

The busy Java developer's guide to Scala: Of traits and behaviors

Using Scala's version of Java interfaces

Skill Level: Introductory

[Ted Neward \(ted@tedneward.com\)](mailto:ted@tedneward.com)

Principal
Neward & Associates

29 Apr 2008

Scala doesn't just bring functional concepts to the JVM, it offers us a modern perspective on object-oriented language design. In this month's installment of [The busy Java developer's guide to Scala](#), Ted Neward shows you how Scala exploits traits to make objects simpler and easier to build. As you'll learn, traits are both similar to and different from the traditional polarities offered by Java™ interfaces and C++ multiple inheritance.

Sir Isaac Newton, famous scientist and researcher, is credited with the words: "If I have seen further it is by standing on the shoulders of giants." As an avid historian and political scientist, I might offer a slight revision to the great man's quote: "If I have seen further, it is because I have stood on the shoulders of history." These words reflect another statement, made by the historian George Santayana: "Those who cannot remember the past are condemned to repeat it." In other words, if we cannot look back upon history and learn from the mistakes made by those who came before us (including ourselves), then there is little chance of improvement.

So, you're wondering, what does this philosophizing have to do with Scala? Inheritance, for one thing. Consider the fact that the Java language was created close to 20 years ago, in the heyday of "object-orientation." It was designed to mimic C++, the dominant language of the day, in a naked attempt to woo that language's developers over to the Java platform. Certain decisions were made that seemed obvious and necessary at the time, but in retrospect, we know that some of them are not as beneficial as the creators then believed.

For instance, 20 years ago, it made sense to the creators of the Java language to reject both C++-style private inheritance and multiple inheritance. Since that time, many Java developers have had cause to regret their decision. In this month's guide

to Scala, I revisit the history of multiple and private inheritance in the Java language. Then, you'll see what Scala does to rewrite that history for the greater benefit of us all.

Inheritance in C++ and the Java language

History is that version of events that people have decided to agree on.
— Napoleon Bonaparte

Those who labored in the C++ mines will recall that private inheritance was a way of absorbing behavior from a base class without explicitly accepting the *IS-A* relationship. Marking the base class as "private" allowed the derived class to inherit from it without actually *becoming* one of them. Private inheritance on its own was one of those features that never quite took, however. The idea of inheriting from a base class without being able to downcast to it or upcast to the base just seemed silly.

About this series

Ted Neward dives into the Scala programming language and takes you along with him. In this new developerWorks [series](#), you'll learn what all the recent hype is about and see some of Scala's linguistic capabilities in action. Scala code and Java code will be shown side by side wherever comparison is relevant, but (as you'll discover) many things in Scala have no direct correlation to anything you've found in Java programming— and therein lies much of Scala's charm! After all, if Java code could do it, why bother learning Scala?

Multiple inheritance, on the other hand, was commonly held as a necessary element of object-oriented programming. When modeling a hierarchy of vehicles, the `SeaPlane` clearly needs to inherit from both `Boat` (with its methods `startEngine()` and `sail()`) and `Plane` (with its methods `startEngine()` and `fly()`). A `SeaPlane` acts as both a `Boat` and a `Plane`, doesn't it?

That was the thinking in the golden age of C++, anyway. When we fast-forward to the Java language, we perceive that multiple inheritance is just as flawed as private inheritance. Instead, any Java developer will tell you, `SeaPlane` should inherit from the interfaces `Floatable` and `Flyable` (and probably the interface or base class `EnginePowered`, as well). Inheritance from interfaces means being able to implement all the methods the class requires without encountering the horror of *virtual multiple inheritance* (wherein we attempt to solve the question of which base's `startEngine()` to call when `SeaPlane`'s `startEngine()` method is called).

Unfortunately, tossing out private and multiple inheritance has cost us dearly in terms of code reuse. Java developers may rejoice in being free of virtual multiple inheritance, but the trade-off is often painstaking — and error-prone — work done by the programmer.

