + b  // Functional dependency

update += ( _ =>{
  c = a + b
})

a = 4
update()
b = 8
update()
```

Constraints

- What about expressing functional dependencies as constraints ?

```
val a = 3
val b = 7
val c = a + b // Statement
println(c)
> 10
a= 4
println(c)
> 10
```

```
val a = 3
val b = 7
val c := a + b // Constraint
println(c)
> 10
a= 4
println(c)
> 11
```

EMBEDDING REACTIVE PROGRAMMING IN SCALA

Reactive Values

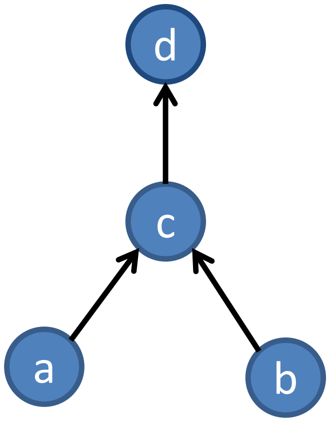
- Vars: primitive reactive values
- Signals: reactive expressions

- Important design property:
  - Signals can be further composed

```
val a = Var(3)
val b = Var(7)
val c = Signal{ a() + b() }
Println(c.getVal)
> 10
a()= 4
println(c.getVal)
> 11
```

Reference Model

- Change propagation model
  - Dependency graph
  - Push-driven evaluation




```
val a = Var(3)
val b = Var(7)
val c = Signal{ a() + b() }
val d = Signal { 2 * c() }
```

SIGNALS AND VARS

Vars

- Vars wrap normal Scala values

- Var[T] is a parametric type.
  - The parameter T is the type the var wraps around
  - Vars are assigned by the “()=“ operator

```
val a = Var(0)
val b = Var("Hello World")
val c = Var(false)
val d: Var[Int] = Var(30)
val e: Var[String] = Var("REScala")
val f: Var[Boolean] = Var(false)


a()= 3
b()=“New World”
c()=true
```

Signals

- Syntax: Signal{sigexpr}
  - Sigexpr should be side-effect free
- Signals are parametric types.
  - A signal Signal[T] carries a value of type T

Signals

- Vars or a signals is called with () in a signal expression are added to the dependencies

```
val a = Var(0)
val b = Var(0)
val s = Signal{ a() + b() } // Multiple vars in a signal expression
```

Signals: Examples

```
val a = Var(0)
val b = Var(0)
val c = Var(0)
val r: Signal[Int] = Signal{ a() + 1 }    // Explicit type in var decl
val s = Signal{ a() + b() }               // Multiple vars is a signal expression
val t = Signal{ s() * c() + 10 }          // Mix signals and vars in signal expressions
val u = Signal{ s() * t() }               // A signal that depends on other signals
```

Signals: Examples

```
val a = Var(0)
val b = Var(2)
val c = Var(true)
val s = Signal{ if (c()) a() else b() }
```

```
def factorial(n: Int) = ...
val a = Var(0)
val s: Signal[Int] = Signal{ // A signal expression can be any code block
  val tmp = a() * 2
  val k = factorial(tmp)
  k + 2  // Returns an Int
}
```

Signals

- Accessing reactive values: getVal

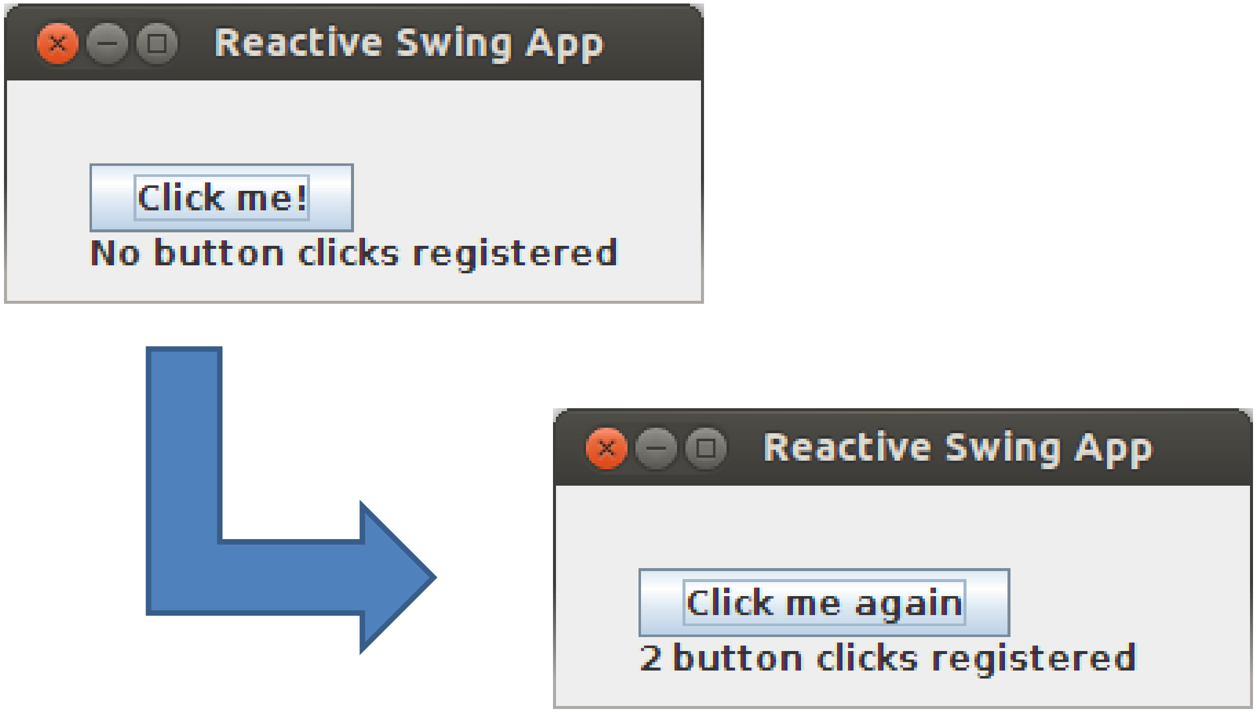
```
val a = Var(0)
val b = Var(2)
val c = Var(true)
val s: Signal[Int] = Signal{ a() + b() }
val t: Signal[Boolean] = Signal{ !c() }

val x: Int = a.getVal
val y: Int = s.getVal

val z: Boolean = t.getVal
println(z)
```

EXAMPLES OF SIGNALS

Example


Reactive Swing App
Click me!
No button clicks registered
Reactive Swing App
Click me again
2 button clicks registered

Example: Observer

