# Scala Coding Guidelines

**Quantexa Coding Guidelines:**

| **General Scala/Spark** | |
|---|---|
| Camel Case | Use camel casing, not underscores! Vals/vars/methods(defs) should all be lowerCamelCase, types/objects should all be UpperCamelCase (also known as ProperCase), for further details please see this style guide.<br><br>**Exceptions**<br><br>1. In ETL processes, the raw data should be read into a parquet maintaining all of the source field names exactly.<br>2. Constant values - use ProperCase. For example, val YearInDays = 365<br>3. When using acronyms and camel case, if it is the first word within a variable name it should be lower case, if it is not the first word, it should be all upper case. E.G:<br>  ○ jiraTaskDescription (and not JIRATaskDescription)<br>  ○ accountIBAN (and not accountIban)<br>4. When using acronyms and proper case, every letter of the acronym should be upper case, e.g.<br>  ○ case class ETLConfig(hdfsPath: String)<br>5. Packages all lower case, including package objects. |
| Abbreviation (don't do it!) | Dont abbreviate words in names, for example:<br><br>• accountDate (not accountDt)<br>• accountDateTime (not accountDtTm, and not accountDatetime)<br><br>**Exceptions**<br><br>• When using Id (for Identification or Identifier), do not shorten and use as lowerCamelCase, for example accountId |
| Split long lines of code | Long lines of code should be split over multiple lines. The Scala style guide suggests 80 characters as an upper bound, but discretion is allowed (R&D use 120 because they have big monitors!). If it doesn't fit on your screen without scrolling then the line definitely needs splitting! |

| **General Scala/Spark** | |
|---|---|
| Comments | If you need to comment your code to explain something, instead question whether you can write the code in a clearer way (generally the way to do this should result in more functional code). Good code should read like a story; if more detail is required, the developer can click through to see more information. A simple example code which follows this notion is as follows:<br><br>Examples of where comments are normally required:<br><br>1. Where decisions are being made which can't be explained through code (e.g. features of the data and business feedback which determined how to write a score)<br>2. "TODO"s and "FIXME" comments (see below) |
| Scaladocs | See style guide. Scaladocs are effectively comments about what something does and for methods/functions the parameters it takes. Objects and key methods/functions should have Scaladocs explaining what they are/do. An example from the style guide:<br><br>1. /** Creates a person with a given name and birthdate<br>2. *<br>3. * @param name their name<br>4. * @param birthDate the person's birthdate<br>5. * @return a new Person instance with the age determined by the<br>6. * birthdate and current date.<br>7. */<br>8. def apply(name: String, birthDate: java.util.Date) = {}<br>9. } |

| **General Scala/Spark** | |
|---|---|
| TODO/FIXME | When something is left to do, use a TODO:<br><br>//TODO: Add support for Longs<br><br>When something may not work as expected, use a FIXME:<br><br>//FIXME: Pattern match will fail if user provides Integer parameters<br><br>In both cases, be **specific** about what the task is, as it may be a different developer who works on the task. |
| Hard-coded configuration | Do not hard code configuration! Examples which should be externalised into a configuration project (e.g. type-safe) include:<br><br>1. Locations of raw data / output data<br>2. Iteration number<br>3. Dates<br>4. Scoring Parameters where the client wishes to be able to change these parameters |
| Regular Expressions | Regular expressions are compiled at run time, and add significant overhead. As such, where possible they should be defined once only. In other words: |

| **Spark only** | |
|---|---|
| Broadcast joins | When one dataset in a join is small it is a good idea to suggest to Spark to broadcast the small dataset.<br><br>largeDF.join(broadcast(smallDF), Seq("foo"))<br><br>A broadcast join is sort of like a hash merge in SAS - instead of shuffling both datasets to do a join the smaller one is 'sent' to each executor which means the large dataset doesn't need to 'move'. This is more efficient. |

| | |
|---|---|
| Joins on same column(s) | **Use** df1.join(df2, Seq("id"))<br><br>**Dont Use** df1.join(df2, df1("id") === df2("id"))<br><br>This is only applicable where you are joining two tables on a column (or columns) with the same name. This avoids ending up with duplicate columns in the output dataframe and is much more succinct, especially when joining on multiple columns |
| Prefer select over drop | Instead of doing a long series of drops<br><br>e.g. df.drop("a").drop("b").drop("c").drop("d").drop("e")<br><br>consider selecting instead<br><br>e.g. df.select("g","h","i","j","k")<br><br>Selecting makes it clear what is on the output dataframe. Furthermore, Spark will not complain if you try and drop a column which doesn't exist (which in general is not desirable).<br><br>Additional benefits of select over drop include being able to choose the order of the columns and also rename them in the same step: df.select($"g".as("moreLogicalName"),...) |
| Column referencing | There are 3 ways of referencing columns:<br><br>• col("columnName")<br>• $"columnName"<br>• 'columnName<br><br>$"columnName" is preferred as it is short, but won't ruin code highlighting if you have code in a text editor such as notepad++. For this reason, avoid 'columnName<br><br>Note that it is very common to do some coding in notepad++ on projects when doing ad-hoc coding work before moving it into an etl/scoring project. |

**External comprehensive coding guidelines:**

http://docs.scala-lang.org/style/

https://github.com/alexandru/scala-best-practices

http://www.lihaoyi.com/post/StrategicScalaStylePracticalTypeSafety.html

https://pavelfatin.com/scala-collections-tips-and-tricks/

https://github.com/alexandru/scala-best-practices

https://twitter.github.io/scala_school/