Rock the JVM!

Scala homework #1: a simple file system

You should start this homework after you have gone through the first two chapters in the course: Scala Basics and Object-Oriented Programming. The corresponding Skill Vaults will be useful here, but are not mandatory.

In this homework, we'll be designing a simulation of a Linux-type file system. You've surely used the command line before – regardless, here's what we are aiming for.

This application will run in a console – preferably in your IDE's console – and it will contain an imaginary file system. So no actual files and folders on your computer, but *representations* of them in memory. You'll be greeted with a "command line" which will look like the following:

```
/home/student $
```

The current working directory

You write here

And just like a normal command line, we'll want to support the following commands:

- **mkdir** – creates a new directory as a child of the current working directory. Receives one argument: the name of the directory to create.
- **cd** – changes the current working directory. Receives one argument, which can be an *absolute* or *relative* path to the current working directory. We're aiming to support relative directories such as "."(dot) and ".." (dot-dot).
- **ls** – displays the children of the current working directory. Receives no arguments.
- **rm** – removes an entry (file or folder). Receives one argument: the name of the entry to remove.
- **echo** – displays something to console. Can also be used to write to files. The syntax

  ```
  /home/student $ echo hello world > file.txt
  ```

  will write the contents "hello world" to a file called file.txt, regardless of whether the file exists (will overwrite) or not (will create a new file). If instead of the token ">" (greater than) we use ">>", the command will have an *append* behavior instead

of overwrite. This command will be tricky, as it can receive one or multiple arguments and has a lot of edge cases to consider.
- **cat** – will display the contents of a file to the console. Receives one argument: the name of the file to open.
- **touch** – will create an empty file. Receives one argument: the name of the file to create.
- **mv** – will move an entry (file or directory) to a new location. Receives two arguments: the name of the entry (as an absolute or relative path) and the name of the destination directory (as an absolute or relative path).

The purpose of this homework is to have you accustomed to the object-oriented aspects of the Scala language and get you comfortable designing and manipulating *immutable data structures*.

If you have any trouble whatsoever, feel free to ask in the course forum. We will respond promptly. We also encourage you to help fellow students, as you'll get tremendous experience with a variety of problems you otherwise wouldn't want to encounter in your production work for the first time…

This is a choose-your-destiny-style assignment. We're including indications on how to complete this homework (coming up next), but the design and implementation choice is ultimately yours – as is the difficulty. We'll include some follow-up tasks that you can challenge yourself with. The harder the challenge, the greater the rewards.

We're now going to go through an example of how to reason and design such an application.

Prologue

We'd recommend you start with making something that actually runs and then build up and expand. Of course, thinking through basic design is key - in this way you'll need a minimum of refactoring whenever you decide you want more features or implement more intricate functionalities.

Before we start discussing the implementation, we'll assume you know how to implement your own list, in the style we talked about in the List creation demo. If you prefer, you can use Scala's List predefined type. We recommend you use your own List version - you might spot problems with your code, will give you more experience and confidence using your code, and this whole project will feel 100% your work.

Having that settled, let's go.

The big ideas

So in our imaginary file system, we'll have two major types of elements:

- directory entries (files and folders) and
- commands.

Let's ignore commands for now and set up the first type of entities. It's probably something you're also thinking: organize files and folders under the same class hierarchy - create a DirEntry abstract class, then a Directory and File subclasses.

We're going to suggest the following structure:

```
abstract class DirEntry(val parentPath: String, val name: String)

class Directory(
     override val parentPath: String,
     override val name: String,
     val contents: List[DirEntry])
extends DirEntry(parentPath, name)

class File(
     override val parentPath: String,
     override val name: String,
     val contents: String)
extends DirEntry(parentPath, name)
```

So every directory entry will have a parent path and a name. Directories have an additional list of children, and Files have contents as text.

If you thought of a slightly different design where a directory entry might have a link to its parent Directory instead of a parent path (a String), the idea is intuitive, but problematic [1]. You can experiment.

The key idea that we want to emphasize right at the beginning (and multiple times throughout this homework) is that **whenever we modify the folder structure or otherwise change the contents of our file system in any way, be it by deleting a directory or changing the contents of a file, we will be returning a new folder structure.** That is, our file system will be an immutable structure. Of course, as we rebuild and re-create the new file system contents, we will be reusing most of our currently available data so as not to be spatially wasteful. But immutability is a key consideration in our design.

Now, because after every command our file system's state is very likely to change, we'll store that in a special class:

```
class State(val root: Directory, val wd: Directory, val output: String)
```

in which root is the file system's parent directory, wd is the current working directory, and output is the message obtained from the previous command. This design is based on the idea that when you write

```
/home/student $ mkdir rockthejvm
```

then the current state of the world is 1) the root folder and 2) the working directory instance that you can find by going from the root to the home/student subfolders. After you actually run your command, the *new* state of the world will be

- a new root folder because, remember, our file system is immutable and thus modifying it by creating a new directory will return a new one
- a different Directory instance as working directory - obtained from the new root and going to /home/student: the path is the same but actual Directory instances (in memory) will likely differ
- a possible command output - for example, if there is already a file or folder in /home/student named 'rockthejvm', this will be something like *"Error: entry 'rockthejvm' is already present in /home/student"*

Good, so we have file system elements and the state of the world. Let's talk about commands for a second. We want to implement multiple commands so it makes sense to create an object-oriented type hierarchy here as well. Let's consider, then, the following

```
trait Command
```

This trait will have the property that it can run on a State of the world - mostly on the root and working directory, because commands don't really care about previous commands... unless you want to extend this homework[2].

