How Scala Experience Improved Our Java Development

A Case Study at PhET Interactive Simulations

Sam Reid
University of Colorado
reids@colorado.edu

Abstract

PhET Interactive Simulations at the University of Colorado creates free, open-source educational simulations. After developing several simulations in Scala, we identified several advantageous techniques and patterns in Scala which we were able to transfer to subsequent Java development projects. Specifically, our experience with Scala helped us attain the following advantages in our Java development: improved object initialization, improved code organization, reduced code duplication and improved management of boilerplate code. These effect of these changes has been to make our code easier to write, read and maintain. These ideas are not specific to our application domain, but should work equally well in broad range of domains. We also discuss how adoption of these Scala-like patterns in Java code can simplify the learning curve for Java developers who want to learn Scala.

Keywords Scala, Java, Closures, Declarative Programming, Design Patterns, Functional Programming, Programming Style

1. Introduction

PhET Interactive Simulations at the University of Colorado[PhET] creates free and publicly available open source software for science education. Simulations (sims) in physics, biology, chemistry, mathematics and other fields help students to visualize, interact with and experiment with different scientific phenomena. The simulations are primarily written in Java and Flash, but we have recently used Flex and Scala. Our sims have been translated into 58 languages and are launched over 2 million times per month. Our Scala sims [Forces and Motion, Ramp: Forces and Motion, Ladybug Motion 2D, Gravity Force Lab] were launched over 50,000 times in the last month.

Our recent Scala experience provided us an opportunity to reflect on patterns and idioms that we took for granted in our Java development, and we were able to transfer some of Scala's advantages to subsequent Java development projects. Specifically, we were able to improve object initialization, reduce code duplication in implementations of the observer pattern, keep more related code together and to improve management of boilerplate code in functional-style programming. These advantages transferred to Java through a change in Java programming style, development of appropriate APIs and usage of IDE features. Many other Scala features were valuable to our product development, but in this paper we restrict our focus to Scala features that helped us to enhance our Java development.

The techniques we describe for improved Java development are not specific to our domain of interactive science simulations, and we expect them to work well on a variety of different Java projects. Furthermore, since application of these patterns leads to Java code that is "Scala-flavored", these patterns have the capacity to help Java developers learn Scala more quickly. We begin with a discussion of how Scala provides improved support for object initialization over idiomatic Java, and how we were able to attain similar benefits in our Java development.

2. Object Initialization

The best way to configure a Java instance is by passing all configurational parameters into the constructor–however, some APIs do not expose all configurational settings via constructor parameters, or addition configuration is necessary. In this case, the idiomatic way to initialize an object in Java is to instantiate the object, then to call a variety of mutators on the object (see the Java Language Tutorial [Oracle]). For instance:

```
//idiomatic Java
JFrame frame = new JFrame("Test");
frame.setContentPane(new JButton("Push me"));
frame.pack();
frame.setVisible(true);
...
//More application code
```

This mutation-based style of object initialization has several disadvantages. First, it requires a name and reference for a temporary variable (in this case, "frame") that is only used for configuring the object, and is not used elsewhere. This extraneous variable declaration pollutes the variable namespace and makes it less apparent to the maintainer where the variable is relevant, since it can be used anywhere in its declared scope. Another problem with mutation-based initialization is that the object transitions through several states before reaching the final correct state. Having an object in an incorrect or inconsistent state can lead to introduction of intermediate bugs, or difficulty in debugging. For instance, if the developer introduces additional functionality, it may separate the initialization code, or lead to a case in which the object is used or inspected before its initialization is complete.

Named variables for object declaration can cause a bug if the developer copies and pastes initialization code, and neglects to change some of the variable name references. For instance, the new program might read:

```
JFrame frame1 = new JFrame("Test");
frame1.setContentPane(new JButton("Push me"));
frame1.pack();
```

```
frame1.setVisible(true);

JFrame frame2 = new JFrame("Second frame");
frame1.setContentPane(new JButton("Push me"));//
    bug from copy-paste
frame2.pack();
frame2.setVisible(true);
```

The copy-paste bug on line 6 would cause incorrect behavior because the pack method is called on the wrong instance.

