Notes

Scala 1

General 1

Access 1

Testing 2

Pattern Matching 2

Links 2

Variable Binding 3

Sealed 3

Match Sequences 4

Guards (if statements in cases) 4

Patterns to pick out arguments 4

Options 5

Implicit Class 6

Implicit Functions 6

Implicit parameters 7

Functions 8

Emacs 9

Highlighting 9

Macros 9

Undo 9

Git 9

Cloning 9

Remotes 10

Forks 11

File Management 12

Logs and Diffs 12

Avoid merge commits 13

Kafka 15

SBT 15

# Scala

## General

“def” = evaluate at time of call

“val” = evaluate at time of definition

“lazy val” = evaluate when evaluated the first time

## Access

From <http://www.ibm.com/developerworks/library/j-scala07298/>

“you can use import anywhere inside the client Scala file, not just at the top of the file and correspondingly, will have scoped relevance”

“use import to bring not just nested types into lexical scope, but any member”

“import can take multiple, comma-separated targets”

“Scala:

* Uses "public" by default
* Specifies "private" to mean "accessible only to this scope"

By contrast, "protected" is definitely different from its counterpart in Java code; where a Java protected member is accessible to both subclasses and the package in which the member is defined, Scala chooses to grant access only to subclasses. This means that Scala's version of protected is more restrictive (although arguably more intuitively so) than the Java version.”

“access modifiers in Scala can be "qualified" with a package name, indicating a level of access *up to* which the member may be accessed”

“the *object-private* specification, illustrated by private[this], which stipulates that the member in question can only be seen by members called on that same object, not from different objects, even if they are of the same type”

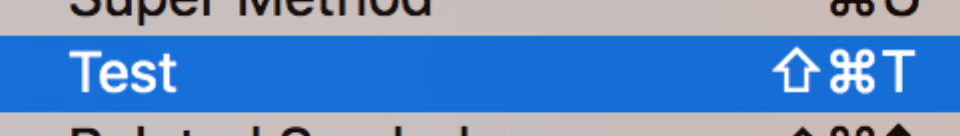
## Testing

Add the following dependency:

“org.scalatest” % “scalatest\_2.10” % “2.1.0” % “test”

to build.sbt and let Idea load the library.

Use Idea to create a test by using “Navigate -> Test”



Your test class should extend org.scalatest.FunSuite and you can do this automatically by selecting ScalaTest as the Testing Library.



## Pattern Matching

### Links

<http://www.artima.com/pins1ed/case-classes-and-pattern-matching.html>

### Variable Binding

“A variable pattern matches any object, just like a wildcard. Unlike a wildcard, Scala binds the variable to whatever the object is. You can then use this variable to act on the object further.”

case Some(x) => println(“I got x: ” + x)

“In addition to the standalone variable patterns, you can also add a variable to any other pattern. You simply write the variable name, an at sign (@), and then the pattern. This gives you a variable-binding pattern. The meaning of such a pattern is to perform the pattern match as normal, and if the pattern succeeds, set the variable to the matched object just as with a simple variable pattern.”

case x @ Some(\_) => println(“I got the entire Option object, not just what is inside: ” + x)

“A constructor pattern looks like

case BinOp("+", e, Number(0)) => println(“good”)

It consists of a name (e.g. BinOp) and then a number of patterns within parentheses (e.g. "+", e, and Number(0)). Assuming **the name designates a case class**, such a pattern means to first check that the object is a member of the named case class, and then to check that the constructor parameters of the object match the extra patterns supplied.”

### Sealed

When using the “match” statement, you must supply every possible match pattern because unlike Java’s switch statement, there is no “default” fall through statement. You would usually do this on your own by specifying a pattern that matches anything at the end.

case \_ => println(“matches everything and anything”)

“In fact, you can enlist the help of the Scala compiler in detecting missing combinations of patterns in a match expression. To be able to do this, the compiler needs to be able to tell which are the possible cases; make the superclass of your case classes **sealed**. A sealed class cannot have any new subclasses added except the ones in the same file.”

**sealed** abstract class Expr

  case class Var(name: String) extends Expr

  case class Number(num: Double) extends Expr

### Match Sequences

Sequence patterns can be thought of as a special case to constructor patterns for case classes.