So it makes sense for a command to have a method

```
def apply(state: State): State
```

which means it transforms a State into another State. If you think another second about it, your Scala bulb will probably light up: a Command will behave like a function from State to State!

All commands will extend this trait. We'll go through an example in a moment. Let's piece these things together and have a skeleton application running.

So we have:

1. File system elements

We described these before:

```
abstract class DirEntry(val parentPath: String, val name: String) { }
```

They can be Directories:

```
class Directory(
      override val parentPath: String,
      override val name: String,
      val contents: List[DirEntry]) extends DirEntry(parentPath, name) {
  // TODO your code here
}

object Directory {
  val SEPARATOR: String = "/"
  val ROOT_PATH: String = "/"
  def empty(parentPath: String, name: String) =
    new Directory(parentPath, name, List())
  def newRoot: Directory =
    Directory.empty("", "")
}
```

or Files:

```
class File(
      override val parentPath: String,
      override val name: String,
      val contents: String) extends DirEntry(parentPath, name) {
}
```

Notice how in this example we have a small companion object for Directory which has some utility functions and "static" values such as the standard forward-slash separator.

2. The States of the world

```
class State(val root: Directory, val wd: Directory, val output: String) {
  def showShell: Unit =
    print(State.SHELL_TOKEN)

  def show: Unit = {
    println(output)
    showShell
  }
  // builds a NEW State!
  def withMessage(message: String): State =
    new State(root, wd, message)
  }

object State {
  val SHELL_TOKEN = "$ "
  def apply(root: Directory, wd: Directory, output: String = "") =
    new State(root, wd, output)
}
```

We highly encourage you to make use of the companion object style of writing Scala code - it'll prove very useful as you get more experienced and, with time, will become second nature to you. In this case, our companion object for State contains an apply factory method that builds a State out of three arguments, one of which has a default value (the empty output).

3. Commands

```
trait Command {
  def apply(state: State): State
}
```

Of course, with a companion object - we'll be creating a variety of commands after parsing our user's input, so it makes sense to write something like

```
object Command {
  def unknownCommand(name: String) =
    new Command {
      override def apply(state: State): State =
        state.withMessage(name + ": command not found")
    }
  def from(input: String): Command = {
    val tokens = input.trim.split(" ")
    unknownCommand(tokens(0))
  }
}
```

Notice:

- The “from” method is our factory method. We called it "from", you could call it whatever you want, including "apply"!
- unknownCommand creates an instance of an anonymous class which inherits from Command, and whose implementation creates a new State out of the current one, but with the message *"(your command): command not found"*.

4. An application entry point

```
object Filesystem extends App {
  val firstRoot = Directory.newRoot
  var state = State(firstRoot, firstRoot)
  val scanner = new Scanner(System.in)
  while(true) {
    state.show
    state = Command.from(scanner.nextLine()).apply(state)
  }
}
```

This is also fairly simple, but likely deservers some explanation. We first create our file system's root and the first state of the world, something like a let-there-be-light state. Then in an infinite loop we're showing our current state, then we're changing the state by passing the user's input to Command's factory method. That creates a Command object (which will in fact be an implementation of Mkdir, or Cd etc), and we're calling the *apply* method on our current state and return a (potentially) new

one. Because we have to update our state of the world, we are forced to make "state" a *var*. That will be our *only var* in the code.

You will find this code scaffolding on the virtual machine's desktop. You can alternatively download an archive with a running project in the course materials.

[1] An intuitive idea – often used in the mutable object-oriented world) – is to hold instance references everywhere. In our case, for DirEntries, it might make sense to have something like

```
abstract class DirEntry(val parentPath: Directory, val name: String) { }
```

instead of our suggested approach with parent paths as Strings.

But remember, our key consideration in the implementation of this assignment is the *immutability* of our folder structure. This is not negotiable – our apologies that we're forcing this onto you at this stage but you'll see the benefit of immutability when you get onto more complex projects.

Keeping immutability in mind, you'll discover… interesting problems to solve. We highly encourage you to experiment with this design, as you'll learn some valuable lessons along the way and will make you a better software developer and architect.

[2] You can extend this homework until you have your own virtual Linux file system in memory. Really, the sky is the limit. Some suggestions:

1. Add another type of entities to the game: users and permissions. Have a superuser who can access everything and run any command. Implement a "sudo" command, which will allow a non-superuser to execute only one command as a superuser. You will likely need the following commands:
    a. sudo (command)
    b. chmod (directory entry) (permissions in a format of your choice)
    c. adduser
    d. deluser
2. Extend State to hold not just the last command's output, but the entire command that was run. Implement special Linux shortcut commands like "!!" (the last run command).

3. Extend commands to return an exit code, which will mean success or failure. Create chained commands that can be run as here

```
/home/student $ command1 && command2
```

   with the following operators:
   a. && (double ampersand) - runs command2 *only if* command1 was successful
   b. ; (semicolon) – runs command2 after command1
4. Add an exit command, which will terminate the application gracefully.