Scala provides a declarative syntax for solving these problems. The idiomatic Scala technique for instantiating an object (which does not expose all relevant parameters via constructor arguments) is to create an anonymous subclass with an initializer block which calls additional statements [Odersky et al.]:

```
//idiomatic Scala
new JFrame("Test"){
  setContentPane(new JButton("Push the button"))
  pack()
}.setVisible(true)
```

Other modern languages such as JavaFX Script, Flex MXML and Android XML use a similar declarative syntax for object initialization.

Fortunately, Java provides a similar, though rarely used, language construct called an 'instance initializer' or 'double brace initialization'. This technique is most commonly applied to initialization of collection classes, such as:

```
List<Double> list = new ArrayList<Double>(){{
    add(3.0);
    add(4.0);
}};
```

However, this approach can be used productively for any instance initialization, not just for collection instances. For instance, the previous example could be written:

```
new JFrame("Test"){{
  setContentPane(new JButton("Push the button"));
  pack();
}}.setVisible(true);
```

Here is a larger example constructed to help accentuate some of the differences. First, the idiomatic Java version:

```
JTextField numeratorTextField = new JTextField( "
    enter numerator" );
JTextField denominatorTextField = new JTextField(
     "enter denominator" );
JPanel textFieldPanel = new JPanel( new
    GridLayout( 2, 1 ) );
textFieldPanel.add( numeratorTextField );
textFieldPanel.add( denominatorTextField );

JPanel contentPanel = new JPanel();
contentPanel.add( textFieldPanel );
final JButton evaluateFractionButton = new
    JButton( "evaluate fraction" );
evaluateFractionButton.addActionListener(...);
contentPanel.add( evaluateFractionButton );
JTextField resultTextField new JTextField( "
    result" )
contentPanel.add( resultTextField );

JFrame frame = new JFrame( "Test" );
frame.setContentPane( contentPanel );
frame.pack();
frame.setVisible( true );
```

Second, the "double-brace initialization" version:

```
new JFrame("Test"){{
  setContentPane(new JPanel(){{
    add(new JPanel(new GridLayout(2,1)){{
      add(new JTextField("enter numerator"));
      add(new JTextField("enter denominator"));
    }});
    add(new JButton("evaluate fraction"){{
      addActionListener(...);
    }});
    add(new JTextField("result"));
    }});
    pack();
}}.setVisible(true);
```

In either version, there are 3 text fields, 2 panels, a button and a frame. In the idiomatic Java version, all components are created in the same scope, and in order to understand the relationship between them, the reader must trace the creation of each instance to its usage; in this sense it is like a many-to-many map from variable declarations to usages. In the double-brace initialization example, the component layout can be understood more quickly because the code structure mirrors the component layout structure.

Furthermore, the idiomatic Java version has a bottom-up structure in which the leaves are constructed first then assembled. In contrast, the double-brace initializer pattern has a top down structure, where the top level components are described first, and branches and leaves are created and attached as necessary. The top-down approach is often a more natural way of thinking about or describing the system (e.g., "a panel that contains a button"). The bottom-up style also makes it possible to introduce the commonly-written bug in which an object is instantiated and configured but not added to its container. Pollution of the variable namespace in the bottom-up style also makes it possible for the developer to inadvertently use or incorrectly modify references after they are created.

The double-brace initialization pattern can be thought of as a way to "factor-out" duplicated code, where the code that is removed is the variable name followed by the dot operator. Also, usage of the double-brace initialization pattern makes conversion to top-level classes trivial; the content from the double-brace initializer can be copied into the main constructor of the new top-level class without modification (the mutation-based approach requires manually removing duplicated variable name and dot operator usages).

Finally, note that the idiomatic Java version is more verbose, requiring more code to be read, understood, debugged and maintained.

