IP Lecture 15: Programming with Abstract Datatypes

Joe Pitt-Francis

—with thanks to Gavin Lowe—

Reading: Chapter 3 and Section 7.3 of Programming in Scala.

Abstract datatypes

The Scala application programming interface (API) includes many useful abstract datatypes, such as various types of sequence, sets, and mappings.

We can program with these, treating them as mathematical objects.

Elsewhere we have seen how some of these abstract datatypes are implemented: but we can use them without knowing anything about the internal implementation. This is known as data abstraction.

Lewis Carroll's word path game

A popular word game "Doublets", invented by Lewis Carroll, is to find a path of words, each differing from the previous in a single letter, linking two given words. Carroll used the example of finding such a path linking grass to green, and gave the solution grass, crass, cress, tress, trees, frees, freed, greed, green, although in fact there are shorter solutions.

In this lecture we will write a program to find such paths.

The `List` type

Scala has a class of lists, very similar to (and based on) Haskell lists.

The type of lists holding data of type `A` is written as `List[A]` (recall that Scala uses square brackets for parametric polymorphism).

A list holding values `a`, `b`, `c`, `...`, `z` can be written as `List[A](a, b, c, ..., z)`; in particular, the empty list can be written as `List[A]()`. The type parameter can be omitted for non-empty lists.

If `xs` is a list, then `xs.head` and `xs.tail` are its head and tail.

`x::xs` represents the value `x` consed onto the front of `xs`.

Lists also have lots of operations like `map` and `filter`; see the API documentation[a].

[a] Google for "Scala API"; also see `http://dcsobral.blogspot.com/2011/12/using-scala-api-documentation.html`.

The `List` type

Note that lists are immutable: you can't change a list once you've created it; however, you can construct new lists from the first list; and if a variable references a list, you can change the variable to reference a different list.

Paths as lists

In our word-path program, we'll represent a path as a list of `Strings`:

```
type Path = List[String]
```

for example

```
List("grass", "crass", "cress", "tress", "trees")
```

For later, let's write a function to print a path

```
/** Print the path, separating the elements with commas
  * Pre: path is non-empty */
def printPath(path: Path) = {
  print(path.head)
  for(w <- path.tail) print(", "+w)
  println
}
```

Neighbours

In the word paths program, we want to define a function

```
def neighbours(w: String) : Path = ...
```

that returns all the words that differ from `w` in precisely one position.

Recall that earlier we implemented a dictionary. We can initialise a dictionary by, e.g.,

```
val dict = new Dictionary("knuth_words")
```