“You can match against sequence types like List or Array just like you match against case classes. Use the same syntax, but now you can specify any number of elements within the pattern.”

case **List(0, \_, \_)** => println("found it")

This matches any list that starts with zero and has a length of three. To match an arbitrary length list you need to use “\_\*”

case List(0, \_\*) => println("found it")

### Guards (if statements in cases)

“Scala restricts patterns to be linear: a pattern variable may only appear once in a pattern.”

case BinOp("+", x, x) => BinOp("\*", x, Number(2))

This will fail to compile because you used “x” twice in the pattern. But you really want the two arguments to be the same or else you can’t just assume that on the right side of the case statement. That is, if you use

case BinOp("+", x, y)

That will match any two arguments and they don’t have to be the same. To resolve this, add an “if” (before the arrow) to guard against matches that succeed according to your pattern but is not exactly what you wanted.

case BinOp("+", x, y) **if x == y** => BinOp("\*", x, Number(2))

### Patterns to pick out arguments

If you already have a value defined like this

val myTuple = (123, "abc")

And you would like to draw out the data from the value, you can use a pattern like this to assign the data in the tuple to other values.

val (number, string) = myTuple

Now the values “number” and “string” have values from the tuple, 123 and “abc”, respectively.

You can do this with any constructor and not just with tuples

val exp = new BinOp("\*", Number(5), Number(1))

val BinOp(op, left, right) = exp

Now, the values op, left, and right have the data “\*”, Number(5), and Number(1), respectively. A nice way to remember this is to think that the pattern is “de-constructing” the constructor.

## Options

You can treat options **like a sequence**, even though they are not. If there is “something” in it, it behaves like a sequence of one value and if it is “None” then it behaves like an empty sequence.

Because of this, you can write less verbose code than your average match-case blocks. For example:

val result:Option[Type] = functionThatReturnsOption

result.foreach( t => println(“got it:” + t) )

The foreach won’t do anything to an empty list but will do something for the one object in the option.

The other thing you can do with Options is to map them. Options provide a map method that takes any Some[A] to Some[B] and None to None. So if I have an Account object that I just looked up from some DAO and that DAO returned an Option of Some[Account] or None and I wanted the balance (an integer) from the account, I can map the value:

someAcct.map(\_.getBalance)

and this will return either Some[Int] or None

However, if the Account class had a method “address” that also returned an option and I do:

someAcct.map(\_.getAddress)

I would get Some(Some(Address)) or None. I get a double Option[Option[]] because the getAddress method also returns an option. Do get rid of this, I can use the flatmap method in Option to flatten out the nested Option objects.

someAcct.flatMap(\_.getAddress)

You can call “filter” on them like a sequence. The filter method will return the Some[] to you if the predicate returns true; otherwise it returns None or if the option was None to begin with.

someAcct.filter(\_.getState == “MD”)

## Implicit Class

<http://alvinalexander.com/scala/scala-2.10-implicit-class-example>

“In Scala 2.10, you define an *implicit class*, and then define methods within that class to implement the behavior you want.”

## Implicit Functions

From <https://twitter.github.io/scala_school/advanced-types.html>

Implicit functions can automatically plopped into places where the input/output types fit.

implicit def strToInt(x: String) = x.toInt

math.max("123", 111)

The second line would usually fail since the max function expects an integer. But with the implicit function, scala calls it automatically for you.

Implicit functions are helpful with type conversions. You can specify in your class that your client must have an implicit conversion function when they instantiate your class or else a compile time error is generated. You use the **<%** operator, which I think is called “viewable” so type A must be viewable as an Int. I guess the percent sign looks like a pair of glasses.

class Container[A <% Int] { def addIt(x: A) = 123 + x }

The class Container takes a **generic** type that is bounded by the existence of an implicit function to convert a type A to Int. So with the implicit function we already defined above we can create a Container of Strings.

