The busy Java developer's guide to Scala: Implementation inheritance

Objects meet functions in Scala's inheritance

Ted Neward

May 28, 2008

Scala gives you just as much support for implementation inheritance as the Java™ language does — but Scala's inheritance yields a few surprises. This month, Ted Neward introduces you to polymorphism done the Scala way, blending functional and object-oriented language styles while still mapping perfectly to the Java platform's inheritance model.

View more content in this series

For the better part of 20 years, a staple of object-oriented language design has been the notion of inheritance. Languages that do not support inheritance, such as Visual Basic, are derided for being "toy languages," unsuited to real work. Meanwhile, languages that do support inheritance do so differently, leading to many hours of debate. Is multiple inheritance really necessary (as the maker of C++ decided), or is it gratuitous and ugly (as determined by the makers of C# and the Java language)? Ruby and Scala are two newer languages that have taken the middle course on multiple inheritance — as I discussed last month when introducing Scala's traits (see Related topics).

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 — and therein lies much of Scala's charm! After all, if Java could do it, why bother learning Scala?

Like all the cool languages, Scala also supports implementation inheritance (see Related topics). In the Java language, a single-implementation-inheritance model allows you to extend base classes and add new methods and fields, and so on. Despite some syntactic changes, Scala's implementation inheritance looks and feels much the same as it does in the Java language. The differences have to do with the ways that Scala fuses object and functional language design, and they're well worth exploring this month.
Plain Old Scala Object

As I've done in previous articles in this series, I'll use the `Person` class as a starting point for exploring Scala's inheritance system. Listing 1 shows the `Person` class definition:

**Listing 1. Hey, I’m a Person**

```scala
// This is Scala
class Person(val firstName:String, val lastName:String, val age:Int)
{
    def toString = "[Person: firstName="+firstName+" lastName="+lastName+
    " age="+age+"]"
}
```

`Person` is a fairly simple POSO (Plain Old Scala Object), with three read-only fields. You might recall that to make them read-write, you need only change the `val` values to `var` variables in the declaration of the primary constructor.

Anyway, using the `Person` type is also pretty trivial, as demonstrated in Listing 2:

**Listing 2. PersonApp**

```scala
// This is Scala
object PersonApp
{
    def main(args : Array[String]) : Unit =
    {
        val bindi = new Person("Tabinda", "Khan", 38)
        System.out.println(bindi)
    }
}
```

Hardly earth-shattering code, but it gives us a starting point.

Abstract methods in Scala

As this system develops, it becomes apparent that the `Person` class lacks a fairly important piece to being a `Person`, which is the act of *doing something*. Many of us define ourselves by what we do with our lives, rather than just existing and taking up space. So, I'll add a new method, shown in Listing 3, which gives `Person` some purpose:

**Listing 3. Well, doSomething!**

```scala
// This is Scala
class Person(val firstName:String, val lastName:String, val age:Int)
{
    override def toString = "[Person: firstName="+firstName+" lastName="+lastName+
    " age="+age+"]"

    def doSomething = // uh.... what?
}
```

Which raises a problem: what, exactly, do `Persons` do? Some `Persons` paint, some sing, some write code, some play video games, and some don’t do much of anything at all (ask any teenager’s
parents about that). So I need to create subclasses of Person, rather than trying to incorporate these activities directly into the Person itself, as shown in Listing 4:

Listing 4. This Person does little

```scala
// This is Scala
class Person(val firstName:String, val lastName:String, val age:Int) {
    override def toString = "[Person: firstName="+firstName+" lastName="+lastName+" age="+age+"]"
    def doSomething = // uh.... what?
}
class Student(firstName:String, lastName:String, age:Int) extends Person(firstName, lastName, age) {
    def doSomething =
    {
        System.out.println("I'm studying hard, Ma, I swear! (Pass the beer, guys!)")
    }
}
```

When I try to compile my code, I discover that it won't compile. This is because the definition for the `Person.doSomething` method doesn't work yet; either the method needs a full body (perhaps throwing an exception to indicate that it should be overridden in a derived class), or else it needs to have no body, similar to how an abstract method works in Java code. I try the abstract route in Listing 5:

Listing 5. abstract class Person

```scala
// This is Scala
abstract class Person(val firstName:String, val lastName:String, val age:Int) {
    override def toString = "[Person: firstName="+firstName+" lastName="+lastName+" age="+age+"]"
    def doSomething; // note the semicolon, which is still optional
        // but stylistically I like having it here
}
class Student(firstName:String, lastName:String, age:Int) extends Person(firstName, lastName, age) {
    def doSomething =
    {
        System.out.println("I'm studying hard, Ma, I swear! (Pass the beer, guys!)")
    }
}
```

Note how I adorned the Person class with the abstract keyword. abstract indicates to the compiler that, yes, this class is supposed to be abstract. In this regard, Scala is no different from the Java language.

Objects, meet functions

Due to Scala's fusing of objects and functional-language styles, I could actually model my Person, as described above, but without creating subtypes. It's a bit of a mind-warp, but it really does
underscore Scala's integration of these two design styles and the very interesting ideas that come of it.

Recall from previous articles that Scala treats functions as values, just as it does any other value in the language, like Int, Float, or double. I can leverage that in this case by modeling my Person to have doSomething not as a method to be overridden in a derived class, but as a function value to be invoked, replaced, and extended. Listing 6 shows this approach:

**Listing 6. A hard-working Person**

```scala
// This is Scala
class Person(val firstName:String, val lastName:String, val age:Int)
{
  var doSomething : (Person) => Unit =
    (p:Person) => System.out.println("I'm " + p + " and I don't do anything yet!");

def work() =
  doSomething(this)

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

object App
{
  def main(args : Array[String]) =
  {
    val bindi = new Person("Tabinda", "Khan", 38)
    System.out.println(bindi)
    bindi.work()

    bindi.doSomething =
      (p:Person) => System.out.println("I edit textbooks")
    bindi.work()

    bindi.doSomething =
      (p:Person) => System.out.println("I write HTML books")
    bindi.work()
  }
}
```

This idea of using functions as first-class modeling tools is quite a common trick in dynamic languages such as Ruby, Groovy, and ECMAScript (aka JavaScript), as well as many functional languages. While it's possible to do in other languages (C++ through pointers-to-functions and/or pointers-to-member-functions, or in Java code through anonymous inner class implementations of an interface reference), it's far more work than what Scala (and Ruby, Groovy, ECMAScript, and others) demands. This is an extension of the "higher order functions" concept that functional programmers toss around. (See Related topics for more about higher order functions.)

Thanks to Scala's view of functions as values, you can employ function values anywhere you might need to switch around functionality at runtime. You might recognize this approach as the Role pattern, a variation on the Gang of Four Strategy pattern where object roles (such as Person's current employment status) are better represented as runtime values than in the static type hierarchy.
Constructors up the hierarchy

Recall, if you will, from your days writing Java code, that sometimes a derived class needs to pass parameters from its constructor up to its base class constructor, in order to allow base class fields to be initialized. In Scala, because the primary constructor appears on the class declaration, instead of as a "traditional" member of the class, passing parameters up to the base takes on a whole new dimension.

In Scala, primary constructor parameters are passed on the class line, but you can also use the val modifier on these parameters in order to easily introduce accessors (and mutators, in the case of var) on the class itself.

So, the Scala class `Person` from Listing 5 turns into the Java class in Listing 7, as viewed by javap:

Listing 7. Translation, please?

```java
// This is javap
C:\Projects\scala-inheritance\code>javap -classpath classes Person
Compiled from "person.scala"
public abstract class Person extends java.lang.Object implements scala.ScalaObject{
    public Person(java.lang.String, java.lang.String, int);
    public java.lang.String toString();
    public abstract void doSomething();
    public int age();
    public java.lang.String lastName();
    public java.lang.String firstName();
    public int $tag();
}
```

The basic rules of the JVM are still at work: derived classes of `Person` have to pass something up to the base class when constructed, regardless of what the language insists. (Actually, this isn't entirely true, but the JVM gets a tad grumpy when languages try to bypass this rule, so most continue to support it in one way or another.) Scala, of course, needs to adhere to this rule, not only because it wants to keep the JVM happy, but also because it wants to keep the Java base classes happy as well. So that means that, somehow, Scala has to enable a syntax allowing derived classes to call up to the base, all the while preserving the syntax that allows us to introduce those accessors and mutators on the base class.

