Homework 8: Implicits

In this assignment, you'll first learn how to use two modern Java APIs on your own and then make them more "Scala-like" using implicits. Before you start directly working on the assignment, you will have to read the API documentation and online tutorials to learn how the APIs work. We encourage you to freely use any resource from the web for this assignment.

Note: This assignment can be done using implicit classes exclusively. Scala also supports implicit arguments and implicit functions, but you don't need them for this assignment.

1 Preliminaries

You should create a directory-tree that looks like this:

```
./implicits
├── project
│   └── plugins.sbt
└── src
    ├── main
    │   └── scala ..... Your solution goes here
    └── test
        └── scala ..... Yours tests go here
```

The `project/plugins.sbt` file must have exactly this line:

```
addSbtPlugin("edu.umass.cs" % "cmpsci220" % "3.0.1")
```

2 Java NIO

The java.nio.file classes are a new (since 2011) API for working with files and directories that is a lot easier to use than the legacy API, which dates back to the early days of Java. For example, the static methods `Files.readAllBytes` and `Files.write` let you read and write to files with just one line of code.

Moreover, the `java.nio.file.Paths` object is easier to use than `java.io.File`. For example, to get a reference to the directory `"a/b/c"`, we can simply write:

```
import java.nio.file.Paths
Paths.get("a").resolve("b").resolve("c")
```

The "method call chain" above is lot easier to write and read than the nested constructors in the legacy code below:

```
import java.io.File
new File(new File("a", "b"), "c")
```

However, we ought to be able to do better in Scala. It would be really convenient if we could simply create a `java.nio.file.Path` object by writing:

```
"a" / "b" / "c"
```

The expression above should be the same as `Paths.get("a").resolve("b").resolve("c")`. The first part of this assignment is to "Scala-ize" the Java NIO file API in this manner.

2.1 Programming Task

Create an object called `PathImplicits` (in `src/main/scala`). When a block of code begins with `import PathImplicits._`, we should be able to write the following:

1. We should be able to construct paths by using the slash-operator to separate strings. For example, the expression `"usr" / "bin" / "scala"` should be equivalent to the expression `Paths.get("usr", "bin", "scala")`.
2. Similarly, given two paths, we should be able to join them using the slash-operator. For example, if `p1` is `"usr" / "local"` and `p2` is `"bin" / "scala"` then `p1 / p2` should be equivalent to `Paths.get("usr", "local", "bin", "scala")`.
3. Paths should have a `write` method to create files, where `write` takes a string to write to the file. For example, the following program should create a file called `greeting.txt` with just the line `Hello, world`:

   ```
   val p = Paths.get("greeting.txt")
   p.write("Hello, world\n")
   ```

4. Paths should also have a `read` method that returns the contents of the file as a string. For example, after creating the file above, the expression `p.read()` should produce `"Hello, world\n"`.
5. Finally, paths should have an `append` method that appends to the end of a file. For example, after evaluating `p.append("Hello, again")`, the file `greeting.txt` should contain:

   ```
   Hello, world
   Hello, again
   ```

   For convenience, the append method should create a new file if no file exists.

3 The Java 8 Time API

Java 8, which was released in 2014, introduced a new API for working with dates and times, which is much easier to use than the old `java.util.Calendar` API (from 1997) and the `java.util.Date` API (from 1996 and deprecated). Notably, the new `java.time` API uses immutable objects exclusively, unlike the older APIs, which are exemplars of bad imperative design. For this assignment, we are going to focus on the java.time.LocalDate class, which just represents a date, without a timezone or time of day.

For example, we can create a date that represents Jan 31 2016 as follows:

```
import java.time.LocalDate
val date = LocalDate.of(2016, 1, 31)
```

We can add days to a date as follows:

```
val feb2 = date.plusDays(2)
```

Note that the expression above does not modify the original `date` object, but produces a new object. In fact, the `LocalDate` class is immutable and the API is designed in a functional way. However, we can make it even easier to use. It would be more convenient if we could write `31 jan 2016` in Scala code to mean `LocalDate.of(2016, 1, 31)` and `(31 jan 2016) + 2.days` to mean `2 feb 2016`, and so on.

3.1 Programming Task

Create an object called `DateImplicits` (in `src/main/scala`). When a block of code begins with `import DateImplicits._`, we should be able to write the following:

6. We should be able to write `15.jan`, `4.feb`, `20.dec`, and so on to create `LocalDate` objects that represent dates in the current year. (i.e., the year in which the code is running.) **This kind of code should work for all months of the year, with the usual three-letter abbreviations of month names.**
7. To create `LocalDate` objects that represent dates in other years, we should be able to write `15.oct(1989)`. Note that this is the same as writing `15 oct 1989`.
8. **(Tricky)**. If `x` is a `LocalDate`, we should be able to add days, months, and years in the following way: `x + 10.days`, `x + 2.months`, and `x + 5.years`.

We should be able to combine both notations. For example, `(31 jan 2016) + 2.days` should be equal to `LocalDate.of(2016, 2, 2)`. Note that the parentheses around the date are necessary because `1 jan 2016 + 2.days` is interpreted as `1 jan (2016 + 2.days)`.

4 Hand In

From the `sbt` console, run the command `submit`. The command will create a file called `submission.tar.gz` in your assignment directory. Submit this file using Moodle.

For example, if the command runs successfully, you will see output similar to this:

```
Created submission.tar.gz. Upload this file to Moodle.
[success] Total time: 0 s, completed Jan 17, 2016 12:55:55 PM
```

Note: The command will not allow you to submit code that does not compile. If your code doesn't compile, you will receive no credit for the assignment.