and look up a word `w1` by

```
dict.isWord(w1)
```

Dictionary

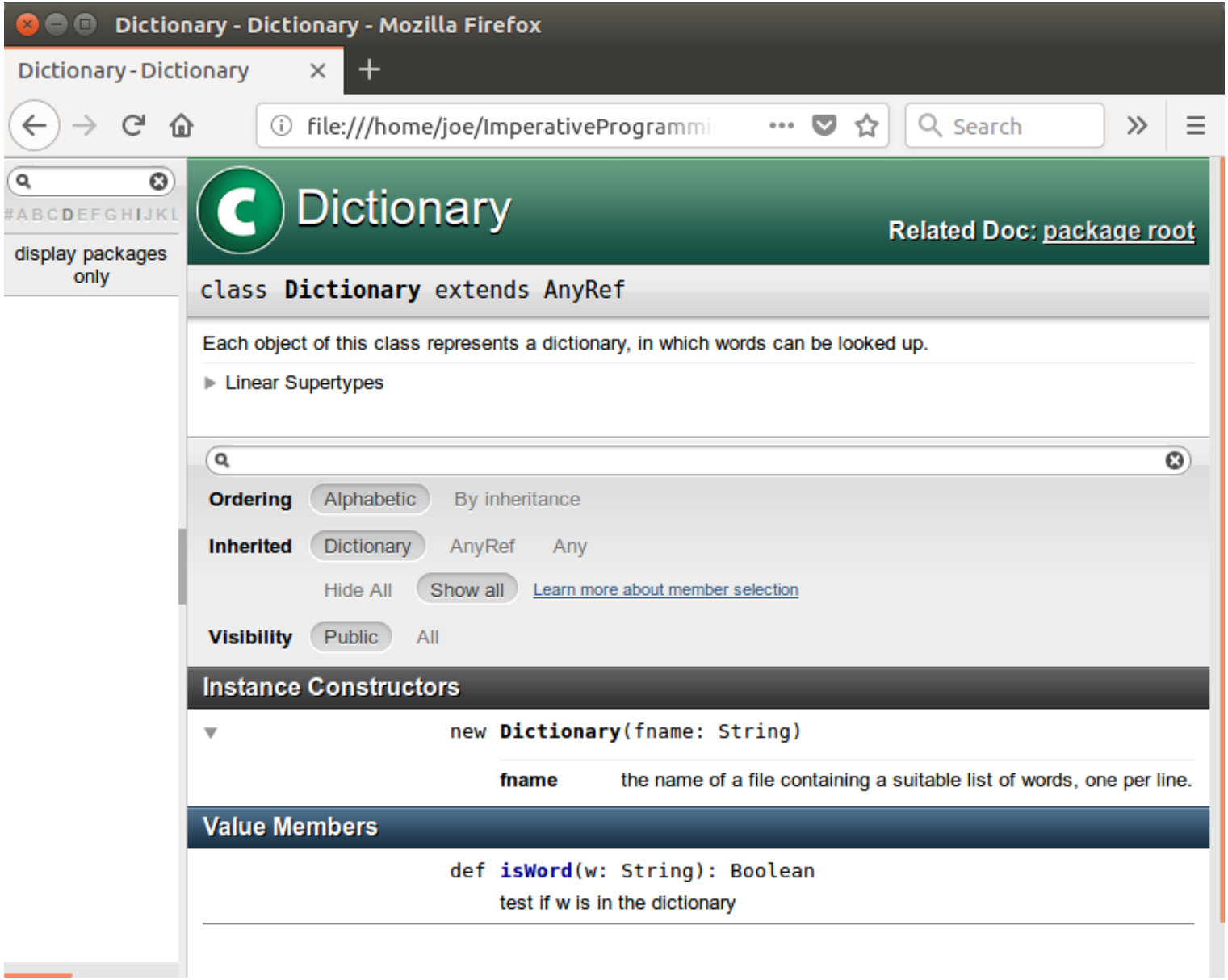
```
/** Each object of this class represents a dictionary, in which
  * words can be looked up.
  * @param fname the name of a file containing a suitable list
  * of words, one per line. */

class Dictionary(fname: String){
  /** A Set object holding the words */
  private val words = new scala.collection.mutable.HashSet[String]

  /** Initialise dictionary from fname */
  private def initDict(fname: String) = ...
  // Initialise the dictionary
  initDict(fname)

  /** test if w is in the dictionary */
  def isWord(w: String) : Boolean = words.contains(w)
}
```

scaladoc

We can generate HTML documentation of the `Dictionary` class by typing:

```
> scaladoc Dictionary.scala
```

Type `man scaladoc` for more options. See `https://wiki.scala-lang.org/display/SW/Syntax` for a summary of syntax, and `http://docs.scala-lang.org/style/scaladoc.html` for a style guide.


Dictionary - Dictionary - Mozilla Firefox
Dictionary - Dictionary
file:///home/joe/ImperativeProgrammi
Search
#ABCDEFGHIJKL
display packages only
Dictionary
Related Doc: package root
class Dictionary extends AnyRef
Each object of this class represents a dictionary, in which words can be looked up.
Linear Supertypes
Ordering Alphabetic By inheritance
Inherited Dictionary AnyRef Any
Hide All Show all Learn more about member selection
Visibility Public All
Instance Constructors
new Dictionary(fname: String)
fname the name of a file containing a suitable list of words, one per line.
Value Members
def isWord(w: String): Boolean
test if w is in the dictionary

Neighbours

Consider the word `w1` that is formed from `w` by replacing the `i`th character by `c`. We could define `w1` by

```
val w1 = w.take(i) + c + w.drop(i+1)
```

or

```
val w1 = w.patch(i, List(c), 1)
```

We want to consider all such `w1` (for all suitable values of `i` and `c`), provided `c` is different from the current `i`th character of `w` (`c != w(i)`) and that `w1` is in the dictionary (`dict.isWord(w1)`).

Neighbours

We can define the `neighbours` function as follows, building up the neighbours we have found in a list `result`:

```
def neighbours(w: String) : Path = {
  var result = List[String]() // build up the result
  for(i <- 0 until w.length; c <- 'a' to 'z')
    if(c != w(i)){
      val w1 = w.patch(i,List(c),1) // replace ith character
                                    // of w with c
      if(dict.isWord(w1)) result = w1 :: result
    }
  result
}
```

Searching for a solution

We want a function

```
def findPath(start: String, target: String)
```

That searches for a path from `start` to `target`. Clearly, if the search succeeds, the function should return the path that is found. But what should it return if no path is found?

Recall the Haskell `Maybe` type

```
data Maybe a = Just a | Nothing
```

Scala has a similar type `Option[A]`, with subtypes `Some[A]` and `None`.

We therefore have `findPath` return a result of type `Option[Path]`:

```
def findPath(start: String, target: String): Option[Path]
```

Breadth-first search

We're now ready to consider how to solve the word path puzzle, to find a shortest path from a starting word `start` to a target word `target`, where each word in the path is a neighbour of the previous.

