Objectives

- (Practice) To install and run the TextEdit case study
- (Theory) To explain the text editor case study and the difficult issues it demonstrates

Background

It is assumed that you have experience of using object-oriented programming techniques in e.g. Scala, Java, C#, or similar. It is also assumed that you have prior experience using an IDE such as Eclipse or IntelliJ.

Resources

You should refer to the following resources accessible via Blackboard:

- Topic 2 Lecture videos 2A, 2B, …
- **Topic 2 folder** (zipped) – contains all the code and design documentation for the text editor case study
- External website for Scala doc and other **Learning Resources** (see the folder on Blackboard)

Introduction

This topic is self-study. We assume that you have set up your Scala IDE (either use Eclipse or IntelliJ) and (re-)familiarised yourself with basic Scala syntax (see Topic 1 Getting Started).

The text editor case study is the main activity in this topic. You should install the case study (see below), study the Scala code, run the demo programs, and watch the videos explaining what is going on. These videos are in the Weekly Topics -> Topic 2 folder on Blackboard and are numbered 2A, 2B, etc.

The purpose of the case study is to demonstrate to you some straightforward problems that arise when programming in traditional and object-oriented styles. The cause of the problems will be identified, and it will be argued that these issues arise frequently and naturally in traditional programming. We ask whether it is possible to avoid these problems completely by using a different programming paradigm such as using a functional style instead.

Activities

2.1 Check your Scala IDE environment

Open your Scala IDE and check everything is set up ok (see Topic 1 Getting Started). Ensure that all code currently in your Scala project (e.g. example code and exercises you have been using as part of Topic 1) has no syntax errors. It is dangerous to add code to a project if anything existing in the project has syntax errors. You should always leave your session with no syntax errors in your Scala project. We therefore promote an incremental approach to code development – entering classes one at a time rather than all at once. This will also help you to focus on each piece of code as you enter it.

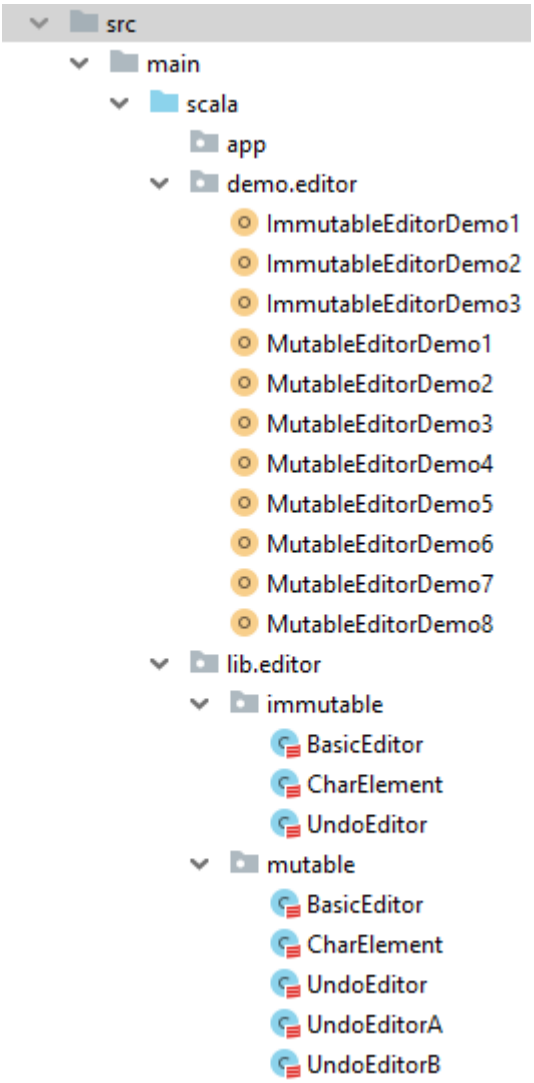
2.2 Install and run the text editor case study

Within your (Eclipse)
src
folder or (IntelliJ)
src->main->scala
folder you should have the following package structures:

demo->editor
lib->editor->immutable
lib->editor->mutable

Download the file **topic-2-folder.zip** and uncompress this in your own area. You will see that the folder structure mirrors that which you should set up in the IDE and enables you to see easily where the Scala source files should be copied to.