Reusable behavior, revisited

Events ... may be roughly divided into those which probably never happened and those which do not matter.
— William Ralph Inge

The JavaBeans specification is a foundation of the Java platform, giving rise to the POJO that so much of our Java ecosystem depends on. We're all aware of the idea that properties in Java code are managed by `get()`/`set()` pairs, as shown in Listing 1:

Listing 1. Person POJO

```
//This is Java
public class Person
{
    private String lastName;
    private String firstName;
    private int age;

    public Person(String fn, String ln, int a)
    {
        lastName = ln; firstName = fn; age = a;
    }

    public String getFirstName() { return firstName; }
    public void setFirstName(String v) { firstName = v; }
    public String getLastName() { return lastName; }
    public void setLastName(String v) { lastName = v; }
    public int getAge() { return age; }
    public void setAge(int v) { age = v; }
}
```

That looks fairly simple and not so hard to do. But what if you wanted to provide notification support — such that third parties could register with a POJO and receive callbacks when a property changes? According to the JavaBeans spec, you would then have to implement the `PropertyChangeListener` interface and its single method, `propertyChange()`. If you wanted to allow any of the POJO's `PropertyChangeListeners` to "vote" on the changes to properties, then your POJO would also need to implement the `VetoableChangeListener` interface, which requires the `vetoableChange()` method to be implemented.

At least, that's how it's supposed to work.

In fact, the `PropertyChangeListener` interface must be implemented by would-be recipients of the notifications of property changes, and the sender (in this case the `Person` class) must offer up public methods that take instances of this interface, as well as the name of the property that the listener wants to listen to. The end result is the more complicated `Person` shown in Listing 2:

Listing 2. Person POJO, take 2

```
//This is Java
public class Person
```

```
{
    // rest as before, except that inside each setter we have to do something
    // like:
    // public setFoo(T newValue)
    // {
    //     T oldValue = foo;
    //     foo = newValue;
    //     pcs.firePropertyChange("foo", oldValue, newValue);
    // }

    public void addPropertyChangeListener(PropertyChangeListener pcl)
    {
        // keep a reference to pcl
    }
    public void removePropertyChangeListener(PropertyChangeListener pcl)
    {
        // find the reference to pcl and remove it
    }
}
```

Keeping the reference to the property change listener means the `Person` POJO has to keep some kind of collection class (such as `ArrayList`) to contain all the references. The POJO then has to be instantiated, inserted into, and removed from — and, because these actions are not atomic, it also has to contain appropriate synchronization protections.

Finally, if a property changes, the list of property listeners must be notified, usually by iterating through the collection of `PropertyChangeListeners` and calling `propertyChange()` on each one of them. And that process includes passing in a new `PropertyChangeEvent` describing the property, the old value, and the new value, as demanded by the `PropertyChangeEvent` class and the JavaBeans spec.

It's no wonder that so few of our written POJOs support listener notification: it's a ton of work, and it has to be repeated, by hand, for every JavaBean/POJO created.

Work, work, work where's the workaround?

Interestingly, had C++'s support for private inheritance carried over into the Java language, we could use it today to resolve some of the conundrums of the JavaBeans specification. A base class could provide the POJO's basic `add()` and `remove()` methods, the collection class, and the `firePropertyChanged()` method to notify the listeners of property changes.

We could still do that with the Java class, but because Java lacks private inheritance, the `Person` class would have to inherit from a base `Bean` class and would therefore be *upcastable* to `Bean`. This would preclude `Person` from inheriting from any other class. Multiple inheritance might save us from the latter problem, but it would also lead us back to virtual inheritance, which we definitely want to avoid.

The Java language solution to this problem is the well-known idiom of a *support* class, in this case `PropertyChangeSupport`: instantiate one of these inside the POJO, put the necessary public methods on the POJO itself, and each of the public methods calls into the `Support` class to do the dirty work. Here's the `Person` POJO updated to use `PropertyChangeSupport`:

Listing 3. Person POJO, take 3

```
//This is Java
import java.beans.*;

public class Person
{
    private String lastName;
    private String firstName;
    private int age;

    private PropertyChangeSupport propChgSupport =
        new PropertyChangeSupport(this);

    public Person(String fn, String ln, int a)
    {
        lastName = ln; firstName = fn; age = a;
    }

    public String getFirstName() { return firstName; }
    public void setFirstName(String newValue)
    {
        String old = firstName;
        firstName = newValue;
        propChgSupport.firePropertyChange("firstName", old, newValue);
    }

    public String getLastName() { return lastName; }
    public void setLastName(String newValue)
    {
        String old = lastName;
        lastName = newValue;
        propChgSupport.firePropertyChange("lastName", old, newValue);
    }

    public int getAge() { return age; }
    public void setAge(int newValue)
    {
        int old = age;
        age = newValue;
        propChgSupport.firePropertyChange("age", old, newValue);
    }

    public void addPropertyChangeListener(PropertyChangeListener pcl)
    {
        propChgSupport.addPropertyChangeListener(pcl);
    }
    public void removePropertyChangeListener(PropertyChangeListener pcl)
    {
        propChgSupport.removePropertyChangeListener(pcl);
    }
}
```

I'm not sure about you, but the complexity of that code almost makes me want to take up assembly language again. What's worse is knowing you'll have to repeat this exact sequence of code in every POJO you write. Half of the work in Listing 3 is in the POJO itself, and therefore cannot be reused — except by the traditions of "cut and paste" programming.

Now let's see what Scala has to offer in the way of a better workaround.

Traits and behavioral reuse in Scala

Everyone has the obligation to ponder well his own specific traits of

character. He must also regulate them adequately and not wonder whether someone else's traits might suit him better.
— Cicero

Scala enables you to define a new construct that lies halfway between an interface and a class, called a *trait*. Traits are unusual in that a class can incorporate as many of them as desired, like interfaces, but they can also contain behavior, like classes. Also, like both classes and interfaces, traits can introduce new methods. But unlike either, the definition of that behavior isn't checked until the trait is actually incorporated as part of a class. Or, put differently, you can define methods that aren't checked for correctness until they're incorporated into a trait-using class definition.

Traits may sound complex, but they're easier understood once you've seen them in action. To get started, here's the `Person` POJO redefined in Scala:

Listing 4. Scala's Person POJO

```
//This is Scala
class Person(var firstName:String, var lastName:String, var age:Int)
{
}
```

You could also ensure that your Scala POJO has the `get()`/`set()` methods expected in Java POJO-based environments, simply by using the `scala.reflect.BeanProperty` annotation on the class parameters `firstName`, `lastName`, and `age`. For now, I'll leave those methods out of the equation to keep things simple.

If the `Person` class wants to be able to accept `PropertyChangeListeners`, it can do so as shown in Listing 5:

Listing 5. Scala's Person POJO with listeners

```
//This is Scala
object PCL
  extends java.beans.PropertyChangeListener
{
  override def propertyChange(pce:java.beans.PropertyChangeEvent):Unit =
  {
    System.out.println("Bean changed its " + pce.getPropertyName() +
      " from " + pce.getOldValue() +
      " to " + pce.getNewValue())
  }
}
object App
{
  def main(args:Array[String]):Unit =
  {
    val p = new Person("Jennifer", "Aloi", 28)

    p.addPropertyChangeListener(PCL)

    p.setFirstName("Jenni")
    p.setAge(29)

    System.out.println(p)
  }
}
```

Notice how using the `object` in Listing 5 enables me to register a static method as a listener — something I could never do in my Java code without explicitly creating the `Singleton` class and instantiating it. This is just more evidence for the theory that Scala learns from the historical [pain points](#) of Java development.

The next step for `Person` is to provide the `addPropertyChangeListener()` method and fire `propertyChange()` method calls at each of the listeners when the properties change. In Scala, doing this in a reusable way is as easy as defining and using a trait, as shown in Listing 6. I call this trait `BoundPropertyBean` because "notified" properties are formally called *bound properties* in the JavaBeans specification.

