Imperative Programming Part 3
Problem Sheet 3

Peter Jeavons*

Trinity Term 2019

1. **[Programming]** Write a simple graphical user interface for the `WordPaths` program developed in Imperative Programming Part 2.

   One simple way to do this is to follow the pattern of the `TemperatureConverter` program discussed in the lectures and extend `SimpleSwingApplication`. Your program should provide text fields to allow the user to input the start and target words, and some way to initiate a call to `WordPaths.findPath` and display the resulting path (if found).

2. The Scala collection framework contains a generic trait `Set[A]` for immutable sets. The documentation states that:

   > To implement a concrete set, you need to provide implementations of the following methods:

   ```
   def contains(key: A): Boolean
   def iterator: Iterator[A]
   def +(elem: A): This
   def -(elem: A): This
   ```

   > If you wish that methods like take, drop, filter return the same kind of set, you should also override:

   ```
   def empty: This
   ```

   Implement a concrete class `MySet[T](elements:Set[T])` that extends `Set[T]`.

3. In this question, we will use the following trait, which specifies that a Scala type `T` is equipped with a partial order `<=` and that corresponding least upper bounds can be computed by `lub`:

   ```
   trait PartialOrder[T] {
     def <=(that: T): Boolean // checks this <= that. Partial order on T.
     def lub(that: T): T // returns the least upper bound of this and that.
   }
   ```

   (a) One standard way of defining a partial order $\leq_U$ over sets is to use the subset relation:

   $$X_1 \leq_U X_2 \iff X_1 \subseteq X_2.$$

   Enhance your concrete set class from Question 2 so that it extends `PartialOrder[MySet[T]]` and so implements ordering by inclusion for finite sets.

*Based on earlier material by Mike Spivey, Joe Pitt-Francis and Milos Nikolic.

(b) Let X be a set with a partial order $\leq$ on the elements. We say that a subset X_0 of X is *upward closed* if

$$\forall x, y \in X.\ (x \in X_0 \land x \leq y) \implies y \in X_0.$$

The *upward closure* of a subset X_0 of X is defined to be the set Y_0 given by:

$$Y_0 = \{y \in X \mid \exists x \in X_0.\ x \leq y\}.$$

Define a generic class `UpSet` so that for any Scala set `s` representing a finite set X_0 with elements of type T, calling `new UpSet(s)` should create an object representing the upward closure of X_0. Your class should provide just the following methods for set membership and set intersection:

```
def contains(x: T): Boolean
def intersection(that: UpSet[T]): UpSet[T]
```

Note that the upward closure of a finite set X_0 may be infinite, so you may find it helpful to represent the upward closure by its *minimal elements*, that is, the elements $x \in X_0$ such that there does not exist $y \in X_0$ with $y < x$.

(c) Enhance your `UpSet` class so that it extends `PartialOrder[UpSet[T]]` and so implements ordering by inclusion for (possibly infinite) upward closed sets.

4. A bag B of elements of type T can be represented by a function

$$f_B : T \to \mathbb{N}$$

Using this observation, develop a generic immutable bag class by completing the following code skeleton:

```
class Bag[T](...) {
  def add(x:T): Bag[T] = ...
  def remove(x:T): Bag[T] = ...
  def count(x:T): Int = ...

  def union(that:Bag[T]): Bag[T] = ...
}
```

The following code provides some test cases:

```
val b0: Bag[Any] = (new Bag((x) => List(0, 0.0, "zero", 0).count(x==_)))
val b1: Bag[Any] = new Bag((x) => List(1, 1.1, "one", 1).count(x==_))
val b2: Bag[Int] = new Bag((x) => List(2,3).count(x==_))

val b3: Bag[Any] = b0 union b1

println(b0.add("zero").count("zero") + b1.count(1.1) + b2.count(2))
                                                    // Should print 4
println(b3.remove(0).count(0)+","+b3.count(1)+","+b3.count(2))
                                                    // Should print 2,2,0
```

The definition of `Bag[T]` is *invariant*, so, for example, `b3 union b2` will give a type mismatch error.