The image to the right shows the installed programs displayed within IntelliJ. If you are using Eclipse then the **main->scala** subfolders will be absent from **src**.



Note that it is important to get the package structures correct and mapping onto the folder structures. When you are creating the packages within the IDE you need to be sure to create a new **package** and not a new **source folder** or **folder** by mistake.

The use of packages to organise code in a hierarchy is common across JVM languages and will likely be familiar to you if you have coded previously in Java.

Once installed you can run the demo programs to see what happens. However, you will understand these better once you have followed the videos on Blackboard that accompany this topic.

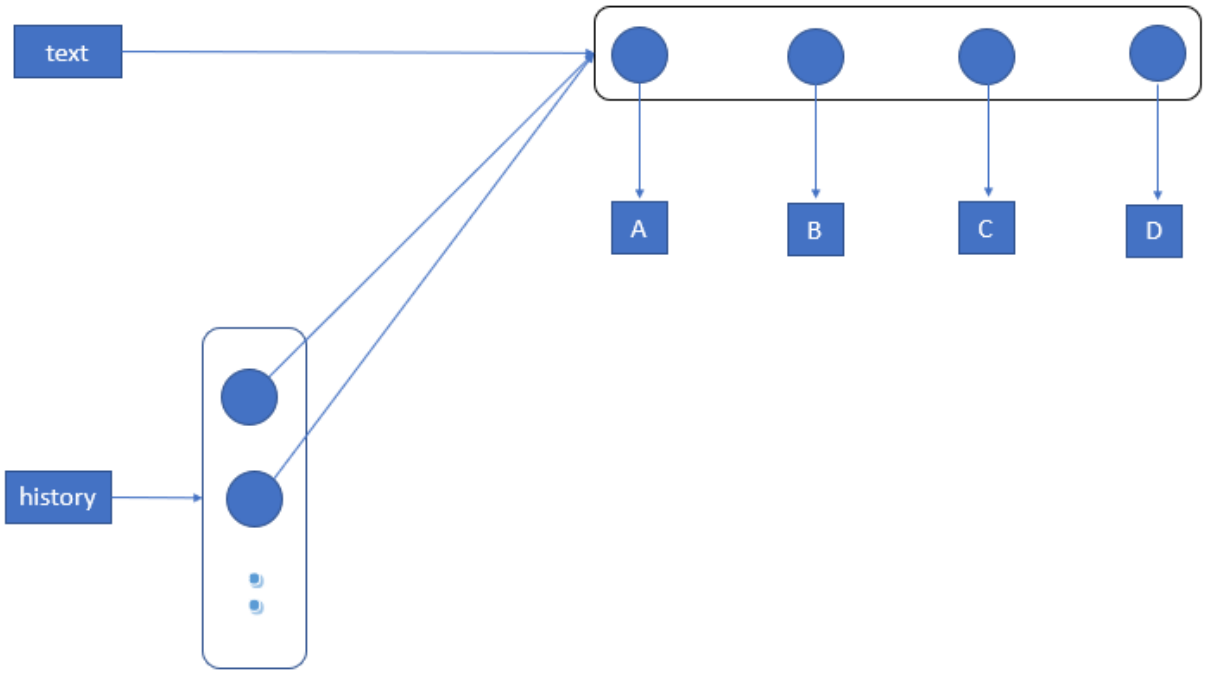
*It is important to stress at this stage: the text editor case study is **not** written in a functional style. It has been written using an object-oriented style and makes extensive use of mutable data structures. We begin with this to provide context and to make a contrast. Your (and our) future programs on this module will look nothing like this!*

2.3 Watch the videos and study the code

Watch the videos on Blackboard that accompany this case study. They are numbered 2A, 2B, etc.