Scala provides a superior mechanism for object initialization in named and default parameters. Ideally, a class would provide constructor arguments for all relevant parameters; in some cases this number can be overwhelming, but with default and named parameters, Scala is able to construct the instance properly initially, without requiring any side-effect based mutation. Unfortunately, it is only possible to used named parameters if the API exposes all relevant parameters as constructor arguments (instead of set methods), which is sometimes not provided by 3rd party code.

```
//Scala with named parameters and appropriate API
new JFrame(
    title = "hello",
    contentPane = ...
);

//Java with instance initializers
new JFrame(){{
  setTitle("hello");
  setContentPane(...);
}};
```

```
//Idiomatic Java
JFrame f = new JFrame();
f.setTitle("hello");
f.setContentPane(...);
```

2.1 Disadvantages

While the double-brace initialization pattern works well in a variety of cases, there are a few caveats and contraindications.

One of the primary disadvantages occurs when the inner instance is declared in an outer class with methods with identical signatures. In this case, there may be some confusion or difficulty in understanding which instance will receive the method dispatch. The rule is that the method is dispatched on the object with the innermost scope with a matching method signature, but sometimes with several nested objects of similar/differing types it can be difficult to quickly identify the receiving object; further difficulty arises when it is necessary to call a method on the outer class from the inner class. This problem can be worked around by giving the outer class a name like so:

```
new JFrame("Test"){{
  JFrame parent = this;
  setContentPane(new JButton("Press Me"){{
    Point parentLoc = parent.getLocation();
  }});
}}
```

When an instance is being supplied as a 2nd or later argument in a method call, or when this pattern is used to instantiate multiple instances in a method call, usage of the double brace initialization pattern can make code harder to read. Since Java doesn't support named arguments, IDE support is often necessary in these cases to understand how the instances are being used. In cases like these, it can sometimes be beneficial to assign the instance to a declared variable (which may be initialized with a double-brace initializer) to simplify its readability at the usage point. Variable declarations are also appropriate when factoring out duplicated code, when readability can be improved or when there is a complex interplay between several instances. But often in the case of object creation, configuration and passing, temporary variable declarations can be productively replaced with double-brace initialization.

Since double-brace initialization creates an anonymous subclass, this increases the executable file size by the overhead of a new class. In a complied and Proguarded JAR, this amounts to approximately 0.67kb per double-brace initialization usage. For projects in which the runtime size is of utmost importance, it may be necessary to avoid using the double-brace initialization pattern to avoid this cost. Similarly, instance initializers in Scala incur this cost of a new anonymous class. For our projects, this cost is not practically significant, as it would take approximately 20 anonymous class declarations to equal 1% of a typical deployed JAR file size. No runtime time disadvantages were immediately perceptible, but we didn't experiment quantitatively on the additional time required to load these anonymous classes, or other runtime performance costs incurred by this pattern.

Care should be taken to ensure that equality tests according to the equals() method properly handle anonymous subclasses. Equality tests that perform class instance equality tests (such as aClass.equals(bClass)) will always report that double-brace initialized objects are unequal. Proper implementation of the equals method will avoid this problem. In our domain, this is rarely problematic since equality tests are seldom required where double-brace initialization patterns are applied.

Despite its disadvantages, double-brace initialization has worked well throughout our Java projects, with the largest benefits in user interface and structured graphics code. In the next section, we show how our experience with Scala helped us to reduce code duplication in implementations of the Observer pattern in Java development.

3. Observer Pattern

When implementing the Observer pattern in Java, we found that we were often duplicating initialization code, such as:

```
public View(Model model){
    model.addObserver(new Observer(){
        public void changed(){
            update();
        }
    });
    update();//duplicate call
}
private void update(){
    //Update the view state based on the model
        state
}
```

This was problematic since inadvertent omission of the update() call in the constructor would lead to inconsistent startup states for the view. This also causes the update method implementation (line 9) to become spatially separated from its usages in line 4 and 7. In Scala, we solved this problem by using a method with a named parameter that would both invoke the block and register it as a listener with the instance to be observed:

```
def invokeAndPass(addListener: (=> Unit) => Unit)
    (update: => Unit) = {
    update
    addListener(update)
}
```

