

# The busy Java developer's guide to Scala: Enhancing the Scitter library

Skill Level: Intermediate

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

02 Jun 2009

Scala is fun to talk about in the abstract, but using it in a practical way makes the difference between seeing it as a "toy" and using it on the job. In this follow-up article to his [introduction to Scitter](#), a Scala client library for accessing Twitter, Scala enthusiast Ted Neward offers a more interesting and useful set of features for the client library.

Welcome back, Scala fans. [Last month](#), we talked about Twitter, the micro-blogging site currently enjoying tremendous interest among social-networking types, and how its XML-/REST-based API makes it an interesting playground for developers to investigate and explore. Toward that end, we began to flesh out the basic structure of "Scitter," a Scala library for accessing Twitter.

## About this series

Ted Neward dives into the Scala programming language and takes you along with him. In this 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?

We have several goals for Scitter:

- To make it substantially easier to access Twitter than just opening an HTTP connection and doing the work "by hand"

- To be as easily accessible to Java clients
- To make it easy to mock out for testing

We won't necessarily flesh out the fullness of the entire Twitter API in this installment, but we'll get some of the core bits in place, with an eye toward making it easy for others to finish the job once this library hits a public source control repository.

## The story so far: Scitter 0.1

Let's start with a quick reminder of where we left off:

### Listing 1. Scitter v0.1

```
package com.tedneward.scitter
{
  import org.apache.commons.httpclient._, auth._, methods._, params._
  import scala.xml._

  /**
   * Status message type. This will typically be the most common message type
   * sent back from Twitter (usually in some kind of collection form). Note
   * that all optional elements in the Status type are represented by the
   * Scala Option[T] type, since that's what it's there for.
   */
  abstract class Status
  {
    /**
     * Nested User type. This could be combined with the top-level User type,
     * if we decide later that it's OK for this to have a boatload of optional
     * elements, including the most-recently-posted status update (which is a
     * tad circular).
     */
    abstract class User
    {
      val id : Long
      val name : String
      val screenName : String
      val description : String
      val location : String
      val profileImageUrl : String
      val url : String
      val protectedUpdates : Boolean
      val followersCount : Int
    }
    /**
     * Object wrapper for transforming (format) into User instances.
     */
    object User
    {
      /**
       * def fromAtom(node : Node) : Status =
       * {
       *
       * }
       */
      /**
       * def fromRss(node : Node) : Status =
```

```

    {
    }
    */
    def fromXml(node : Node) : User =
    {
        new User {
            val id = (node \ "id").text.toLong
            val name = (node \ "name").text
            val screenName = (node \ "screen_name").text
            val description = (node \ "description").text
            val location = (node \ "location").text
            val profileImageUrl = (node \ "profile_image_url").text
            val url = (node \ "url").text
            val protectedUpdates = (node \ "protected").text.toBoolean
            val followersCount = (node \ "followers_count").text.toInt
        }
    }

    val createdAt : String
    val id : Long
    val text : String
    val source : String
    val truncated : Boolean
    val inReplyToStatusId : Option[Long]
    val inReplyToUserId : Option[Long]
    val favorited : Boolean
    val user : User
}
/**
 * Object wrapper for transforming (format) into Status instances.
 */
object Status
{
    /*
    def fromAtom(node : Node) : Status =
    {
    }
    */
    /*
    def fromRss(node : Node) : Status =
    {
    }
    */
    def fromXml(node : Node) : Status =
    {
        new Status {
            val createdAt = (node \ "created_at").text
            val id = (node \ "id").text.toLong
            val text = (node \ "text").text
            val source = (node \ "source").text
            val truncated = (node \ "truncated").text.toBoolean
            val inReplyToStatusId =
                if ((node \ "in_reply_to_status_id").text != "")
                    Some((node \ "in_reply_to_status_id").text.toLong)
                else
                    None
            val inReplyToUserId =
                if ((node \ "in_reply_to_user_id").text != "")
                    Some((node \ "in_reply_to_user_id").text.toLong)
                else
                    None
            val favorited = (node \ "favorited").text.toBoolean
            val user = User.fromXml((node \ "user")(0))
        }
    }
}