(new Container[String]).addIt("123")

But not a Container of Floats

(new Container[Float]).addIt(123.2F)  
<console>:8: error: could not find implicit value for evidence parameter of type (Float) => Int

## Implicit parameters

They are used as an equivalent to Haskell’s Type Classes. Similar to the above examples, our type class will be a trait. For example, if I created an API that consist of a class “MyApiClass” and a function that will take some parameter of an unknown type (It is some class of my clients, that I can’t know ahead of time) and the function must use that parameter so it can produce an instance of MyApiClass as a result. How would my client be able to use my API? In Java, I would provide an interface that my client will need to implement and the client will need to pass its implementation into my function. This couples my API with my client’s main class hierarchy. In scala, we have another option: specify an implicit parameter in the definition of my API function and the definition of a “type class” trait. Doing that allows the client code to create an object outside of its main class hierarchy that extends my trait to provide the implementation needed for my function to work. In Java, this just means that the client must pass another object into my function and that object will implement my interface instead of the client’s main object. This is also happening in scala however the object is passed in automatically (or implicitly) by the compiler just because the compiler sees the right type of object in scope. So it’s a round about way of doing things which in Java would add the cost of an extra parameter in the client’s method calls. But in scala, that cost is taken up by the compiler.

class MyApiClass

trait CanDoStuffMyApiNeeds[A] { def doStuff(param: A): MyApiClass }

def myApiMethod[T](param: T)(implicit i: CanDoStuffMyApiNeeds[T]) = i.doStuff(param)

class SomeClientUsingMyApi

val client = new SomeClientUsingMyApi

scala> myApiMethod(client)

<console>:13: error: could not find implicit value for parameter i: CanDoStuffMyApiNeeds[SomeClientUsingMyApi]

myApiMethod(client)

// now create an object that extends CanDoStuffMyApiNeeds in scope

implicit object glueClientToApi extends CanDoStuffMyApiNeeds[SomeClientUsingMyApi] { def doStuff(param: SomeClientUsingMyApi) = new MyApiClass }

// call it again and it’s okay

scala> myApiMethod(client)

res1: MyApiClass = MyApiClass@551aa95a

Additionally, if instead of taking a client typed parameter that you can’t know about, your API might provide all of the types that the function can use. Instead of polluting your own types with concerns that might not have anything to do with your classes. Here’s another example from <http://docs.scala-lang.org/tutorials/tour/implicit-parameters>:

abstract class SemiGroup[A] {

def add(x: A, y: A): A

}

abstract class Monoid[A] extends SemiGroup[A] {

def unit: A

}

object ImplicitTest extends App {

implicit object StringMonoid extends Monoid[String] {

def add(x: String, y: String): String = x concat y

def unit: String = ""

}

implicit object IntMonoid extends Monoid[Int] {

def add(x: Int, y: Int): Int = x + y

def unit: Int = 0

}

def sum[A](xs: List[A])(implicit m: Monoid[A]): A =

if (xs.isEmpty) m.unit

else m.add(xs.head, sum(xs.tail))

println(sum(List(1, 2, 3)))

println(sum(List("a", "b", "c")))

}

## Functions

<http://alvinalexander.com/scala/how-to-use-functions-as-variables-values-in-scala-fp>

“Assigning an existing function/method to a function variable”

val c = scala.math.cos \_

The value “c” is now a function that I can pass into another function.

# Emacs

## Highlighting

<http://stackoverflow.com/questions/18090378/turn-on-background-color-when-highlighting-with-c-spc-on-a-mac>

M-x transient-mark-mode to toggle the highlighting

## Macros

c-x, ( to start

c-x, ) to end

f4 to execute

## Undo

c-\_

# Git

<https://git-scm.com/docs>

<http://ftp.newartisans.com/pub/git.from.bottom.up.pdf>

## Cloning

Make your own local repository by copying from a remote:

git clone -o MY\_NAME\_FOR\_REMOTE -b BRANCH <https://github.com/REPO.git> MY\_DIR\_TO\_WRITE\_TO

Or if you grabbed the remote without the branch information like this

git clone -o REMOTE\_NAME URL DIRECTORY

You can fetch the branch later and switch to it with this

git fetch REMOTE\_NAME BRANCH\_NAME

git checkout BRANCH\_NAME

## Remotes

Ask for a list of references on any remote repository

git ls-remote REMOTE\_NAME\_OR\_URL

If you're in a git repository that you cloned, you can leave off the REMOTE\_NAME\_OR\_URL and it'll default to the remote you cloned from.

References are "alias" to sha1 commit ID numbers.