Listing 6. Holy behavioral reuse, Batman!

```
//This is Scala
trait BoundPropertyBean
{
  import java.beans._

  val pcs = new PropertyChangeSupport(this)

  def addPropertyChangeListener(pcl : PropertyChangeListener) =
    pcs.addPropertyChangeListener(pcl)

  def removePropertyChangeListener(pcl : PropertyChangeListener) =
    pcs.removePropertyChangeListener(pcl)

  def firePropertyChange(name : String, oldVal : _, newVal : _) : Unit =
    pcs.firePropertyChange(new PropertyChangeEvent(this, name, oldVal, newVal))
}
```

Again, I'm still making use of the `PropertyChangeSupport` class from the `java.beans` package, not only because it provides about 60 percent of the implementation details I need, but also because that way I have the same behavior as those JavaBeans/POJOs that use it directly. Any additional enhancements to this "Support" class will be propagated through my trait as well. The difference is that now the `Person` POJO doesn't need to worry about using `PropertyChangeSupport` directly, as shown in Listing 7:

Listing 7. Scala's Person POJO, take 2

```
//This is Scala
class Person(var firstName:String, var lastName:String, var age:Int)
  extends Object
  with BoundPropertyBean
{
  override def toString = "[Person: firstName=" + firstName +
    " lastName=" + lastName + " age=" + age + "]"
}
```

After compiling, a quick glance at the `Person` definition reveals that it has the public methods `addPropertyChangeListener()`, `removePropertyChangeListener()`, and `firePropertyChange()`, just as the Java version of `Person` did. In effect, Scala's `Person` version got these new

methods by virtue of just the one additional line of code: the *with* clause in the class declaration that marks the class `Person` as inheriting from the trait `BoundPropertyBean`.

Unfortunately, I'm not quite done yet; the `Person` class now has the support to accept, remove, and notify listeners, but the default methods generated by Scala for the `firstName` member doesn't make use of them. And, equally unfortunate, as of this writing Scala doesn't have a nifty annotation to *automagically* generate the get/set methods that use the `PropertyChangeSupport` instance, so I have to write them myself, as shown in Listing 8:

Listing 8. Scala's `Person` POJO, take 3

```
//This is Scala
class Person(var firstName:String, var lastName:String, var age:Int)
  extends Object
  with BoundPropertyBean
{
  def setFirstName(newvalue:String) =
  {
    val oldvalue = firstName
    firstName = newvalue
    firePropertyChange("firstName", oldvalue, newvalue)
  }

  def setLastName(newvalue:String) =
  {
    val oldvalue = lastName
    lastName = newvalue
    firePropertyChange("lastName", oldvalue, newvalue)
  }

  def setAge(newvalue:Int) =
  {
    val oldvalue = age
    age = newvalue
    firePropertyChange("age", oldvalue, newvalue)
  }

  override def toString = "[Person: firstName=" + firstName +
    " lastName=" + lastName + " age=" + age + "]"
}
```

A good trait to have

Traits are hardly a *functional* concept; rather, they're the result of a decade's worth of hindsight in object programming. In fact, you might find yourself using the following trait without even realizing it in simple Scala programs:

Listing 9. Begone, foul `main()`!

```
//This is Scala
object App extends Application
{
  val p = new Person("Jennifer", "Aloi", 29)

  p.addPropertyChangeListener(PCL)

  p.setFirstName("Jenni")
  p.setAge(30)

  System.out.println(p)
}
```

The `Application` trait defines the same `main()` method that you've been defining by hand all along. In fact, it includes one other useful little tidbit: a *timer*, which times the execution of the application if the system property `scala.time` is passed to the `Application`-implementing code — as shown in Listing 10:

Listing 10. Timing is everything

```
$ scala -Dscala.time App
Bean changed its firstName from Jennifer to Jenni
Bean changed its age from 29 to 30
[Person: firstName=Jenni lastName=Aloi age=30]
[total 15ms]
```

Traits in the JVM

Any sufficiently advanced technology is indistinguishable from magic.

— Arthur C Clarke

At this point, it's fair to ask how the seeming magic of this interface-with-methods construct (*aka* trait) gets mapped onto the JVM. In Listing 11, our good friend `javap` shows us what's happening behind the magic curtain:

Listing 11. Person, cracked open

```
$ javap -classpath C:\Prg\scala-2.7.0-final\lib\scala-library.jar;classes Person
Compiled from "Person.scala"
public class Person extends java.lang.Object implements BoundPropertyBean,scala.
ScalaObject{
    public Person(java.lang.String, java.lang.String, int);
    public java.lang.String toString();
    public void setAge(int);
    public void setLastName(java.lang.String);
    public void setFirstName(java.lang.String);
    public void age_$eq(int);
    public int age();
    public void lastName_$eq(java.lang.String);
    public java.lang.String lastName();
    public void firstName_$eq(java.lang.String);
    public java.lang.String firstName();
    public int $tag();
    public void firePropertyChange(java.lang.String, java.lang.Object, java.lang
.Object);
    public void removePropertyChangeListener(java.beans.PropertyChangeListener);

    public void addPropertyChangeListener(java.beans.PropertyChangeListener);
    public final void pcs_$eq(java.beans.PropertyChangeSupport);
    public final java.beans.PropertyChangeSupport pcs();
}
```

Notice the class declaration for `Person`. This POJO implements an interface called `BoundPropertyBean`, which is how the trait maps to the JVM itself: as an interface. But what about the implementation of the trait's methods? Remember that the compiler can pull all sorts of tricks, just as long as the final result obeys the semantic meaning of the Scala language. In this case, it drops the method

implementations and field declarations defined in the trait into the class that implements the trait, `Person`. Running `javap` with `-private` makes this pretty obvious — if it wasn't already from the last two lines of the `javap` output (referencing the `pcs` val defined in the trait):

Listing 12. Person cracked open, take 2

```
$ javap -private -classpath C:\Prg\scala-2.7.0-final\lib\scala-library.jar;classes Person
Compiled from "Person.scala"
public class Person extends java.lang.Object implements BoundPropertyBean,scala.
ScalaObject{
    private final java.beans.PropertyChangeSupport pcs;
    private int age;
    private java.lang.String lastName;
    private java.lang.String firstName;
    public Person(java.lang.String, java.lang.String, int);
    public java.lang.String toString();
    public void setAge(int);
    public void setLastName(java.lang.String);
    public void setFirstName(java.lang.String);
    public void age_$eq(int);
    public int age();
    public void lastName_$eq(java.lang.String);
    public java.lang.String lastName();
    public void firstName_$eq(java.lang.String);
    public java.lang.String firstName();
    public int $tag();
    public void firePropertyChange(java.lang.String, java.lang.Object, java.lang.Object);
    public void removePropertyChangeListener(java.beans.PropertyChangeListener);

    public void addPropertyChangeListener(java.beans.PropertyChangeListener);
    public final void pcs_$eq(java.beans.PropertyChangeSupport);
    public final java.beans.PropertyChangeSupport pcs();
}
```

In fact, this explanation also answers the question of how the execution of the trait's methods can be deferred until they're used for checking. Because the trait's methods aren't really a "part" of any class until the trait is implemented by that class, the compiler can leave out checking some aspects of the methods' logic until later. This is useful because it allows a trait to call `super()` without having to know what the actual base class of the class implementing the trait will be.

Notes of a trait-or

In the `BoundPropertyBean`, I use trait functionality in the construction of the `PropertyChangeSupport` instance. Its constructor wants the bean on which the properties are notified, and in the trait defined earlier, I passed `"this"`. Because the trait isn't really defined until it's implemented on `Person`, `"this"` will refer to the `Person` instance, not the `BoundPropertyBean` trait itself. This particular facet of traits — the delayed resolution of definitions — is subtle, but it can be powerful for this kind of "late-binding."

In the case of the `Application` trait, the magic takes place in two parts; the `main()` method of the `Application` trait provides the ubiquitous entry point for Java applications, and also does the check for the `-Dscala.time` system property to see if it should track the execution time. However, because `Application` is a trait, the method actually "shows up" on the subclass (`App`). To execute this method, it is necessary to create the `App` singleton, which means constructing an instance of

App, which means "playing" the body of the class, which effectively executes the application. Only after that is complete does the trait's `main()` get invoked and display the time spent executing.

It's a bit backward but it works, with the caveat that the application doesn't have access to any command-line parameters passed in to `main()`. It also demonstrates how the trait's behavior is "deferred" down into the implementing class.

Traits and collections

If you're not part of the solution, you're part of the precipitate.
— Henry J Tillman

Traits are particularly powerful when they combine concrete behavior with abstract declarations to provide convenience to the implementer. For example, consider the classic Java collection interfaces/classes `List` and `ArrayList`. The `List` interface guarantees that the contents of this collection can be traversed in the same order they were inserted, or, in more formal terms, that "positional semantics are honored."