```
/* Create the graphics */
title = "Reactive Swing App"
val button = new Button {
  text = "Click me!"
}
val label = new Label {
  text = "No button clicks registered"
}
contents = new BoxPanel(Orientation.Vertical) {
  contents += button
  contents += label
}
```

```
/* The logic */
listenTo(button)
var nClicks = 0
reactions += {
  case ButtonClicked(b) =>
    nClicks += 1
    label.text = "Number of button clicks: " + nClicks
    if (nClicks > 0)
      button.text = "Click me again"
}
```

Example: Signals

```
title = "Reactive Swing App"
val label = new ReactiveLabel
val button = new ReactiveButton

val nClicks = button.clicked.fold(0) {(x, _) => x + 1}

label.text = Signal { ( if (nClicks() == 0) "No" else nClicks() ) + " button clicks registered" }

button.text = Signal { "Click me" + (if (nClicks() == 0) "!" else " again " )}

contents = new BoxPanel(Orientation.Vertical) {
  contents += button
  contents += label
}
```

Example: Smashing Particles

```
class Oval(center: Signal[Point], radius: Signal[Int]) { … }

val base = Var(0)  // Increases indefinitely
val simpleTime = Signal{ base() }
val time = Signal{simpleTime() % 200}    // cyclic time

val point1 = Signal{ new Point(20+time(), 20+time()) }
new Oval(point1, time)
… // 4 times
```

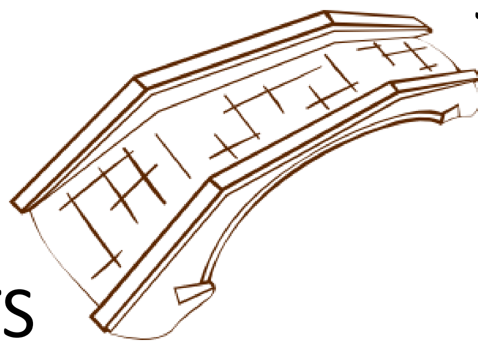
BASIC CONVERSION FUNCTIONS

REScala design principles

- Signals (and events) are objects fields
  - Inheritance, late binding, visibility modifiers, …

- Conversion functions
  bridge signals and events

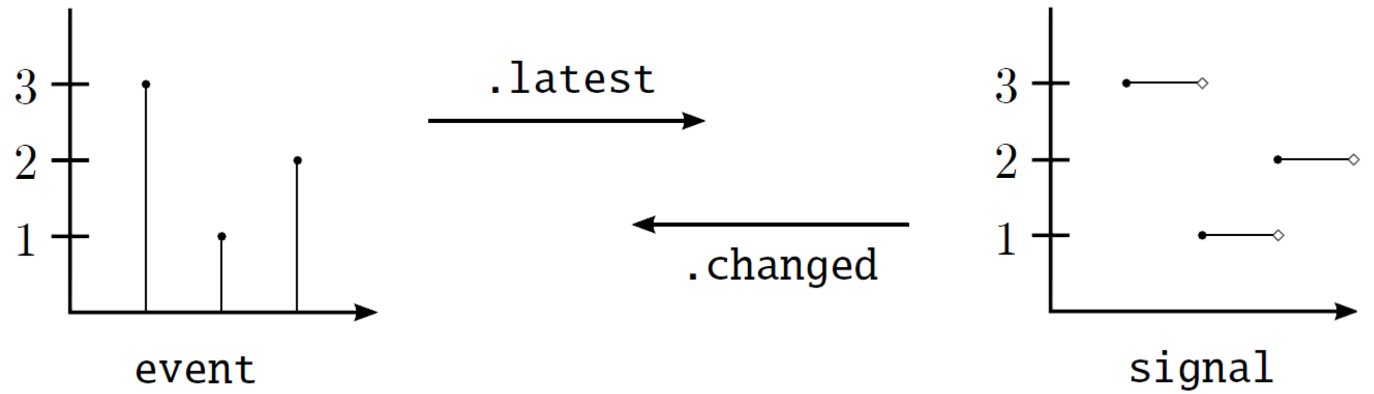
SIGNALS



EVENTS

Basic Conversion Functions

- **`Changed :: Signal[T] -> Event[T]`**
- **`Latest :: Event[T] -> Signal[T]`**



Example: Changed

```
val SPEED = 10
val time = Var(0)
val space = Signal{ SPEED * time() }
space.changed += ((x: Int) => println(x))
while (true) {
  Thread sleep 20
  time() = time.getVal + 1
}

-- output --
10
20
30
40
 ...
```

Example: Latest

```
val senseTmp = new ImperativeEvent[Int]() // Fahrenheit
val threshold = 40

val fahrenheitTmp = senseTmp.latest
val celsiusTmp = Signal{ fahrenheitTmp() - 32 }
val alert = Signal{ if (celsiusTmp() > threshold ) “Warning ” else “OK”  }
```

QUESTIONS?