If you're inside a git directory, you can list the remotes your local git is tracking (most likely from cloning, pushing, or pulling)

git remote

If there isn't one listed that you want, you can add it with

git remote add WHATEVER\_NAME URL

After adding a remote, you can fetch and merge it into your repo

Fetch references (branches, tags, updated objects, etc.) from a remote repo to your repo

git fetch REMOTE\_NAME\_OR\_URL

Fetching will get all the changes in references from the remote but it will not touch files in your working directory. You can merge the changes yourself after the fetch with:

git merge REMOTE/BRANCH

The merge will **merge and commit** to your local branch unless you run it with --no-commit option.

This tells me what remote branches I am tracking in my local repo and where it'll push and pull from

git remote show REMOTE\_NAME

List your remotes

git remote -v

Change which the URL of your remotes

git remote set-url REMOTE\_NAME NEW\_URL

Rename your remote

git remote rename OLD\_NAME NEW\_NAME

This tells me which local branch I am currently using

git branch -vv

If I want to add another remote for my local to track:

git checkout --track REMOTE/BRANCH\_IN\_REMOTE

(above) will create a local branch for me and then I can switch to it in my local with

git checkout BRANCH

All in one:

git checkout -t REMOTE\_NAME -b BRANCH\_NAME

Once I am in my local branch I can **reconfigure** where it pushes and pulls from

git branch -u REMOTE/SOME\_OTHER\_BRANCH\_IN\_REMOTE

## Forks

Synching your forked repo with the original

1. Make sure you added the original as a remote (assume you named it ‘origin’)
2. Fetch the original remote/branch (creates a new branch locally for you)
3. Switch over to your forked branch
4. Merge in changes from the original branch

git remote add YOUR\_ORIGIN\_REMOTE\_NAME YOUR\_ORIGIN\_REMOTE\_URL

git fetch YOUR\_ORIGIN\_REMOTE\_NAME

git checkout YOUR\_FORK\_BRANCH

git merge YOUR\_ORIGIN\_REMOTE\_NAME/YOUR\_ORIGIN\_BRANCH\_NAME

The above will synch up your local repo but if you want to update the branch that you’re on, then push the merged changes from your local to your remote.

git push

Merge from the upstream master to your local so that you can commit it back into your forked remote. Note that by default, merge will commit to your local repo if there are no merge conflicts. If there are, you will need to resolve them by hand and then commit them into your repo. Or you can abort the merge and git will return merged files back to the way it was before merging although it may not work all the time, so it is best to only do merges when you have already committed everything to your repo.

git merge upstream/master

## File Management

List what is checked into your repo

git ls-files

Revert a file back to is checked in

git checkout FILE

Remove files from git (not from your working directory)

git rm FILE

Delete a branch on a **remote**

git push REMOTE\_NAME --delete BRANCH\_NAME

Delete a branch locally

git branch --delete BRANCH\_NAME

## Logs and Diffs

One of the more helpful options is -p, which shows the difference introduced in each commit. You can also use -2, which limits the output to only the last two entries:

git log -p -2

Doing a diff on a previous version

git diff HEAD~1:FILE\_PATH YOUR\_FILE\_PATH

For example, diffing what I have locally in my BDFD-284 branch in the src directory to what is in second look master

git diff BDFD-284:src upstream/master:src

git diff upstream/master myfork/BDFD-284

Diff two commits with

git diff COMMIT1\_SHA1 COMMIT2\_SHA1

You can get the hashes by using git log.

## Avoid merge commits

<http://kernowsoul.com/blog/2012/06/20/4-ways-to-avoid-merge-commits-in-git/>

Whenever we merge changes in from another branch (e.g. merging updates into our forked repo), the merge happens and then we must commit the merged changes on top of our own changes. And if we later merge our forked branch back to the trunk, the history will show this merge which is just noise since it really represents changes already done to the trunk. To avoid this, we need to run “rebase” which will merge changes from trunk into your local repo at a point that is where your local repo diverged. In other words, it *should* “rewind” your commits, apply the merge, and then “replay” your commits on top of the merged changes.

Loosely, a regular “pull” equals a “fetch” + “merge” and a “pull --rebase” equals a “fetch” + “rebase”. So to understand the difference between a regular pull and a pull with rebase, we should look at the difference between merge and rebase. See <https://www.atlassian.com/git/tutorials/merging-vs-rebasing/conceptual-overview>

One thing I have seen over and over in write-ups about rebasing is to **never** do it in a public branch; it should only be done in your local repo/branch. One gotcha in particular refers to creating a Pull Request. Once created, the Pull Request is now public and doing a rebase here would confuse Git and people who are reviewing your request. So, you should rebase before creating the Pull Request and only do merges afterwards.