```

```

}

/**
 * Object for consuming "non-specific" Twitter feeds, such as the public timeline.
 * Use this to do non-authenticated requests of Twitter feeds.
 */
object Scitter
{
  /**
   * Ping the server to see if it's up and running.
   *
   * Twitter docs say:
   * test
   * Returns the string "ok" in the requested format with a 200 OK HTTP status code.
   * URL: http://twitter.com/help/test.format
   * Formats: xml, json
   * Method(s): GET
   */
  def test : Boolean =
  {
    val client = new HttpClient()

    val method = new GetMethod("http://twitter.com/help/test.xml")

    method.getParams().setParameter(HttpMethodParams.RETRY_HANDLER,
      new DefaultHttpClientRetryHandler(3, false))

    client.executeMethod(method)

    val statusLine = method.getStatusLine()
    statusLine.getStatusCode() == 200
  }
  /**
   * Query the public timeline for the most recent statuses.
   *
   * Twitter docs say:
   * public_timeline
   * Returns the 20 most recent statuses from non-protected users who have set
   * a custom user icon. Does not require authentication. Note that the
   * public timeline is cached for 60 seconds so requesting it more often than
   * that is a waste of resources.
   * URL: http://twitter.com/statuses/public_timeline.format
   * Formats: xml, json, rss, atom
   * Method(s): GET
   * API limit: Not applicable
   * Returns: list of status elements
   */
  def publicTimeline : List[Status] =
  {
    import scala.collection.mutable.ListBuffer

    val client = new HttpClient()

    val method = new GetMethod("http://twitter.com/statuses/public_timeline.xml")

    method.getParams().setParameter(HttpMethodParams.RETRY_HANDLER,
      new DefaultHttpClientRetryHandler(3, false))

    client.executeMethod(method)

    val statusLine = method.getStatusLine()
    if (statusLine.getStatusCode() == 200)
    {
      val responseXML =
        XML.loadString(method.getResponseBodyAsString())

      val statusListBuffer = new ListBuffer[Status]
    }
  }
}

```

```

        for (n <- (responseXML \\ "status").elements)
            statusListBuffer += (Status.fromXml(n))

        statusListBuffer.toList
    }
    else
    {
        Nil
    }
}
}
}
/**
 * Class for consuming "authenticated user" Twitter APIs. Each instance is
 * thus "tied" to a particular authenticated user on Twitter, and will
 * behave accordingly (according to the Twitter API documentation).
 */
class Scitter(username : String, password : String)
{
    /**
     * Verify the user credentials against Twitter.
     *
     * Twitter docs say:
     * verify_credentials
     * Returns an HTTP 200 OK response code and a representation of the
     * requesting user if authentication was successful; returns a 401 status
     * code and an error message if not. Use this method to test if supplied
     * user credentials are valid.
     * URL: http://twitter.com/account/verify_credentials.format
     * Formats: xml, json
     * Method(s): GET
     */
    def verifyCredentials : Boolean =
    {
        val client = new HttpClient()

        val method = new GetMethod("http://twitter.com/help/test.xml")

        method.getParams().setParameter(HttpMethodParams.RETRY_HANDLER,
            new DefaultHttpClientRetryHandler(3, false))

        client.getParams().setAuthenticationPreemptive(true)
        val creds = new UsernamePasswordCredentials(username, password)
        client.getState().setCredentials(
            new AuthScope("twitter.com", 80, AuthScope.ANY_REALM), creds)

        client.executeMethod(method)

        val statusLine = method.getStatusLine()
        statusLine.getStatusCode() == 200
    }
}
}
}

```

It's a bit lengthy, but pretty easy to distill into a couple of basic components:

- Case classes `User` and `Status` represent the basic types that Twitter sends back to its clients in response to an API call, complete with methods to construct into and extract from XML representations.
- A `Scitter` singleton object handles those operations that don't require an authenticated user.
- A `Scitter` instance (parameterized by username and password) is used for

operations that require an authenticated user.

### About the Twitter API

If you're not familiar with the Twitter API, it's worth taking a few minutes to go check out the [Twitter API Wiki page](#) for more details. The basic fundamentals are simple — parameters are passed as part of the URL query, the responses can be in one of four formats (JSON, XML, ATOM, or RSS), and so on — but as with all APIs, the devil is in the details and assuming readers have the Twitter API open in a browser somewhere while they read this article makes it easier to focus on the Scala parts of the discussion.