While the videos are playing you should also refer back to the code in the IDE. You may for example pause the video from time to time to ensure that your understanding of the code matches the points that are being made in the videos. Some of the points that are highlighted in the videos are very subtle and it is worth your while investing some effort in following this case study through completely to understand fully what is being said.

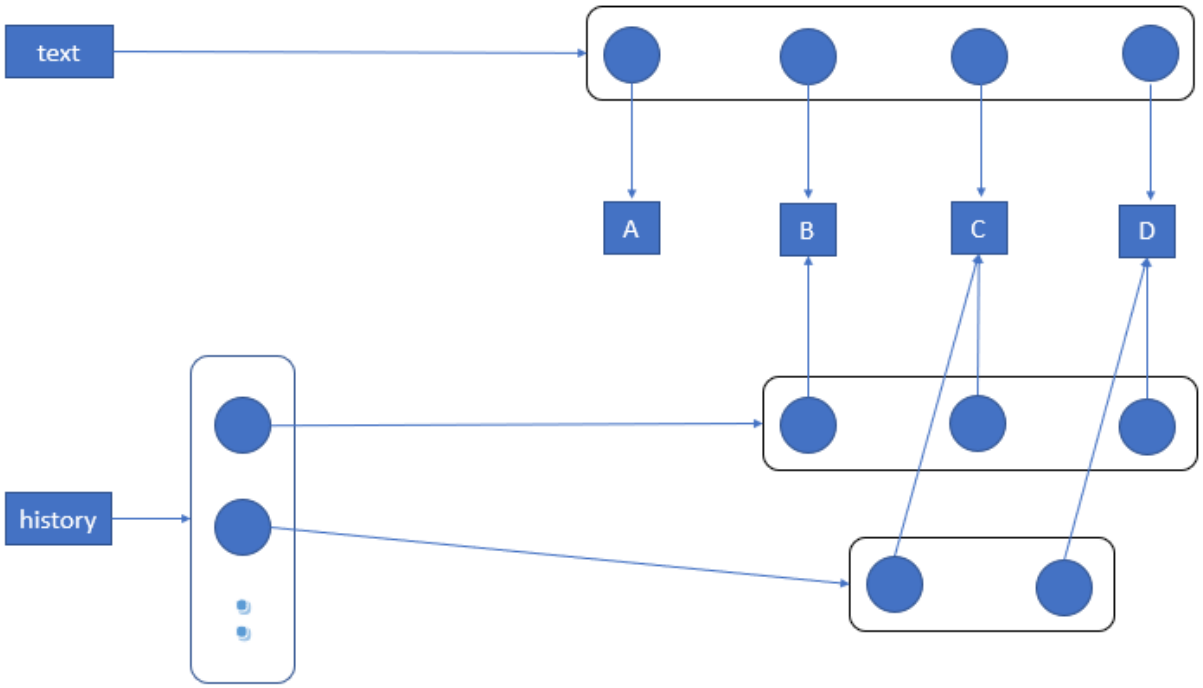
Figure 1. Structure sharing



The picture shows that pushing **text** onto the stack merely pushes copies of the reference (pointer) to the ListBuffer. The items on the stack all refer to the same structure. If the ListBuffer is modified via any of the references then these changes will be visible via all references simultaneously. This is known as structure sharing.

If the purpose of the stack is to store previous versions of the state of the ListBuffer between modifications then this design is completely broken. This is the effect of **sharing mutable state**.