Here is an example of something I have done. I have created a branch off of master on the pubic repository. Locally, I have both master and my branch checked out. I synched up both master and my branch locally. I made changes to my branch locally and I committed locally. Someone else made changes to master on the public repo. I switched to my local master and did a pull to update my local master with the new commits. I switch back to my local branch and issue a

git rebase master

After that, my local commits look like that happen after the last commit to master by someone else. So when I merge my branch into master, it will look the same way.

|  |  |
| --- | --- |
| My local started with this: |  |
| Master was updated to this: |  |
| After the rebase, my local repo looks like this: |  |

Instead of doing a pull on my local master, I have also done a fetch followed by a merge.

git checkout master

git fetch upstream

git merge upstream/master

git checkout BRANCH

# Kafka

Create a topic

/Applications/kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 --partitions 1 --replication-factor 1 --topic mytopic

List all topics

/Applications/kafka/bin/kafka-topics.sh --list --zookeeper localhost:2181

Write to topic

/Applications/kafka/bin/kafka-console-producer.sh --broker-list localhost:9092 --topic mytopic

Read from topic

/Applications/kafka/bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic mytopic --from-beginning

# SBT

The % method is used to construct an Ivy module ID from strings

Keys have an overloaded method called in used to set the scope. The argument to in can be an instance of any of the scope axes.

For example, set the name scoped to the Compile configuration

name in Compile := "hello"

it’s important to understand that in and := are just methods, not magic. Scala lets you write them in a nicer way, but you could also use the Java style:

name.in(Compile).:=("hello")

To change the value associated with the compile key, you need to write compile in Compile or compile in Test. Using plain compile would define a new compile task scoped to the current project, rather than overriding the standard compile tasks which are scoped to a configuration.

Assignment with := is the simplest transformation, but keys have other methods as well. If the T in SettingKey[T] is a sequence, i.e. the key’s value type is a sequence, you can append to the sequence rather than replacing it.

* += will append a single element to the sequence.
* ++= will concatenate another sequence.

Unmanaged dependencies work like this: add jars to lib and they will be placed on the project classpath.

Dependencies in lib go on all the classpaths (for compile, test, run, and console).

There’s nothing to add to build.sbt to use unmanaged dependencies, though you could change the unmanagedBase key if you’d like to use a different directory rather than lib.

sbt uses Apache Ivy to implement managed dependencies

you can simply list your dependencies in the setting libraryDependencies

The key “libraryDependencies” is a settings key of a sequence (aka list) of “module IDs”

Module ID objects are created with the “%” method and can be chained together like this:

libraryDependencies += groupID % artifactID % revision % configuration

Using ivy, sbt will download your dependencies from either its standard list of repositories or ones you described and store them in the “.ivy2” directory in your home directory. The “update” task will do this. The “compile” task depends on “update”

Of course, you can also use ++= to add a list of dependencies all at once:

libraryDependencies ++= Seq(  
  groupID % artifactID % revision,  
  groupID % otherID % otherRevision  
)

resolvers does not contain the default resolvers; only additional ones added by your build definition.

sbt combines resolvers with some default repositories to form externalResolvers.

Therefore, to change or remove the default resolvers, you would need to overrideexternalResolvers instead of resolvers.

If you want a dependency to show up in the classpath only for the Test configuration and not the Compile configuration, add % "test" like this:

libraryDependencies += "org.apache.derby" % "derby" % "10.4.1.3" % "test"

If your project is in directory hello, and you’re adding sbt-site plugin to the build definition, create hello/project/site.sbt and declare the plugin dependency by passing the plugin’s Ivy module ID to addSbtPlugin:

addSbtPlugin("com.typesafe.sbt" % "sbt-site" % "0.7.0")

If you’re adding sbt-assembly, create hello/project/assembly.sbt with the following:

addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.11.2")

Not every plugin is located on one of the default repositories and a plugin’s documentation may instruct you to also add the repository where it can be found:

resolvers += Resolver.sonatypeRepo("public")