So far within the two Scitter types, we have only the test, verifyCredentials, and public\_timeline APIs covered. While those help verify that the basics of HTTP access (using the Apache HttpClient library) works and that our basic form of converting the XML responses into Status objects works, right now we can't even do the basic "what are my friends saying" public timeline query, nor have we taken even the basic steps towards preventing "repeat yourself" kinds of problems within the codebase, much less started looking at ways to make it easier to mock out the network access code for testing.

Clearly we've got a long way to go in this episode.

## Connecting

The first thing that bugs me about the code is that I'm repeating the sequence of operations to create an HttpClient instance, initialize it, parameterize it with the necessary authentication parameters, and so on, in every one of the methods of both the Scitter object and class. When there are only three methods between the two of them, it might be manageable, but it's clearly not going to scale, and there are a lot of methods left to do. Plus it's going to be really difficult to go back into those methods later and introduce some kind of mocking and/or local/offline-testing capability. So let's fix that.

This really isn't a Scala-ism, per se, that we're introducing here; it's just good plain Don't-Repeat-Yourself kind of thinking. As such, I'm going to start with a basic OO-oriented approach: create a helper method to do the actual work:

### Listing 2. DRYing out the codebase

```
package com.tedneward.scitter
{
  // ...
  object Scitter
  {
    // ...
    private[scitter] def execute(url : String) =
    {
      val client = new HttpClient()
```

```

    val method = new GetMethod(url)

    method.getParams().setParameter(HttpMethodParams.RETRY_HANDLER,
        new DefaultHttpMethodRetryHandler(3, false))

    client.executeMethod(method)

    (method.getStatusLine().getStatusCode(), method.getResponseBodyAsString())
  }
}
}

```

### Keep DRY in mind

For those of you who have not read *Pragmatic Programmer* or haven't read it in a long time, DRY stands for "Don't Repeat Yourself." It basically suggests that any time you're typing the same bits of code over and over again, you're repeating yourself and you will eventually find yourself wishing those bits of code that do exactly the same thing were gathered into one place (like a method) so they can be easily fixed, enhanced, or replaced, later. And if you haven't read *Pragmatic Programmer* by now, shame on you. That's your homework assignment for the month.

Notice a couple of things about this: first of all, I'm returning a tuple from the `execute()` method, containing both the status code and the response body; this is one of the powerful parts of having tuples as a baked-in part of the language because it becomes really easy to hand back what, in effect, turns out to be multiple return values from a single method invocation. Of course, in Java code, we could do the same thing by creating a top-level or nested class that contains the tuple elements, but that's going to require a whole slew of code that's specific to this one particular method. Or we could hand back a `Map` with `String` keys and `Object` values, but then we lose type-safety in a big way. Tuples are not a game-changing feature, just another one of those "niceties" that makes Scala a powerful language to use.

Because I'm using a tuple, I'm going to want to use another of Scala's syntactic idioms to capture both results into local variables, such as this rewritten version of `Scitter.test`:

### Listing 3. Is this a DRY run?

```

package com.tedneward.scitter
{
  // ...
  object Scitter
  {
    /**
     * Ping the server to see if it's up and running.
     *
     * Twitter docs say:
     * test
     * Returns the string "ok" in the requested format with a 200 OK HTTP status code.
     * URL: http://twitter.com/help/test.format
     * Formats: xml, json
     * Method(s): GET
     */
  }
}

```

```
    */
    def test : Boolean =
    {
        val (statusCode, statusBody) =
            execute("http://twitter.com/statuses/public_timeline.xml")
            statusCode == 200
    }
}
```

In fact, I could easily strike `statusBody` entirely and replace it with `_` because I don't ever use the second parameter (`test` has no return body), but I'll need the body for the other calls, so I leave it here for demonstrative purposes.

Notice how `execute()` doesn't leak any of the details that deal with the actual HTTP communication — this is Encapsulation 101. This will make it easier to replace `execute()` with another implementation later (which we'll do later) or to potentially optimize the code by reusing a single `HttpClient` object instead of reinstantiating a new one each time.