Figure 2. Structure sharing at next level

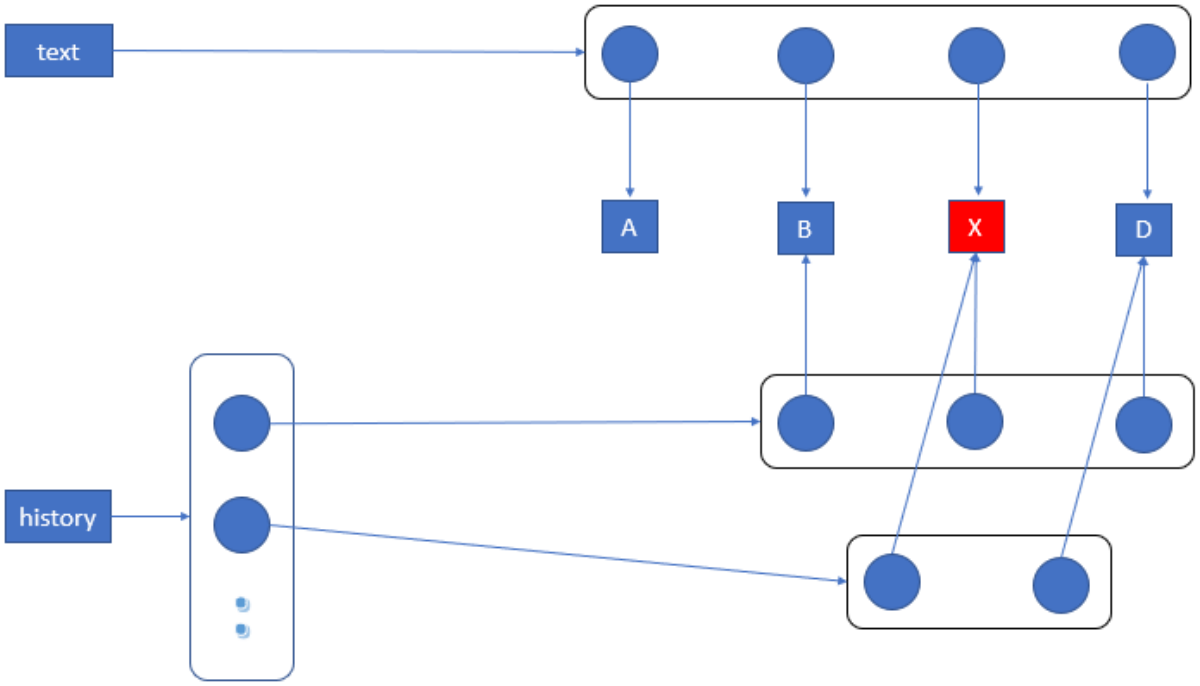


The picture here shows an attempt to improve upon the previous design. Instead of pushing copies of **text** onto the stack the data structure itself is *cloned* each time. Thus the items on the stack now refer to separate ListBuffer instances in memory. In the example we can see that the current text buffer has four characters and, previously, it had three, and before that it had two. We see the evolution of the text buffer as characters B and then A are inserted at the front of the string.

However, the problems of shared mutable state remain but at a slightly deeper level. *Cloning* the ListBuffer does indeed make a copy of the data structure. However, its contents are references to objects (Character Elements) and it is these references that are *cloned* – not the character elements themselves.

If we were to modify the third character in the text buffer ('C') and make it into 'X', say, then the historical states would appear to show 'X' in this position – all memory of 'C' would have been changed. This is another undesirable side-effect due to **sharing mutable state**.

Figure 3. Modifying one of the characters



The picture here shows the scenario described in Figure 2. The third element in the **text** buffer has been changed from a 'C' to a 'X'. However, due to the shared mutable state, the previous "copies" on the history stack now show that 'X' was always in that position in the text string, and never 'C'.

The point here is that in a model which has collections of mutable objects are susceptible to all manner of unwanted or unpredicted side-effects. This behaviour becomes apparent when you see errors in the program output (if you are lucky). If the shared state is quite deeply embedded then these programs can be extremely difficult to debug.

If the cause of the problem is shared mutable state, we will propose writing all our programs without it! The **Functional Programming** model we will adopt in the remainder of this module will use only **immutable objects**. It is perfectly safe to structure share with immutable objects because side-effects (such as those we have seen) are not possible.

2.4 Make your own notes

You could do this below in your copy of this document. Some of the issues you will be aware of already from previous experience. In this case you might just wish to note down a familiar situation. It is likely, however, that some of the issues are novel, or you may not have considered them deeply before. In this case you should really write down your understanding of the issues and their implications. This will help you in all your programming – not just for this module.

- Put your own notes here as you follow the videos and study the code…
-