This method would be applied like this:

```
invokeAndPass(model.addListener) {
    //Update the view state based on the model
        state
}
```

Since this invokes the method and registers with the model to be called back for change events, it solves the problem of duplicated calls and code dislocation. We also found it productive to create general wrappers for swing components such as:

```
class MyCheckBox(text: String,
    actionListener: Boolean => Unit,
    getter: => Boolean,
    addListener: (() => Unit) => Unit)
        extends CheckBox(text){...}
```

This component could be used in application code like so:

```
add(new MyCheckBox(``visible'', visible = _,
    visible, addListener));
```

To attain these same benefits in Java, we developed a class Property<T> which is used to represent the Observable component of the Observer pattern, with some additional functionality. To solve the problem of duplicated update calls, the Property immediately calls back to the listener's update method upon listener registration. Not only does this free the developer from needing to call update() manually, but it ensures that the view synchronizes with the model as soon as possible. Since the callback implementation can be implemented in an anonymous inner class implementation, the implementation can appear near the point of usage, thus improving code organization. The Property interface also combines

the getter, setter and addListener methods described in the parameters of the MyRadioButton above into a cohesive interface, so that it can be productively re-used in different types of components.

Once we introduced this pattern, we also found that other duplicated code could be productively factored into Property<T>, such as guarded setting (that is, only notifying observers if the state truly changed as determined by equals()), and reset functionality, which is a domain-specific behavior used to reset our simulation states. By chaining together multiple Property objects, we ensure that program values are updated instantly whenever dependency values change. In general, this leads to a tree of dependencies that cascades changes from the model through to the view. Furthermore, declaring the update tree in the declaration of the properties has shown to be advantageous because it is consolidated instead of scattered around. By constructing appropriate consumers of Property<T> instances, such as PropertyRadioButton (analogous to MyCheckBox above), we can approximate a behavior similar to variable binding in JavaFX Script, so that the view is automatically synchronized with the model.

For instance, here is an idiomatic Java version, showing the computation of momentum = mass times velocity ($p = mv$) and providing an interface for listening to changes:

```
//idiomatic Java
double mass = 3;
double velocity = 4;
public double getMomentum(){
    return mass*velocity;
}
public interface Listener{
    void massChanged(double newMass);
    void velocityChanged(double newVelocity);
    void momentumChanged(double newMomentum);
}
public void setMass(double mass){
    this.mass = mass;
    for (Listener listener : listeners){
        listener.massChanged(mass);
        listener.momentumChanged(getMomentum());
    }
}
...
```

The following Property<T>-based Java code block also sets momentum to be the product of mass and velocity, but also automatically updates and notifies its listeners whenever mass or velocity changes. In this model, all 3 variables are independently observable with the same interface, so that different views and controllers have the potential for code re-use. In this example, type declarations are omitted to improve readability:

```
//Property<T>-based Java
mass = new Property<Double>(3.0);
velocity = new Property<Double>(4.0);
momentum = new Property<Double>() {{
    final VoidFunction0 update = new
        VoidFunction0() {
        public void apply() {
            setValue( mass.getValue() * velocity.
                getValue() );
        }
    };
    mass.addChangeListener( update );
    velocity.addChangeListener( update );
}};
System.out.println( ``momentum.getValue() = '' +
    momentum.getValue() );
```

We have also started experimenting with an internal DSL for combining Property instances. For instance, with the right API support, the above could be rewritten as:

```
//Property<T>-based Java with internal DSL
DoubleProperty mass = new DoubleProperty(3.0);
DoubleProperty vel new DoubleProperty(4.0);
DoubleProperty momentum = mass.times(vel);
System.out.println( "momentum.getValue() = " +
    momentum.getValue() );
```

Another advantage of the Property<T> paradigm is that it isolates and standardizes object mutability; values wrapped in the Property should be immutable, and the Property provides uniform and standardized support for managing and observing object state changes. This pattern also allowed us to reduce a significant amount of duplicated code in setting up components such as user interface components, as in the MyCheckBox example above.