### Premature optimization

By the way, the optimization here would be useful only if it turns out that `HttpClient` does something non-trivial in its constructor. Specifically, it would *not* be a useful optimization to try and cache the `HttpClient` object just to save on object allocations — the garbage collector, much maligned within the Java performance literature for years, is actually quite good at allocating and collecting short-lived objects, like the `HttpClient` object and its kin in this particular case. Most of all, though, I'm going to apply this optimization only if and when I discover that the `execute()` method is somehow a bottleneck or a waste of resources. Act accordingly.

Next, notice how the `execute()` method is on the `Scitter` object? This means that I'll be able to use it from the various `Scitter` instances (at least for now I can, until I do something inside `execute()` that prevents me from doing so) — this is why I've marked `execute()` as `private[scitter]`, which means that everything inside the `com.tedneward.scitter` package can see it.

(By the way, if you haven't already, run the tests to make sure that everything still works fine. I'm going to assume you are doing that as we go through the code, so if I forget to mention it, that doesn't mean you get to forget to do it.)

By the way, supporting the `Scitter` class is going to require a username and password for authenticated access, so I'll just create an overloaded version of the `execute()` method that takes two additional `Strings` as parameters:

### Listing 4. An even DRYer version

```

package com.tedneward.scitter
{
  // ...
  object Scitter
  {
    // ...
    private[scitter] def execute(url : String, username : String, password : String) =
    {
      val client = new HttpClient()
      val method = new GetMethod(url)

      method.getParams().setParameter(HttpMethodParams.RETRY_HANDLER,
        new DefaultHttpClientRetryHandler(3, false))

      client.getParams().setAuthenticationPreemptive(true)
      client.getState().setCredentials(
        new AuthScope("twitter.com", 80, AuthScope.ANY_REALM),
        new UsernamePasswordCredentials(username, password))

      client.executeMethod(method)

      (method.getStatusLine().getStatusCode(), method.getResponseBodyAsString())
    }
  }
}

```

In fact, given that the two `execute()`s do pretty much the same thing modulo the authentication bits, we can rewrite the first `execute()` entirely in terms of the second with the caveat that Scala requires us to be explicit about the return type of the overloaded `execute()`:

### Listing 5. Desert DRY

```

package com.tedneward.scitter
{
  // ...
  object Scitter
  {
    // ...
    private[scitter] def execute(url : String) : (Int, String) =
      execute(url, "", "")
    private[scitter] def execute(url : String, username : String, password : String) =
    {
      val client = new HttpClient()
      val method = new GetMethod(url)

      method.getParams().setParameter(HttpMethodParams.RETRY_HANDLER,
        new DefaultHttpClientRetryHandler(3, false))

      if ((username != "") && (password != ""))
      {
        client.getParams().setAuthenticationPreemptive(true)
        client.getState().setCredentials(
          new AuthScope("twitter.com", 80, AuthScope.ANY_REALM),
          new UsernamePasswordCredentials(username, password))
      }

      client.executeMethod(method)

      (method.getStatusLine().getStatusCode(), method.getResponseBodyAsString())
    }
  }
}

```

So far, so good. We've DRY'ed the communication parts of the Scitter code so let's move on to the next thing on the list: Let's get a list of my friends' tweets.

## Connecting (to friends)

The Twitter API states that the `friends_timeline` API call "returns the 20 most recent statuses posted by the authenticating user and that user's friends." (It also points out, for those of you who use Twitter from the Twitter Web site directly, that "This is the equivalent of '/home' on the Web.") This is a pretty basic requirement for any Twitter API, so let's add that to the `Scitter` class; we add it to the class and not the object because, as the docs point out, this is done on behalf of the authenticating user, which I've decided belongs in the `Scitter` class and not the `Scitter` object.

Here we get tossed a curveball, though: The `friends_timeline` call accepts a list of "optional parameters" including `since_id`, `max_id`, `count`, and `page` to control the results returned. This is going to be a tricky operation because Scala doesn't natively support the idea of "optional parameters" the way that other languages (like Groovy, JRuby, or JavaScript) do, but let's deal with the simple stuff first — let's create a `friendsTimeline` method that just executes the normal, non-parameterized call:

### Listing 6. "Tell me what company thou keepst..."

```
package com.tedneward.scitter
{
  class Scitter
  {
    def friendsTimeline : List[Status] =
    {
      val (statusCode, statusBody) =
        Scitter.execute("http://twitter.com/statuses/friends_timeline.xml",
                       username, password)

      if (statusCode == 200)
      {
        val responseXML = XML.loadString(statusBody)

        val statusListBuffer = new ListBuffer[Status]

        for (n <- (responseXML \\ "status").elements)
          statusListBuffer += (Status.fromXml(n))

        statusListBuffer.toList
      }
      else
      {
        Nil
      }
    }
  }
}
```

