On this page you can find a list of common issues that we detected while looking at some submissions.

Some of the style issues can be detected by the automated style checker that we also use for the grading process. The style checker, which is based on Scalastyle, can be executed locally in sbt by running the styleCheck task.

Common Issues

#1 Avoid Casts and Type Tests

Never use isInstanceOf or asInstanceOf - there's always a better solution, both for the assignments, and also for any real-world Scala project. If you find yourself wanting to use casts, take a step back and think about what you're trying to achieve. Re-read the assignment instructions and have another look at the corresponding lecture videos.

#2 Indentation

Make sure your code is properly indented, it becomes a lot more readable.

This might seem trivial and not very relevant for our exercises, but imagine yourself in the future being part of a team, working on the same files with other coders: it is very important that everybody respects the style rules to keep the code healthy.

If your editor does not do indentation the way you would like it to have, you should find out how to change its settings. In Scala, the standard is to indent using 2 spaces (no tabs).

#3 Line Length and Whitespace

Make sure the lines are not too long, otherwise your code is very hard to read. Instead of writing very long lines, introduce some local value bindings. Using whitespace uniformly makes your code more readable.

Example (long line, missing spaces):

```
1  if(p(this.head))this.tail.filter0(p, accu.incl(this.head))else this.tail.filter0(p, accu)
```

Better:

```
1  if (p(this.head))
2    this.tail.filter0(p, accu.incl(this.head))
3  else
4    this.tail.filter0(p, accu)
```

Even better (see #4 and #6 below):

```
1  val newAccu =
2    if (p(this.head)) accu.incl(this.head)
3    else accu
4  this.tail.filter0(p, newAccu)
```

#4 Use local Values to simplify complex Expressions

When writing code in functional style, methods are often implemented as a combination of function calls. If such a combined expression grows too big, the code might become hard to understand.

In such cases it is better to store some arguments in a local value before passing them to the function (see #3 above). Make sure that the local value has a meaningful name (see #5 below)!

#5 Choose meaningful Names for Methods and Values

The names of methods, fields and values should be carefully chosen so that the source code is easy to understand. A method name should make it clear what the method does. No, temp is never a good name :-)

A few improvable examples:

```
1  val temp = sortFuntion0(list.head, tweet)   // what does sortFunction0 do?
2  def temp(first: TweetSet, second : TweetSet): TweetSet = ...
3  def un(th: TweetSet,acc: TweetSet): TweetSet = ...
4  val c = if (p(elem)) accu.incl(elem) else accu
5  def loop(accu: Trending, current: TweetSet): Trending = ...
6  def help(ch: Char, L2: List[Char], compteur: Int): (Char, Int) = ...
7  def help2(L: List[(Char, Int)], L2: List[Char]): List[(Char, Int)] = ...
```

#6 Common Subexpressions

You should avoid unnecessary invocations of computation-intensive methods. For example

```
1  this.remove(this.findMin).ascending(t + this.findMin)
```

invokes the this.findMin method twice. If each invocation is expensive (e.g. has to traverse an entire data structure) and does not have a side-effect, you can save one by introducing a local value binding:

```
1  val min = this.findMin
2  this.remove(min).ascending(t + min)
```

This becomes even more important if the function is invoked recursively: in this case the method is not only invoked multiple times, but an exponential number of times.

#7 Don’t Copy-Paste Code!

Copy-pasting code is always a warning sign for bad style! There are many disadvantages:

- The code is longer, it takes more time to understand it
- If the two parts are not identical, but very similar, it is very difficult to spot the differences (see example below)
- Maintaining two copies and making sure that they remain synchronized is very error-prone
- The amount of work required to make changes to the code is multiplied

You should factor out common parts into separate methods instead of copying code around. Example (see also #3 above for another example):

```
1  val googleTweets: TweetSet = TweetReader.allTweets.filter(tweet =>
2    google.exists(word => tweet.text.contains(word)))
3  val appleTweets: TweetSet = TweetReader.allTweets.filter(tweet =>
4    apple.exists(word => tweet.text.contains(word)))
```

This code is better written as follows:

```
def tweetsMentioning(dictionary: List[String]): TweetSet =
  TweetReader.allTweets.filter(tweet =>
    dictionary.exists(word => tweet.text.contains(word)))

val googleTweets = tweetsMentioning(google)
val appleTweets  = tweetsMentioning(apple)
```

#8 Scala doesn't require Semicolons

Semicolons in Scala are only required when writing multiple statements on the same line. Writing unnecessary semicolons should be avoided, for example:

```
def filter(p: Tweet => Boolean): TweetSet = filter0(p, new Empty);
```

#9 Don't submit Code with "print" Statements

You should clean up your code and remove all print or println statements before submitting it. The same will apply once you work for a company and create code that is used in production: the final code should be free of debugging statements.

#10 Avoid using Return

In Scala, you often don't need to use explicit returns because control structures such as if are expressions. For example, in

```
def factorial(n: Int): Int = {
  if (n <= 0) return 1
  else return (n * factorial(n-1))
}
```

the return statements can simply be dropped.

#11 Avoid mutable local Variables

Since this is a course on functional programming, we want you to get used to writing code in a purely functional style, without using side-effecting operations. You can often rewrite code that uses mutable local variables to code with helper functions that take accumulators. Instead of:

```
def fib(n: Int): Int = {
  var a = 0
  var b = 1
  var i = 0
  while (i < n) {
    val prev_a = a
    a = b
    b = prev_a + b
    i = i + 1
  }
  a
}
```

prefer:

```
def fib(n: Int): Int = {
  def fibIter(i: Int, a: Int, b: Int): Int =
    if (i == n) a else fibIter(i+1, b, a+b)
  fibIter(0, 0, 1)
}
```

#12 Eliminate redundant “If” Expressions

Instead of

```
1  if (cond) true else false
```

you can simply write

```
1  cond
```

(Similarly for the negative case).

Other styling issues?

Please post to the forum using the style or styleChecktags and we will augment this style guide with suggestions.

✓ Complete