To put this into a more concrete context, let's assume I've written the `Student` class from Listing 5 like so:

Listing 8. Bad Student!

```scala
// This is Scala
// This WILL NOT compile
class Student(val firstName:String, val lastName:String, val age:Int)
    extends Person(firstName, lastName, age)
{
    def doSomething =
    {
        System.out.println("I'm studying hard, Ma, I swear! (Pass the beer, guys!!)"
    }
}
```
The compiler in this case will bark loud and long because I've tried to introduce a new set of methods (firstName, lastName, and age) onto my Student class. Those methods will clash with the similarly named methods on my Person, and the Scala compiler won't necessarily know if I'm trying to override the base class methods (which would be bad because I'd be hiding the implementation and field behind those base class methods), or introduce new methods of the same name (which would be bad because I'd be hiding the implementation and field behind those base class methods). In a bit, you'll see how to successfully override methods from the base class, but that's not what we're after at the moment.

You should also note that in Scala the parameters to Person's constructor don't have to line up one-to-one with the parameters passed to Student's; the rules here are exactly like those of Java's constructors. We do this just for easy reading. Also, Student can demand additional constructor parameters, as it could in the Java language, as shown in Listing 9:

Listing 9. Demanding Student!

```scala
// This is Scala
class Student(firstName:String, lastName:String, age:Int, val subject:String)
  extends Person(firstName, lastName, age)
{
  def doSomething =
  {
    System.out.println("I'm studying hard, Ma, I swear! (Pass the beer, guys!)")
  }
}
```

Yet again, you see how similar Scala code is to Java code, at least when it comes to inheritance and class relationships.

**Differences in syntax**

You may be wondering about the subtleties of the syntax so far. After all, Scala doesn't differentiate fields from methods the way that the Java language does. This is actually a deliberate design decision that allows Scala programmers to "hide" the distinction between fields and methods quite easily from those who use the base class. Consider Listing 10:

Listing 10. What am I?

```scala
// This is Scala
abstract class Person(val firstName:String, val lastName:String, val age:Int)
{
  def doSomething
  def weight : Int
  override def toString = "[Person: firstName="+firstName+" lastName="+lastName+
  " age="+age+"]"
}
class Student(firstName:String, lastName:String, age:Int, val subject:String)
  extends Person(firstName, lastName, age)
{
  def weight : Int =
  age // students are notoriously skinny
  def doSomething =
  {
```
```scala
System.out.println("I'm studying hard, Ma, I swear! (Pass the beer, guys!)")
}

class Employee(firstName:String, lastName:String, age:Int)
  extends Person(firstName, lastName, age)
{
  val weight : Int = age * 4 // Employees are not skinny at all
  def doSomething =
  {
    System.out.println("I'm working hard, hon, I swear! (Pass the beer, guys!)")
  }
}
```

Notice how `weight` is defined to take no parameters and return `Int`? This is a "parameterless method." Because it looks strongly similar to what a "property" method looks like in the Java language, Scala will actually permit the definition of `weight` as either a method (as in `Student`) or as a field/accessor (as in `Employee`). This syntactical decision gives you a degree of flexibility in the implementation of abstract-class derivatives. Note that in Java, you could have the same flexibility only if every field were accessed through its get/set methods, even when being accessed from within the same class. Right or wrong, not many Java programmers write their code this way, so the flexibility isn't often used. What's more, Scala's approach works just as easily with hidden/private members as it does with public ones.

