

Extracted from:

Programming Scala

Tackle Multi-Core Complexity on the JVM

This PDF file contains pages extracted from Programming Scala, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2009 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The
Pragmatic
Programmers

Programming Scala

Tackle Multi-Core Complexity
on the Java Virtual Machine



Venkat Subramaniam

Edited by Daniel H Steinberg



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

<http://www.pragprog.com>

Copyright © 2008 Venkat Subramaniam.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.

ISBN-10: 1-934356-31-X

ISBN-13: 978-1-934356-31-9

Printed on acid-free paper.

P1.0 printing, June 2009

Version: 2009-7-7

Traits and Type Conversions

Traits are like interfaces with a partial implementation. Traits provide a middle ground between single and multiple inheritance because you can mix them in or include them in other classes. This allows you to enhance a class with a set of features.

Single implementation inheritance forces you to model everything into a linear hierarchy. However, the real world is full of crosscutting concerns—concepts that cut across and affect abstractions that do not fall under the same class hierarchy. Security, logging, validation, transactions, resource allocation, and management are all examples of such crosscutting concerns in a typical enterprise application. Scala’s traits allow you to apply those concerns to arbitrary classes without the pain that arises from multiple implementation inheritance.

In this chapter, you’ll learn Scala’s support for abstraction and object models. Much of this will feel like magic. Scala’s implicit conversion allows you to treat an instance of one class as an instance of another. This allows you to attach methods to an object without modifying the original class, by implicitly wrapping the instance in a façade. You’ll use that trick to see how to create a DSL.

7.1 Traits

A *trait* is a behavior that can be mixed into or assimilated into a class hierarchy. Say we want to model a Friend. We can mix that into any class, Man, Woman, Dog, and so on, without having to inherit them all from a common base class.

Assume we've modeled a class `Human` and want to make it friendly. A friend is someone who listens. So, here is the `listen` method that we'd add to the `Human` class:

```
class Human(val name: String) {
  def listen() = println("Your friend " + name + " is listening")
}
```

```
class Man(override val name: String) extends Human(name)
class Woman(override val name: String) extends Human(name)
```

One disadvantage of the previous code is the friendly quality does not quite stand out and is merged into the `Human` class. Furthermore, a few weeks into development, we realize we forgot man's best friend. Dogs are great friends—they listen to us quietly when we have a lot to unload. But, how can we make a `Dog` a friend? We can't inherit a `Dog` from a `Human` for that purpose. The Java approach to solving this problem would be to create an interface `Friend` and have `Human` and `Dog` implement it. We're forced to provide different implementations in these two classes irrespective of whether the implementations are different.

This is where Scala's traits come in. A *trait* is like an interface with a partial implementation. The `vals` and `vars` you define and initialize in a trait get internally implemented in the classes that mix the trait in. Any `vals` and `vars` defined but not initialized are considered abstract, and the classes that mix in these traits are required to implement them. We can reimplement the `Friend` concept as a trait:

[Download](#) TraitsAndTypeConversions/Friend.scala

```
trait Friend {
  val name: String
  def listen() = println("Your friend " + name + " is listening")
}
```

Here we have defined `Friend` as a trait. It has a `val` named `name` that is treated as abstract. We also have the implementation of a `listen()` method. The actual definition or the implementation of `name` will be provided by the class that mixes in this trait. Let's look at ways to mix in the previous trait:

[Download](#) TraitsAndTypeConversions/Human.scala

```
class Human(val name: String) extends Friend
```

[Download](#) TraitsAndTypeConversions/Man.scala

```
class Man(override val name: String) extends Human(name)
```

[Download](#) TraitsAndTypeConversions/Woman.scala

```
class Woman(override val name: String) extends Human(name)
```

The class Human mixes in the Friend trait. If a class does not extend from any other class, then use the extends keyword to mix in the trait. The class Human and its derived classes Man and Woman simply use the implementation of the listen() method provided in the trait. We can override this implementation if we like, as we'll see soon.

