Extracted from:

Programming Scala

Tackle Multi-Core Complexity on the JVM

The Pragmatic Programmers

Programming Scala

Tackle Multi-Core Complexity on the Java Virtual Machine

Venkat Subramaniam

Edited by Daniel H Steinberg

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

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

Chapter 5

Sensible Typing

Static typing, or compile-time type checking, helps you define and verify interface contracts at compile time. Scala, unlike some of the other statically typed languages, does not expect you to provide redundant type information. You don't have to specify a type in most cases, and you certainly don't have to repeat it. At the same time, Scala will infer the type and verify proper usage of references at compile time. Let's explore this with an example:

Download SensibleTyping/Typing.scala

```
var year: Int = 2009
var anotherYear = 2009
var greet = "Hello there"
var builder = new StringBuilder("hello")

println(builder.getClass())
```

Here we defined a variable year explicitly as type Int. We also defined anotherYear as a variable but let Scala infer the type as Int based on what we assigned to that variable. Similarly, we let Scala infer the type of greet as String and builder as StringBuilder. We can query the reference builder to find what type it's referring to. If you attempt to assign some other type of value or instance to any of these variables, you'll get a compilation error. Scala's type inference is low ceremony[1] and has no learning curve; you simply have to undo some Java practices.

Scala's static typing helps you in two ways. First, the compile-time type checking can give you confidence that the compiled code *meets* certain

1. See "Essence vs. Ceremony" in Appendix A, on page 213.

expectations.[2] Second, it helps you to *express* the expectations on your API in a compiler-verifiable format.

In this chapter, you'll learn about Scala's sensible static typing and type inference. You'll also look at three special types in Scala: Any, Nothing, and Option.

5.1 Collections and Type Inference

Scala will provide type inference and type safety for the Java Generics collections as well. The following is an example that uses an ArrayList. The first declaration uses explicit, but redundant, typing. The second declaration takes advantage of type inference.

As an aside, note that the underscore in the import statement is equivalent to the asterisks in Java. So when we type java.util._, we are importing all classes in the java.util package. If the underscore follows a class name instead of a package name, we are importing all members of the class—the equivalent of Java static import:

Download SensibleTyping/Generics.scala

```
import java.util._

var list1 : List[Int] = new ArrayList[Int]
var list2 = new ArrayList[Int]

list2 add 1
list2 add 2

var total = 0
for (val index <- 0 until list2.size()) {
  total += list2.get(index)
}

println("Total is " + total)
```

Here's the output:

```
Total is 3
```

2. As you'll see, this is not a substitute for good unit testing, but you can use the good compiler support as a first level of defense.

Scala is vigilant about the type of the object you instantiate. It prohibits conversions that may cause typing issues.[3] Here's an example of how Scala differs from Java when it comes to handling Generics:

Download SensibleTyping/Generics2.scala

```
import java.util._

var list1 = new ArrayList[Int]
var list2 = new ArrayList

list2 = list1 // Compilation Error
```

We created a reference, list1, that points to an instance of ArrayList[Int]. Then we created another reference, list2, that points to an instance of ArrayList with an unspecified parametric type. Behind the scenes, Scala actually created an instance of ArrayList[Nothing]. When we try to assign the first reference to the second, Scala gives us this compilation error.[4]

```
(fragment of Generics2.scala):6: error: type mismatch;
 found   : java.util.ArrayList[Int]
 required: java.util.ArrayList[Nothing]
list2 = list1 // Compilation Error
        ^
one error found
!!!
discarding <script preamble>
```

Nothing is a subclass of all classes in Scala. By treating the new ArrayList as an instance of ArrayList[Nothing], Scala rules out any possibility of adding an instance of any meaningful type to this collection. This is because you can't treat an instance of base as an instance of derived and Nothing is the bottom-most subclass.

So, how can you create a new ArrayList without specifying the type? One way is to use the type Any. You saw how Scala deals with an assignment when one collection holds objects of type Nothing, while the other does not. Scala, by default, insists the collection types on either side of assignment are the same (you'll see later in Section 5.7, *Variance of Parameterized Type*, on page 73 how you can alter this default behavior in Scala).

3. Of course, Scala has no control over conversions that happen in compiled Java or other language code that you call.
4. Equivalent Java code will compile with no errors but result in a runtime ClassCastException.

Here is an example using a collection of objects of type Any—Any is the base type of all types in Scala:

Download SensibleTyping/Generics3.scala

```
import java.util._

var list1 = new ArrayList[Int]
var list2 = new ArrayList[Any]

var ref1 : Int = 1
var ref2 : Any = null

ref2 = ref1 //OK

list2 = list1 // Compilation Error
```

This time list1 refers to an ArrayList[Int], while list2 refers to an ArrayList[Any]. We also created two other references, ref1 of type Int and ref2 of type Any. Scala has no qualms about letting us assign ref1 to ref2. So, it is equivalent to assigning an Integer reference to a reference of type Object. However, Scala doesn't allow, by default, assigning a collection of arbitrary type instances to a reference of a collection of Any instances (later we'll discuss covariance, which provides exceptions to this rule). You saw how Java Generics enjoy enhanced type safety in Scala.