**From @Override to override**

Frequently, a derived class wants to change the behavior of a method defined in one of its base classes; in Java code, we handle this simply by adding a new method of the same name and signature to the derived class. The downside of this approach is the possibility that a typo or slight ambiguity in the signature will silently fail, which means the code will compile but "do the wrong thing" at runtime.

To address this, the Java 5 compiler introduced the `@Override` annotation. `@Override` verifies for `javac` that a method introduced in a derived class has, in fact, overridden a base class method. In Scala, `override` has become part of the language, and forgetting it will generate a compiler error. Thus, a derived `toString()` method should look as shown in Listing 11:

**Listing 11. That's derivative**

```scala
// This is Scala
class Student(firstName:String, lastName:String, age:Int, val subject:String)
  extends Person(firstName, lastName, age)
{
  def weight : Int =
    age // students are notoriously skinny

  def doSomething =
  {
    System.out.println("I'm studying hard, Ma, I swear! (Pass the beer, guys!)")
  }

  override def toString = "[Student: firstName="+firstName+
    lastNam"e=lastName"+age"+age+
    subject=subject"]"
}
```
Pretty straightforward.

**Making it final**

The flip side of permitting derived overrides, of course, is the act of preventing it: Sometimes, a base class wants to forbid a child class from changing its base-class behavior, or from even having any sort of derived class whatsoever. In the Java language, we do this by applying the modifier `final` to the method, ensuring that it will not be overridden. Or, we might apply `final` to the class as a whole to prevent derivation. Implementation hierarchy works the same way in Scala: We can apply `final` to the method to prevent a child class from overriding it or to the class declaration itself to prevent derivatives.

Bear in mind that all of this discussion about `abstract` and `final` and `override` applies equally to "methods with funny names" (what Java or C# or C++ programmers would call operators) as it does to routinely named methods. So it's common to define a base class or trait that sets a certain expectation for mathematical functionality (call it "Mathable," if you will) that defines abstract member functions "+", ",", "*" and "/", along with any other mathematical operations that should be supported, such as `pow` or `abs`. Then other programmers can create additional types — perhaps a `Matrix` class — that can implement or extend "Mathable," define those members, and look like any other built-in arithmetic type Scala provides "out of the box."

**The difference is in the ...**

If Scala maps to the Java inheritance model so easily, as you've seen so far, it should be possible to have Scala classes inherit from the Java language, and vice versa. In fact, it absolutely has to be possible because Scala, like any language that compiles to Java bytecode, has to produce objects that inherit from `java.lang.Object`. Note that Scala classes might also inherit from other things, too, such as traits, so how the actual inheritance resolution and code-generation works can be different, but in the end, we have to be able to inherit from Java base classes in some fashion. (Remember, traits are something like interfaces with behavior, and the Scala compiler gets this to work by splitting the trait into an interface and dropping the implementation into the class the trait is compiled into.)

As it turns out, however, Scala's type hierarchy is slightly and subtly different from that of the Java language; technically, the base class from which all Scala classes inherit, including types like `Int`, `Float`, `Double`, and the other numeric types, is the `scala.Any` type, which defines a core set of methods available on any type in Scala: `==`, `!=`, `equals`, `hashCode`, `toString`, `isInstanceOf`, and `asInstanceOf`, most of which are pretty easily understood by their names alone. From there, Scala splits into two major branches, where the "primitive types" inherit from `scala.AnyVal`, and the "class types" inherit from `scala.AnyRef`. (`scala.ScalaObject` in turn inherits from `scala.AnyRef`.)