So far, so good; the corresponding method to test this looks something like this:

### Listing 7. "... and I'll tell thee what thou art." (Miguel de Cervantes)

```
package com.tedneward.scitter.test
{
  class ScitterTests
  {
    // ...

    @Test def scitterFriendsTimeline =
    {
      val scitter = new Scitter(testUser, testPassword)
      val result = scitter.friendsTimeline
      assertTrue(result.length > 0)
    }
  }
}
```

Perfect. Looks just like the `publicTimeline()` method from the `Scitter` object and behaves almost exactly the same way, too.

#### Futures....

As part of writing this article, I asked the authors of *Programming in Scala* — Bill Venners, Martin Odersky, and Lex Spoon — what they thought the best and/or most idiomatic way to deal with optional parameters would be and in the ensuing discussion, Martin mentioned that this is an area that they're looking into, most possibly for the 2.8 release. Whether that will take the form of "default parameters" or "optional parameters" isn't quite clear yet, at least not to me, but if this is an area that particularly intrigues you, now is the time to chime in on the mailing lists.

We still have the problem of those optional parameters. Because Scala doesn't have optional parameters as a language feature, the only option at first blush seems to be to create a whole slew of overloaded `friendsTimeline()` methods, taking some factorially governed number of parameters.

Fortunately, there is a better way to do this by combining two of Scala's language features (one of which I haven't mentioned yet) in an interesting way — case classes and "repeated parameters" (look at Listing 8):

### Listing 8. "How do I love thee? ..."

```
package com.tedneward.scitter
{
  // ...

  abstract class OptionalParam
  case class Id(id : String) extends OptionalParam
  case class UserId(id : Long) extends OptionalParam
  case class Since(since_id : Long) extends OptionalParam
}
```

```

case class Max(max_id : Long) extends OptionalParam
case class Count(count : Int) extends OptionalParam
case class Page(page : Int) extends OptionalParam

class Scitter(username : String, password : String)
{
  // ...

  def friendsTimeline(options : OptionalParam*) : List[Status] =
  {
    val optionsStr =
      new StringBuffer("http://twitter.com/statuses/friends_timeline.xml?")
    for (option <- options)
    {
      option match
      {
        case Since(since_id) =>
          optionsStr.append("since_id=" + since_id.toString() + "&")
        case Max(max_id) =>
          optionsStr.append("max_id=" + max_id.toString() + "&")
        case Count(count) =>
          optionsStr.append("count=" + count.toString() + "&")
        case Page(page) =>
          optionsStr.append("page=" + page.toString() + "&")
      }
    }

    val (statusCode, statusBody) =
      Scitter.execute(optionsStr.toString(), username, password)
    if (statusCode == 200)
    {
      val responseXML = XML.loadString(statusBody)

      val statusListBuffer = new ListBuffer[Status]

      for (n <- (responseXML \\ "status").elements)
        statusListBuffer += (Status.fromXml(n))

      statusListBuffer.toList
    }
    else
    {
      Nil
    }
  }
}

```

See the \* tagging the end of the options parameter? That indicates that the parameter is actually a sequence of parameters, very much akin to the Java 5 `varargs` construct. Just as with `varargs`, the number of parameters passed can be zero just as we had before (although we will need to go back into the test code and add a pair of parentheses to the `friendsTimeline` call now, otherwise the compiler won't know if we're trying to call the method with no parameters or trying to use it for partial-application purposes or something similar); we can also start passing those case types around, as in the following:

### Listing 9. "... Let me count the ways" (William Shakespeare)

```

package com.tedneward.scitter.test
{

```

```

class ScitterTests
{
  // ...

  @Test def scitterFriendsTimelineWithCount =
  {
    val scitter = new Scitter(testUser, testPassword)
    val result = scitter.friendsTimeline(Count(5))
    assertTrue(result.length == 5)
  }
}
}

```

Of course, there's always the possibility that a sociopathic client could pass in some truly bizarre sequence of parameters, such as `friendsTimeline(Count(5), Count(6), Count(7))`, but in this case we'll just blindly hand the list over to Twitter (and hope that their error-handling is strong enough to just take the last one specified). Of course, if it becomes a concern, it's not hard to walk through the list of repeated parameters and take the last one of each kind specified before constructing the URL sent to Twitter. In the meantime, *Caveat emptor*.