You can mix in any number of traits. To mix in additional traits, use the keyword with. You will also use the keyword with to mix in your first trait if your class already extends from another class like the Dog in this next example. In addition to mixing in the trait, we have overridden its listen() method in Dog.

[Download](#) TraitsAndTypeConversions/Animal.scala

```
class Animal
```

[Download](#) TraitsAndTypeConversions/Dog.scala

```
class Dog(val name: String) extends Animal with Friend {
  //optionally override method here.
  override def listen = println(name + "'s listening quietly")
}
```

You can call the methods of a trait on the instances of classes that mix it in. You can also treat a reference to such classes as a reference of the trait:

[Download](#) TraitsAndTypeConversions/UseFriend.scala

```
val john = new Man("John")
val sara = new Woman("Sara")
val comet = new Dog("Comet")
```

```
john.listen
sara.listen
comet.listen
```

```
val mansBestFriend : Friend = comet
mansBestFriend.listen
```

```
def helpAsFriend(friend: Friend) = friend.listen
```

```
helpAsFriend(sara)
helpAsFriend(comet)
```

The output from the previous code is shown here:

```
Your friend John is listening
Your friend Sara is listening
```

```
Comet's listening quietly
Comet's listening quietly
Your friend Sara is listening
Comet's listening quietly
```

Traits look similar to classes but have some significant differences. First, they require the mixed-in class to implement the uninitialized (abstract) variables and values declared in them. Second, their constructors cannot take any parameters. Traits are compiled into Java interfaces with corresponding implementation classes that hold any methods implemented in the traits.

Traits do not suffer from the method collision problem that generally arise from multiple inheritance. They avoid it by late binding with the method of the class that mixes them in. So, a call to `super` within a trait resolves to a method on another trait or the class that mixes it in, as you'll see soon.

7.2 Selective Mixins

In the previous example, we mixed the trait `Friend` into the `Dog` class. This allows us to treat *any* instance of the `Dog` class as a `Friend`; that is, all `Dogs` are `Friends`.

You can also mix in traits selectively at an instance level. This will allow you to treat a specific instance of a class as a trait. Let's look at an example:

[Download](#) TraitsAndTypeConversions/Cat.scala

```
class Cat(val name: String) extends Animal
```

`Cat` does not mix in the `Friend` trait, so we can't treat an instance of `Cat` as a `Friend`. Any attempts to do so, as you can see here, will result in compilation errors:

[Download](#) TraitsAndTypeConversions/UseCat.scala

```
def useFriend(friend: Friend) = friend listen
```

```
val alf = new Cat("Alf")
val friend : Friend = alf // ERROR

useFriend(alf) // ERROR
```

Here you can see the errors:

```
(fragment of UseCat.scala):4: error: type mismatch;
 found   : Cat
 required: Friend
val friend : Friend = alf // ERROR
      ^
(fragment of UseCat.scala):6: error: type mismatch;
 found   : Cat
 required: Friend
useFriend(alf) // ERROR
      ^
two errors found
!!!
discarding <script preamble>
!!!
discarding <script preamble>
```

Scala, however, does offer help for cat lovers, and we can exclusively treat our special pet as a `Friend` if we want. When creating an instance, simply mark it using the `with` keyword:

[Download](#) TraitsAndTypeConversions/TreatCatAsFriend.scala

```
def useFriend(friend: Friend) = friend listen

val snowy = new Cat("Snowy") with Friend
val friend : Friend = snowy
friend.listen

useFriend(snowy)
```

Here's the output:

```
Your friend Snowy is listening
Your friend Snowy is listening
```

Scala gives you the flexibility to treat all the instances of a class as a trait or to select only the instances you want. The latter is especially useful if you want to apply traits to preexisting classes.