The Property<T> paradigm has several disadvantages. First, it is more verbose than simply using the raw T types, because getValue() must be called at each usage point. In Scala, this cost could be mitigated by making this an apply() method, which could be suppressed in usages. Second, working with types like Property<Double> and Property<Integer> instead of double and int can create some additional cognitive overhead since this is not idiomatic Java. Finally, when the update method is expensive, other approaches must be taken to ensure that multiple change notifications can be batched together to avoid the expense of redundant update calls. Despite these disadvantages, Property<T> paradigm has helped us to improve our Java development, reduce the number of lines of code and eliminate buggy alternatives.

4. Keeping Related Code Together

In idiomatic Java, private methods are employed in order to prevent code duplication and to isolate specific functionality. However, this paradigm often has the disadvantage that related code becomes spatially dislocated. For instance, here is an example showing how a private method is used during a complex object constructor implementation:

```
public PrismGraphic(){
    //Initialization code
    this.color = createColor(wavelength);
    this.neighborColor = createColor(wavelength
        +10);
    //Many more lines of complex initialization
        code
    //...
}
private Color createColor(double wavelength){
    //code to return the Color for a wavelength
}
```

Scala provides two features that help the developer to keep related code together. In Scala, the primary constructor is synonymous with the class body, and fields and methods can be declared at any location in the top level of the class body. Thus in Scala, the createColor() method could appear immediately adjacent to the usages of createColor(), thus increasing code readability and understandability. Furthermore, Scala allows inner function definitions, which enables the developer to place the createColor() method adjacent to the usages. The other advantage of providing a local inner function implementation is that it provides an restricted scope in which the function can be used; defining a private createColor() function at the class level does not indicate to the maintainer where the method is intended to be used since it is scoped to the entire class definition.

Based on these advantages, we started following the same paradigm in Java by relying more on anonymous inner classes to provide function implementations. Initially, we just applied this pattern to anonymous inner class implementations for listener callbacks (as in Section 3), such as:

```
public View(final Model model){
    model.addListener(new ModelListener(){
        void modelChanged(){
            name = model.getName();
            repaint();
        }
    });
}
```

But we subsequently found it useful in some scenarios to simulate inner functions by using a templated Function declaration, such as:

```
public PrismGraphic(){
    //Initialization code
    Function1<Double,Color> createColor =
        new Function1<Double,Color>(){
            void apply(Double d){
            //code to return the Color for a
                wavelength
            }
    };
    this.color = createColor.apply(wavelength);
    this.neighborColor = createColor.apply(
        wavelength+10);
    //Many more lines of complex initialization
        code
    //...
}
```

While more verbose than Scala, this technique ensures that the code is located near to its usages so that both can be easily seen simultaneously, and since the createColor function object is localized in scope to the constructor, it immediately cues the maintainer that it is only relevant to that scope.

Two disadvantages of using anonymous inner classes to implement functions are: when this pattern is applied recursively it can lead to many levels of nesting that can be difficult to read at a glance without proper commenting, and that it introduces additional boilerplate code. In the next section, we describe an IDE feature that helps to mitigate the cost of the additional boilerplate code.

5. IDE Support for Managing Boilerplate Code

One of the most frequently cited advantages of Scala over Java is the reduction in the amount of boilerplate code, which is code that is unrelated to the problem at-hand but necessary in order to implement the solution. Boilerplate entails two costs: the initial upfront cost of writing the boilerplate code and the increase in the noise level in the code when reading, debugging or maintaining. Our experience with Scala helped us to identify this background noise in Java and to investigate techniques to mitigate it. First, we discuss the issue of boilerplate creation, and then we discuss a technique for hiding boilerplate code in closure creation in Java.