## Compatibility

This does raise an interesting question, though: How easy is it to call this method from Java code? After all, if one of the major goals of this library is to maintain compatibility to Java code, then we need to make sure that Java code doesn't have to jump through too many hoops to use it.

Let's start by tossing the `Scitter` class at our good friend `javap`:

### Listing 10. Oh, yeah, Java code... I remember now....

```

C:\>javap -classpath classes com.tedneward.scitter.Scitter
Compiled from "scitter.scala"
public class com.tedneward.scitter.Scitter extends java.lang.Object implements s
cala.ScalaObject{
    public com.tedneward.scitter.Scitter(java.lang.String, java.lang.String);
    public scala.List friendsTimeline(scala.Seq);
    public boolean verifyCredentials();
    public int $tag()          throws java.rmi.RemoteException;
}

```

Yikes! Two things jump out at me here as concerns. First, `friendsTimeline()` takes a `scala.Seq` as a parameter (which is the repeated parameters feature we just used). Second, the `friendsTimeline()` method, just as with the `publicTimeline()` method from the `Scitter` object (run `javap` on it to double-check if you don't believe me) returns a `scala.List` of elements. How usable are these two types from Java code?

The easiest way to find out is to write a small set of JUnit tests in Java code rather

than in Scala, so let's do that. While we could go and test the construction of the `Scitter` instance and call its `verifyCredentials()` method, those aren't particularly useful — remember, we're not here (in this case) to verify the correctness of the `Scitter` class but to see how easy it is to use the thing from Java code. Toward that end, let's just jump directly to writing a test that will fetch the "friends timeline" — in other words, we want to instantiate a `Scitter` instance and invoke its `friendsTimeline()` method with no parameters.

Doing this is a bit trickier, thanks to the `scala.Seq` parameter that we need to pass in — `scala.Seq` is a Scala trait which means it gets mapped to the underlying JVM as an interface, so we can't just instantiate one directly. We can try the classic Java `null` parameter, but doing so throws an exception at run time. What we need is a `scala.Seq` class that we can easily instantiate from Java code.

Turns out, we find one in the very same `mutable.ListBuffer` type that we use inside the `Scitter` implementation itself:

### Listing 11. And now I remember why I like Scala....

```
package com.tedneward.scitter.test;

import org.junit.*;
import com.tedneward.scitter.*;

public class JavaScitterTests
{
    public static final String testUser = "TESTUSER";
    public static final String testPassword = "TESTPASSWORD";

    @Test public void getFriendsStatuses()
    {
        Scitter scitter = new Scitter(testUser, testPassword);
        if (scitter.verifyCredentials())
        {
            scala.List statuses =
                scitter.friendsTimeline(new scala.collection.mutable.ListBuffer());
            Assert.assertTrue(statuses.length() > 0);
        }
        else
            Assert.assertTrue(false);
    }
}
```

Using the returned `scala.List` isn't a problem because we can just treat it as much like we would any other `Collection` class (though we do miss some of the `Collections` API niceties because the Scala-based methods on `List` all assume you will be interacting with them from Scala), so walking through the results isn't all that difficult, if a bit "old-school" Java code (circa 1995):

### Listing 12. 1995 called; they want their `Vector` back....

```
package com.tedneward.scitter.test;
```

```

import org.junit.*;
import com.tedneward.scitter.*;

public class JavaScitterTests
{
    public static final String testUser = "TESTUSER";
    public static final String testPassword = "TESTPASSWORD";

    @Test public void getFriendsStatuses()
    {
        Scitter scitter = new Scitter(testUser, testPassword);
        if (scitter.verifyCredentials())
        {
            scala.List statuses =
                scitter.friendsTimeline(new scala.collection.mutable.ListBuffer());
            Assert.assertTrue(statuses.length() > 0);

            for (int i=0; i<statuses.length(); i++)
            {
                Status stat = (Status)statuses.apply(i);
                System.out.println(stat.user().screenName() + " said " + stat.text());
            }
        }
        else
            Assert.assertTrue(false);
    }
}