7.3 Decorating with Traits

You can use traits to decorate¹ objects with capabilities. Assume we want to run different checks on an applicant—credit, criminal records,

1. See the Decorator pattern in Gamma et al.'s *Design Patterns: Elements of Reusable Object-Oriented Software* [GHJV95].

employment, and so on. We're not interested in all the checks all the time. An applicant for an apartment may need to be checked for credit and criminal records. On the other hand, an applicant for employment may need to be checked for criminal records and previous employment. If we resort to creating specific classes for these groups of checks, we'll end up creating several classes for each permutation of checks we needed. Furthermore, if we decide to run additional checks, the class handling that group of checks would have to change. No, we want to avoid such class proliferation. We can be productive and mix in only specific checks required for each situation.

Next we'll introduce an abstract class `Check` that runs a general check on the application details:

[Download](#) TraitsAndTypeConversions/Decorator.scala

```
abstract class Check {
  def check() : String = "Checked Application Details..."
}
```

For different types of checks like credit, criminal record, and employment, we create traits like these:

[Download](#) TraitsAndTypeConversions/Decorator.scala

```
trait CreditCheck extends Check {
  override def check() : String = "Checked Credit..." + super.check()
}

trait EmploymentCheck extends Check {
  override def check() : String = "Checked Employment..." + super.check()
}

trait CriminalRecordCheck extends Check {
  override def check() : String = "Check Criminal Records..." + super.check()
}
```

We've extended these traits from the class `Check` since we intend to mix them into only those classes that extend from `Check`. Extending the class gives us two capabilities. One, these traits can be mixed in only with classes that extend `Check`. Second, we can use the methods of `Check` within these traits.

We are interested in enhancing or decorating the implementation of the method `check()`, so we have to mark it as `override`. In our implementation of `check()`, we invoke `super.check()`. Within a trait, calls to method using `super` go through late binding. This is not a call on the base class but instead on the trait mixed in to the left—if this is the leftmost trait

mixed in, the call resolves to the method on the class into which we mixed in the trait(s). We'll see this behavior when we complete this example.

So, we have one abstract class and three traits in the example so far. We don't have any concrete classes—we don't need any. If we want to run checks for an apartment application, we can put together an instance from the previous traits and class:

[Download](#) TraitsAndTypeConversions/Decorator.scala

```
val apartmentApplication = new Check with CreditCheck with CriminalRecordCheck
println(apartmentApplication check)
```

On the other hand, we could run checks for employment like this:

[Download](#) TraitsAndTypeConversions/Decorator.scala

```
val emplomentApplication = new Check with CriminalRecordCheck with EmploymentCheck
println(emplomentApplication check)
```

If you'd rather run a different combination of checks, simply mix in the traits the way you like. The effect of previous two pieces of code is shown here:

```
Check Criminal Records...Checked Credit...Checked Application Details...
Checked Employment...Check Criminal Records...Checked Application Details...
```

The rightmost trait picked up the call to `check()`. It then, upon the call to `super.check()`, passed the call over to the trait on its left. The leftmost traits invoked the `check()` on the actual instance.

Traits are a powerful tool in Scala that allow you to mix in crosscutting concerns, and you can use them to create highly extensible code with low ceremony. Rather than creating a hierarchy of classes and interfaces, you can put your essential code to quick use.

7.4 Method Late Binding in Traits

In the previous example, the method `check()` of the `Check` class was concrete. Our traits extended from this class. We saw how the call to `super.check()` within the traits were bound to either the trait on the left or the class that mixes in. Things get a bit more complicated if the method(s) in the base class are abstract. Let's explore this further here.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style, and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Programming Scala's Home Page

<http://pragprog.com/titles/vsscala>

Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates

<http://pragprog.com/updates>

Be notified when updates and new books become available.

Join the Community

<http://pragprog.com/community>

Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy

<http://pragprog.com/news>

Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/vsscala.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragprog.com/catalog
Customer Service:	orders@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com