5.1 Boilerplate Creation

Modern IDEs have solved the problem of boilerplate creation with features such as smart templates and auto-complete. For instance, to add an ActionListener to a Swing JButton in Java, in IntelliJ, the developer types the following text (here <a> stands for an autocomplete keypress and <p> indicates invocation of a smart template for printing to System.out.):

```
JButton button = new <a>"push me"
button.adda <a> new <a>
<p> hello
```

to create the following code:

```
JButton button = new JButton("push me");
button.addActionListener(new ActionListener(){
 public void actionPerformed(ActionEvent e){
        System.out.println("hello");
  }
});
```

In contrast, in the Scala version, the developer only needs to type approximately the same as in the autocomplete Java version above:

```
val button = new SButton();
button.addA<a>(() => <p>"hello");
```

This creates the more concise Scala implementation:

```
val button = new SButton();
button.addActionListener(() => println("hello"));
```

Of course, boilerplate creation only provides a savings on the initial cost of writing the code–the full code is still checked in to version control and must be read, understood, debugged and maintained. An IDE feature called closure folding can help mitigate these remaining costs by helping the developer to focus on the important parts of the code.

5.2 Boilerplate Folding

IntelliJ Idea provides an IDE setting that allows closures to be folded. That means that a closure like:

```
button.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e){
        System.out.println("hello");
    }
} );
```

is folded by the IDE editor to read:

```
button.addActionListener(ActionListener(
    ActionEvent e){System.out.println("hello");})
```

While the equivalent Scala expression might be written (given the appropriate Scala API):

```
button.addActionListener( (e:ActionEvent) =>
    println("hello") )
```

Specifically, this IDE feature suppresses method names and one nested scope for anonymous inner implementations that have a single method. While closure code folding doesn't solve the underlying problem of the existence of noisy boilerplate code, it mitigates the costs by making closure implementations easier to understand at a glance, and by making it easier to see more related code at the same time. This makes it easier to employ anonymous inner classes for functional programming patterns while maintaining code readability.

6. Learning Scala

A Java developer learning Scala is faced with many new concepts, including syntax, language features, new APIs and different styles. The techniques described in these papers have the capacity to mitigate some of the costs of this learning curve by providing examples

and analogies for some parts of typical Scala style and language features. First, usage of the double-brace initialization pattern in Java transfers over directly to idiomatic Scala anonymous subclass declaration (Section 2). Second, use of Property<T> instances is a step toward functional programming, where the Property<T> is like a Function0<T> which also signifies when its values have changed (Section 3). Declaring local functions through anonymous inner classes is a step toward functional thinking, and toward moe optimal code organization as it would be done in an idiomatic Scala program (Section 4). Finally, by using techniques such as boilerplate folding (Section 5.2), Java code is made to look more concise and more similar to Scala code by hiding some of the unnecessary boilerplate from view.

7. Conclusions

While the techniques proposed in this paper are not idiomatic Java style, they should be preferred because of their tendency to improve code organization, avoid bugs and to improve readability and maintainability. Since we have only recently started applying and refining these ideas, we are unable to make any large-scale maintainability claims; we are also unable to comment on how well a new Java programmer would be able to pick up and move forward with these paradigms, since they are atypical Java. But in the short term, our development and maintenance have been significantly improved. Furthermore, despite the simplicity of the examples in this paper, we found these approaches to scale up well to complex real-world problems. Scala also provided many beneficial language features for which we have not been able to find an comparably expressive Java implementation, such as traits, implicit conversions, operator infix notation (valuable for 2d vector math), for comprehensions and functional programming.

Acknowledgments

Thanks to NSF and the Hewlett Foundation for supporting the PhET Interactive Simulations project and to the other PhET developers John Blanco, Chris Malley and Jon Olson.

References

[PhET] http://phet.colorado.edu

[Ladybug Motion 2D] http://phet.colorado.edu/en/simulation/ladybug-motion-2d

[Forces and Motion] http://phet.colorado.edu/en/simulation/forces-and-motion

[Ramp: Forces and Motion] http://phet.colorado.edu/en/simulation/ramp-forces-and-motion

[Gravity Force Lab] http://phet.colorado.edu/en/simulation/gravity-force-lab

[Odersky et al.] M. Odersky, L. Spoon and B. Venners. Programming in Scala. 2008

[Oracle] http://download.oracle.com/javase/tutorial/uiswing/components/button.html