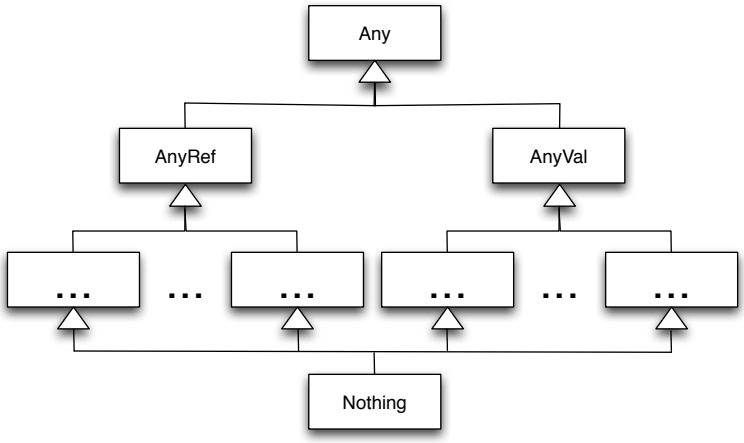
You don't have to specify the type in order to benefit from Scala type checking. You can just rely on the type inference where it makes sense. The inference happens at compile time. So, you can be certain that the type checking takes effect right then when you compile the code.

Scala insists that a nonparameterized collection be a collection of Nothing and restricts assignment between types. These combine to enhance type safety at compile time—providing for a sensible, low-ceremony static typing.

In the previous examples, we used the Java collections. Scala also provides a wealth of collections, as you'll see in Chapter 8, *Using Collections*, on page 105.

5.2 The Any Type

Scala's Any type is a superclass of all types in Scala, graphically illustrated in the following diagram.



Any allows you to hold a common reference to objects of any type in Scala. Any is an abstract class with the following methods: !=(), ==(), asInstanceOf(), equals(), hashCode(), isInstanceOf(), and toString().

The direct descendants of Any are AnyVal and AnyRef. AnyVal serves as a base for all types in Scala that map over to the primitive types in Java—for example, Int, Double, and so on. On the other hand, AnyRef is the base for all reference types. Although AnyVal does not have any additional methods, AnyRef contains the methods of Java's Object such as notify(), wait(), and finalize().

AnyRef directly maps to the Java Object, so you can pretty much use it in Scala like you'd use Object in Java. On the other hand, you can't call all the methods of Object on a reference of Any or AnyVal, even though internally Scala treats them as Object references when compiled to bytecode. In other words, while AnyRef directly maps to Object, Any and AnyVal are type erased to Object much like type erasure of Generics parameterized types in Java.

5.3 More About Nothing

You can see why you'd need Any, but what is the purpose of Nothing?

Scala's type inference works hard to determine the type of expressions and functions. If the type inferred is too broad, it will not help type verification. At the same time, how do you infer the type of an expression or function if one branch returns, say, an Int and another branch throws an exception? In this case, it is more useful to infer the type as Int rather than a general Any. This means that the branch that throws

the exception must be inferred to return an Int or a subtype of Int for it to be compatible. However, an exception may occur at any place, so not all those expressions can be inferred as Int. Scala helps type inference work smoothly with the type Nothing, which is a subtype of all types. Since it is a subtype of all types, it is substitutable for anything. Nothing is abstract, so you would not have a real instance of it anywhere at runtime. It is purely a helper for type inference purposes.

Let's explore this further with an example. Let's take a look at a method that throws an exception and see how Scala infers the type:

```
def madMethod() = { throw new IllegalArgumentException() }

println(getClass().getDeclaredMethod("madMethod", null).
  getReturnType().getName())
```

The method madMethod() simply throws an exception. Using reflection, we're querying the return type of this method with this result:[5]

```
scala.runtime.Nothing$
```

Scala infers the return type of an expression that throws an exception as Nothing. Scala's Nothing is actually quite something—it is a subtype of every other type. So, Nothing is substitutable for anything in Scala.

5.4 Option Type

Scala goes a step further in specifying nonexistence. When you perform pattern matching, for example, the result of the match may be an object, a list, a tuple, and so on, or it may be nonexistent. Returning a null quietly is problematic in two ways. First, the intent that you actually expect nonexistence of a result is not expressed explicitly. Second, there is no way to force the caller of your function to check for nonexistence (null). Scala wants you to clearly specify your intent that sometimes you do actually expect to give no result. Scala achieves this in a type-safe manner using the Option[T] type. Let's look at an example:

Download SensibleTyping/OptionExample.scala

```
def commentOnPractice(input: String) = {
  //rather than returning null
  if (input == "test") Some("good") else None
}
```

5. The $ symbol indicates an internal representation in Scala.

```
for (input <- Set("test", "hack")) {
  val comment = commentOnPractice(input)
  println("input " + input + " comment " +
    comment.getOrElse("Found no comments"))
}
```

Here, commentOnPractice() may return a comment (String) or may not have any comments at all. This is represented as instances of Some[T] and None, respectively. These two classes extend from the Option[T] class. The output from the previous code is as follows:

```
input test comment good
input hack comment Found no comments
```

By making the type explicit as Option[String], Scala forces us to check for the nonexistence of an instance. You're less likely to get NullPointerException because of unchecked null references. By calling the getOrElse() method on the returned Option[T], you can proactively indicate what to do in case the result is nonexistent (None).

5.5 Method Return Type Inference

In addition to inferring the types of variables, Scala also tries to infer the return type of methods. However, there is a catch. It depends on how you define your method. If you define your method with an equals sign (=), then Scala infers the return type. Otherwise, it assumes the method is a void method. Let's look at an example:

Download SensibleTyping/Methods.scala

```
def printMethodInfo(methodName: String) {
  println("The return type of " + methodName + " is " +
    getClass().getDeclaredMethod(methodName, null).getReturnType().getName())
}

def method1() { 6 }
def method2() = { 6 }
def method3() = 6
def method4 : Double = 6

printMethodInfo("method1")
printMethodInfo("method2")
printMethodInfo("method3")
printMethodInfo("method4")
```

We've defined method1() like we normally define methods, by providing a method name, a parameter list within parentheses, and the method body within curly braces. Unfortunately, the way we're used to is not

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