```

Which brings us to the next part, that of passing the parameters into the `friendsTimeline()` method. Unfortunately, the `ListBuffer` type doesn't take a collection as a constructor parameter, so we have to construct the list of params, then pass the collection in on the method call; it's tedious, but not overly burdensome:

### Listing 13. Can I go back to Scala now, please?

```

package com.tedneward.scitter.test;

import org.junit.*;
import com.tedneward.scitter.*;

public class JavaScitterTests
{
    public static final String testUser = "TESTUSER";
    public static final String testPassword = "TESTPASSWORD";

    // ...

    @Test public void getFriendsStatusesWithCount()
    {
        Scitter scitter = new Scitter(testUser, testPassword);
        if (scitter.verifyCredentials())
        {
            scala.collection.mutable.ListBuffer params =
                new scala.collection.mutable.ListBuffer();
            params.$plus$eq(new Count(5));

            scala.List statuses = scitter.friendsTimeline(params);

            Assert.assertTrue(statuses.length() > 0);
            Assert.assertTrue(statuses.length() == 5);

            for (int i=0; i<statuses.length(); i++)

```

```
        {
            Status stat = (Status)statuses.apply(i);
            System.out.println(stat.user().screenName() + " said " + stat.text());
        }
    }
    else
        Assert.assertTrue(false);
}
}
```

So although the Java version is a bit more verbose than its Scala equivalent, so far it's still pretty straightforward to call the Scitter library from any Java client that might want to use it. Excellent.

## Conclusion

Clearly there's a long way yet to go with Scitter, but it's starting to really take shape and form and so far, it's feeling pretty good. We've managed to DRY-ify the communication parts of the Scitter library and to incorporate the optional parameters that Twitter asks for a number of the different API calls it provides — and so far Java clients aren't going to be too deeply put off by the API we're exposing. Granted, it's not nearly as clean an API as the one that Scala can consume naturally, but if Java developers want to use the Scitter library, they don't have too much to do to get started.

The Scitter library still has something of an "objectish" feel to it, but we're starting to see some of Scala's "functionalish" features creeping their way into the system. As we continue building out the library, more and more of those features will begin to worm their way in, anywhere that they help make the code more concise and more clear. And that's as it should be.

For now, it's time for us to part ways while I take a brief hiatus. When I return, I'll add some support for offline testing, plus the ability to update a user's status to the library. Until then, Scala fans, remember: It's always better to be functional than dysfunctional. (Sorry. I like that joke just way too much.)

# Resources

## Learn

- [The busy Java developer's guide to Scala](#) (Ted Neward, developerWorks): Read the complete series.
- ["Scala and XML"](#) (developerWorks, April 2008) shows how Scala can make working with XML a joy.
- This [wiki on the Scala lift framework](#), a framework for writing Web applications, demonstrates the framework's famous underpinnings (Seaside, Rails, Django, Wicket, etc.). Here's a [getting started tutorial](#).
- ["Functional programming in the Java language"](#) (Abhijit Belapurkar, developerWorks, July 2004): Explains 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 (in PDF).
- [Programming in Scala](#) (Martin Odersky, Lex Spoon, and Bill Venner; Artima, December 2007): The first book-length introduction to Scala.
- [Bjarne Stroustrup](#): Designed and implemented C++, which he has described as "a better C."
- [Java Puzzlers: Traps, Pitfalls, and Corner Cases](#) (Addison-Wesley Professional, July 2005) reveals oddities of the Java programming language through entertaining and thought-provoking programming puzzles.
- The [developerWorks Java technology zone](#): Hundreds of articles about every aspect of Java programming.

## Get products and technologies

- The [Jakarta \(Apache\) Commons HttpClient component](#) provides an efficient, up-to-date, feature-rich package implementing the client side of the most recent HTTP standards and recommendations. You may also want the [Commons Logging](#) and [Commons Codec](#) components.
- [Download Scala](#): Start learning it with this series.
- [SUnit](#): Part of the standard Scala distribution, in the *scala.testing* package.

## Discuss

- [developerWorks blogs](#): Get involved in the [developerWorks community](#).

## About the author

Ted Neward

Ted Neward is a Principal Consultant at ThoughtWorks, a global consultancy firm, where he consults, mentors, 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.