Normally, this isn't something you need to worry about directly, but it can have some rare interesting side-effects when considering inheritance across the two languages. For example, consider the `ScalaJavaPerson` in Listing 12:
Listing 12. It's a hybrid!

```scala
// This is Scala
class ScalaJavaPerson(firstName: String, lastName: String, age: Int)
  extends JavaPerson(firstName, lastName, age)
{
  val weight : Int = age * 2 // Who knows what Scala/Java people weigh?
  override def toString = 
    "[SJPerson: firstName=" + firstName + " lastName=" + lastName + " age=" + age + "]"
}
```

... which inherits from this `JavaPerson`:

Listing 13. Look familiar?

```java
// This is Java
public class JavaPerson {
    public JavaPerson(String firstName, String lastName, int age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    public String getFirstName() {
        return this.firstName;
    }

    public void setFirstName(String value) {
        this.firstName = value;
    }

    public String getLastName() {
        return this.lastName;
    }

    public void setLastName(String value) {
        this.lastName = value;
    }

    public int getAge() {
        return this.age;
    }

    public void setAge(int value) {
        this.age = value;
    }

    public String toString() {
        return "[Person: firstName=" + firstName + ", lastName=" + lastName + ", age=" + age + "]";
    }

    private String firstName;
    private String lastName;
    private int age;
}
```
When ScalaJavaPerson compiles, it will extend JavaPerson as normal, but again, as demanded by Scala, it will also implement the scalaObject interface. It will also support the methods inherited from JavaPerson as usual. Note that because ScalaJavaPerson is a Scala type, we can expect it to support assignment into an Any reference, as per Scala's rules:

**Listing 14. Using ScalaJavaPerson**

```scala
// This is Scala
val richard = new ScalaJavaPerson("Richard", "Campbell", 45)
System.out.println(richard)
val host : Any = richard
System.out.println(host)
```

But what happens when I create a JavaPerson in Scala and attempt to assign it to an Any reference as well?

**Listing 15. Using JavaPerson**

```scala
// This is Scala
val carl = new JavaPerson("Carl", "Franklin", 35)
System.out.println(carl)
val host2 : Any = carl
System.out.println(host2)
```

As it turns out, this code compiles and works as expected because Scala can silently ensure that the JavaPerson "does the right thing," thanks to the similarities of the Any type to the java.lang.Object type. In fact, it's almost fair to say that anything that extends java.lang.Object also supports being stored into an Any reference. (There are a few edge cases, I'm told, but I've never run into any myself thus far.)

Net result? For all practical purposes, we can mix and match inheritance across both the Java language and Scala without too much concern. (The big headache will be trying to figure out how to override a Scala "method with a funny name" like "^=!#" or something similar.)

**In conclusion**

As I've shown you this month, the close fidelity between Scala code and Java code means that Scala's inheritance model is easy for Java developers to pick up and understand. Method overriding works the same, member visibility works the same, and so on. Of all the functionality in Scala, inheritance is probably the most similar to what you've seen in your own Java development. The only tricky part is Scala syntax, which is markedly different.

Being comfortable with the overlap (and slight differences) in how the two languages approach inheritance means you can begin to comfortably write your own Scala implementations of Java programs. For example, consider Scala implementations of popular Java base classes and frameworks like JUnit, Servlets, Swing, or SWT. In fact, the Scala team produced a Swing application, called OOPScala (see Related topics), that used JTable to provide simple spreadsheet functionality in a ridiculously small number of lines of code (easily an order of magnitude less than a traditional Java equivalent).
So, if you've been wondering how Scala applies to your production code, you should now be ready to take your first steps to finding out. Just think about writing certain pieces of your next program in Scala. As you've learned this month, you'll have no trouble inheriting from the appropriate base classes and providing overrides just the way you would in your Java programs.
Related topics

- "Scala for Java refugees Part 5: Traits and types" (Daniel Spiewak, Code Commit, February 2008): Another pass at the topic of inheritance in Scala.
- "Implementation Inheritance with Mixins - Some Thoughts" (Debasish Ghosh, Ruminations of a Programmer, February 2008): Discusses the "workable compromises" that define Java language inheritance to date.
- "A Tour of Scala: Higher-Order Functions" (Scala-lang.org): A short example of a higher order function in Scala.
- "OOPScala: A Swing application written in Scala.
- "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, May 2008): 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 pre-print published February 2008): The first book-length introduction to Scala, co-authored by Bill Venners.
- **Download Scala**: Currently in version 2.7.0-final.