What changes are needed to the class definition to make `Bag[T]` *contravariant*? With these changes, what is the type of `b3 union b2`, and what items does this bag contain?

5. Explain why generic mutable collection classes such as `scala.collection.mutable.HashSet[T]` are defined to be *invariant*, rather than *covariant* or *contravariant*. Wouldn't defining them to be covariant or contravariant provide more flexibility?

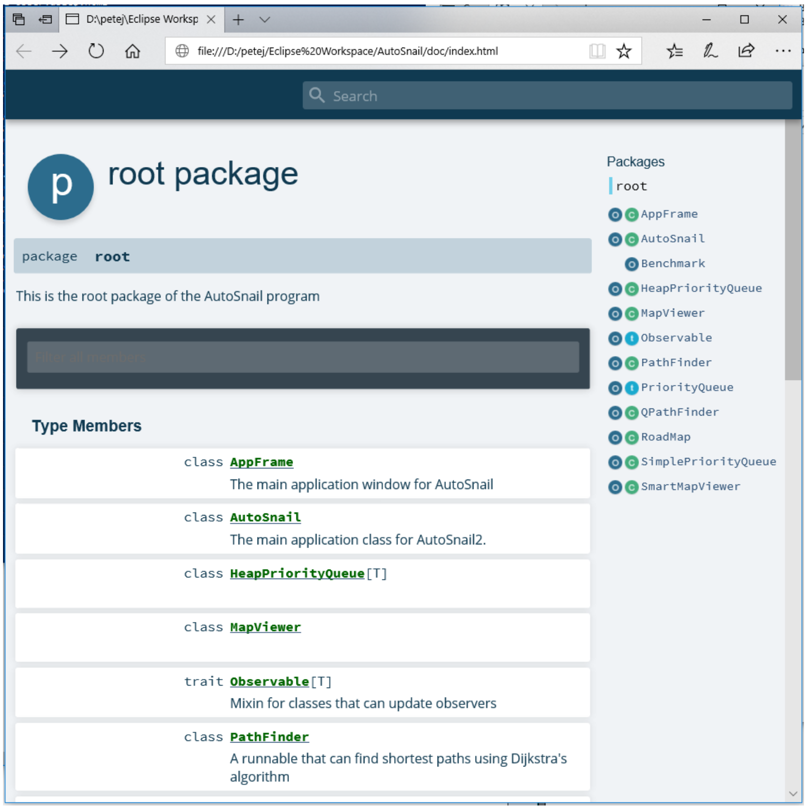

Figure 1: Documentation for the AutoSnail case study

6. Create a full set of documentation for the AutoSnail case study, looking something like Figure 1. (NB: all of this documentation can be (and should be) created using a single command)

   Use this documentation to find a method in the `AppFrame` class that is inherited from `scala.swing.UIElement`.

7. What are the main components of the *Façade* design pattern, and why is it used?

   Which classes in *AutoSnail* implement this design pattern and what roles do they play?

8. [**Programming**] Add to *AutoSnail* some way for the user to change the colour of the sea.

   (You may find it helpful to use the `scala.swing.ColorChooser` class, whose companion object has a `showDialog` method, which offers the user a choice of colours.)

9. (Optional) [**Programming**] In the version of *AutoSnail* we are using, mouse coordinates are converted into the identity of the town they point to by making a linear search of the list of towns. This is done when you click on a town to select it, but also when the mouse hovers over the map, in order to show a tooltip with the name of the town. The program considers that the mouse is pointing at a town when it is within 5 pixels of the town's centre. All this searching of the list of towns may consume large amounts of processor time. Identify a data structure that will substantially speed up the translation from mouse coordinates to towns, and explore whether it has a noticeable impact on performance.