`ArrayList` is a particular type of `List`, storing its contents in an allocated array, whereas `LinkedList` uses a linked-list implementation instead. `ArrayLists` are better for random-access of the contents of the list, whereas `LinkedList` is better for insertions and removal from anywhere but the end of the list. Regardless, it turns out that a phenomenal amount of behavior between these two classes is identical, and as a result, both in turn inherit from a common base class, `AbstractList`.

Had traits been supported in Java programming, they would have been a far superior construct for this tricky kind of "reusable behavior without having to resort to inheriting a common base class" sort of problem. A trait could act as a kind of C++ "private inheritance" mechanism, saving the potential confusion of whether a new `List` subtype should implement `List` directly (and potentially forget to implement the `RandomAccess` interface as well) or extend the base class `AbstractList`. This was sometimes called a "mixin" in C++, though not to be confused with Ruby mixins (or the Scala mixin, which I'll discuss in a later article.)

Within the Scala documentation set, the classic example is the `Ordered` trait, which defines methods-with-funny-names to provide comparison (and thus ordering) capabilities, as shown in Listing 13:

Listing 13. Order, order

```
//This is Scala
trait Ordered[A] {
  def compare(that: A): Int

  def < (that: A): Boolean = (this compare that) < 0
  def > (that: A): Boolean = (this compare that) > 0
  def <= (that: A): Boolean = (this compare that) <= 0
  def >= (that: A): Boolean = (this compare that) >= 0
  def compareTo(that: A): Int = compare(that)
```

```
}
```

Here, the `Ordered` trait (with a parameterized type, *a la* Java 5 generics) defines one abstract method, `compare`, which expects to take an `A` as a parameter and needs to return either less than 1 if this is "less than" that, greater than 1 if this is "greater than" that, or 0 if they are equal. Then it goes on to define the relational operators (`<`, `>`, and so on) in terms of the `compare()` method, as well as the more familiar `compareTo()` method that the `java.util.Comparable` interface also uses.

Scala and Java compatibility

A picture is worth a thousand words. An interface is worth a thousand pictures.
— Ben Shneiderman

Actually, pseudo-implementation-inheritance is not the most common or powerful use of traits within Scala; instead, traits serve as the basic replacement in Scala for Java's interfaces. Java programmers looking to call into Scala should also be familiar with traits as a mechanism for using Scala.

As I have pointed out throughout this series so far, compiled Scala code doesn't always offer high fidelity to the Java language. Recall, for example, that Scala's "methods with funny names" (such as `+` or `\`), are often encoded with characters that aren't directly usable by the Java language syntax (with `$` being the big one to worry about). For that reason, creating a "Java-callable" interface can simplify calling into Scala code.

This particular example is a bit contrived, and the *Scala-isms* used don't really require the layer of indirection that a trait will provide (given I'm not using "methods with funny names"), but bear with me: the concept is the key here. In Listing 14, I want to have a traditional Java-style factory that produces `Student` instances, such as you've commonly seen in a variety of Java object models. To start, I need a Java-compatible interface to the `Student`:

Listing 14. I, Student

```
//This is Scala
trait Student
{
  def getFirstName : String;
  def getLastName : String;
  def setFirstName(fn : String) : Unit;
  def setLastName(fn : String) : Unit;

  def teach(subject : String)
}
```

When compiled, this turns into a POJI: Plain Old Java Interface, as seen by a quick glance with `javap`:

Listing 15. It's a POJI!

```
$ javap Student
Compiled from "Student.scala"
public interface Student extends scala.ScalaObject{
    public abstract void setLastName(java.lang.String);
    public abstract void setFirstName(java.lang.String);
    public abstract java.lang.String getLastName();
    public abstract java.lang.String getFirstName();
    public abstract void teach(java.lang.String);
}
```

Next, I need a class to be the factory itself. Normally, in Java code, this would be a static method on a class (called something like "StudentFactory"), but recall that Scala has no such thing as static methods. Instead, Scala has objects, which are singletons with instance methods. I think that's exactly what I'm looking for here, so I create a StudentFactory object and put my Factory method there:

Listing 16. I Make Students

```
//This is Scala
object StudentFactory
{
    class StudentImpl(var first:String, var last:String, var subject:String)
        extends Student
    {
        def getFirstName : String = first
        def setFirstName(fn: String) : Unit = first = fn
        def getLastName : String = last
        def setLastName(ln: String) : Unit = last = ln

        def teach(subject : String) =
            System.out.println("I know " + subject)
    }

    def getStudent(firstName: String, lastName: String) : Student =
    {
        new StudentImpl(firstName, lastName, "Scala")
    }
}
```

The nested class StudentImpl is the implementation of the Student trait, which thus provides the get()/set() method pairs demanded by it. Remember, despite the fact that traits can have behavior, the fact that it's being modeled against the JVM as an interface means that attempts to instantiate a trait will result in errors claiming that Student is abstract.

The important payout of this trivial little sample, of course, is to write a Java application that can make use of these new Scala-created objects:

Listing 17. Student Neo

```
//This is Scala
public class App
{
    public static void main(String[] args)
    {
        Student s = StudentFactory.getStudent("Neo", "Anderson");
        s.teach("Kung fu");
    }
}
```

```
}
```

Run this and you'll see "I know Kung fu." (That was a long setup for a cheap movie reference, I know.)

In conclusion

People do not like to think. If one thinks, one must reach conclusions. Conclusions are not always pleasant.
— Helen Keller

Traits offer a powerful mechanism for categorization and definition in Scala, both to define an interface for clients to use, *a la* traditional Java interfaces, and as a mechanism for behavioral inheritance based on other behaviors defined within the trait. Perhaps what we need is a new inheritance phrase, *IN-TERMS-OF*, to describe the relationship between a trait and an implementing class.

There are many more ways to use traits than I've described in this article, but part of the goal of the series is to provide enough information about the language to enable further experimentation at home; download the Scala implementation, experiment with it, and see where Scala can plug into your current Java systems. And, as always, if you find Scala useful, if you have a comment on the article, or you (*sigh*) find a bug in the code or prose, [drop me a note](#) and let me know about it.

That's it until next time, functional fans.

Resources

Learn

- [The busy Java developer's guide to Scala](#) (Ted Neward, developerWorks, 2008): Read the complete series.
- ["A Tour of Scala: Mixin Class Composition"](#) (Scala language documentation): Learn more about inheritance in Scala and mix-in class composition.
- ["Thinking in Java: Comparing C++ and Java"](#) (Bruce Eckel, Java Coffee Break): This excerpt from Bruce's book highlights the differences between the Java language and C++.
- ["Dinosaurs Can Take the Pain"](#) (Cay Horstmann, Java.net, January 2008): Invokes the idea that Java programmers are at an evolutionary impasse and proposes some solutions.
- ["What are your Java pain points, really?"](#) (Bill Venners, Artima.com, February 2007): A topic of discussion in the Artima forums, with 264 replies as of this writing.
- [Interoperability happens - Languages](#): Read more of Ted Neward's thoughts about language design and evolution.
- ["Functional programming in the Java language"](#) (Abhijit Belapurkar, developerWorks, July 2004): Understand the benefits and uses of functional programming from a Java developer's perspective.
- ["Scala by Example"](#) (Martin Odersky, December 2007): A short, code-driven introduction to Scala, including the Quicksort application used in this article (PDF).
- [Programming in Scala](#) (Martin Odersky, Lex Spoon, and Bill Venners; Artima preprint published February 2008): The first book-length introduction to Scala, co-authored by Bill Venners.
- The [developerWorks Java technology zone](#): Hundreds of articles about every aspect of Java programming.

Get products and technologies

- [Download Scala](#): Currently in version 2.7.0-final.

Discuss

- [Participate in the discussion forum for this content.](#)
- [developerWorks blogs](#): Get involved in the [developerWorks community](#).

About the author

Ted Neward

Ted Neward is the Principal of Neward & Associates, where he consults, mentors, teaches, and presents on Java, .NET, XML Services, and other platforms. He resides near Seattle, Washington.

Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.