We start by considering the neighbours of `start`. If any equals `target` we are done.

We then consider the neighbours of the neighbours of `start`. If any equals `target` we are done.

We then consider the neighbours of the neighbours of the neighbours of `start`, and so on.

Put another way, we consider the words reachable in one step from `start`; then the words reachable in two steps from `start`; and so on.

This is a breadth-first search.

Queues

In order to implement the breadth-first search, we will use a queue: a sequence-like data structure that allows elements to be accessed in a first-in first-out (FIFO) order.

The class `scala.collection.mutable.Queue[A]` represents mutable queues that can hold data of type `A`. If `q` is a queue of this type, the main operations on `q` are:

- `q.enqueue(x)` or `q += x`: add `x` to the end of `q`;
- `q.dequeue`: remove the front element from `q` and return it;
- `q.isEmpty`: test whether `q` is empty.

Specifying queues

```
/** A queue of data of type A.
  * state: q : seq A
  * init: q = [] */
class Queue[A]{
  /** Add x to the back of the queue.
    * post: q = q0 ++ [x] */
  def enqueue(x: A)

  /** Remove and return the first element of the queue.
    * pre: q ≠ []
    * post: q = tail q0 ∧ returns head q0
    * or post: returns x s.t. q0 = [x] ++ q */
  def dequeue: A

  /** Is the queue empty?
    * post: q = q0 ∧ returns q = [] */
  def isEmpty: Boolean
}
```

Implementing breadth-first search (first attempt)

To implement the breadth-first search, we create a queue, initially containing just `start`.

We then repeatedly remove an element `w` from the queue. We then consider each neighbour `w1` of `w`. If `w1` equals `target`, we are done. Otherwise, we add `w1` to the end of the queue.

If we get to a state where the queue is empty, we have explored all words reachable from `start` without finding `target`; hence there is no path from `start` to `target`.

More precisely —because we want to be able to be able to reconstruct the path— we store `Path`s in the queue. For efficiency, we store these paths in reverse order. Thus each element of the queue is of the form `List(w, w1, w2, ..., wn, start)` representing that `start`, `wn`, ..., `w1`, `w` is a path from `start` to `w`; we continue searching from `w`.

Implementing breadth-first search (first attempt)

```
/** Find a minimum length path from start to target.
  * @return Some(p) for some shortest Path p if one exists;
  * otherwise None. */
def findPath(start: String, target: String) : Option[Path] = {
  val queue = scala.collection.mutable.Queue(List(start))

  while(!queue.isEmpty){
    val path = queue.dequeue; val w = path.head
    for(w1 <- neighbours(w)){
      if(w1==target) return Some((target::path).reverse)
      else queue += w1::path
    } // end of for
  } // end of while
  None // no solutions found
} // end of findPath
```

A better algorithm

The above algorithm works in theory, but in practice is hopelessly inefficient, because it repeats a lot of work.

For example, the algorithm considers paths that revisit a word, for example, `List("grass", "brass", "grass")`.

Further, it considers paths that reach a word for which another (possibly shorter) path has been reached. For example, it considers the path `List("crass", "brass", "grass")`, even though earlier it had found the path `List("crass", "grass")`. Also, it will consider both the paths `List("class", "crass", "grass")` and `List("class", "glass", "grass")`, even though both reach the same word.

The solution is to avoid exploring a path further if it reaches a word that has previously been seen. Hence we use a variable `seen` that holds the set of words that we've previously seen.

A better algorithm

```
/** Find a minimum length path from start to target.
  * @return Some(p) for some shortest Path p if one exists;
  * otherwise None. */
def findPath(start: String, target: String) : Option[Path] = {
  val queue = scala.collection.mutable.Queue(List(start))
  // Keep track of the words we've already considered
  val seen = new scala.collection.mutable.HashSet[String]
  seen += start

  while(!queue.isEmpty){
    val path = queue.dequeue; val w = path.head
    for(w1 <- neighbours(w)){
      if(w1==target) return Some((target::path).reverse)
      else if(!seen.contains(w1)){seen += w1; queue += w1::path}
    } // end of for
  } // end of while
  None // no solutions found
} // end of findPath
```

The main function

```
var dict : Dictionary = null // The dictionary

def main(args: Array[String]) = {
  val t0 = System.currentTimeMillis()
  // parse arguments
  val errMsg =
    "Usage: scala WordPaths [-d dict-file] start target"
  var i=0; var start = ""; var target = ""
  var dictFile = "knuth_words"
  while(i<args.length){
    if(args(i)=="-d"){ dictFile = args(i+1); i += 1 }
    else if(start=="") start = args(i)
    else{ require(target=="", errMsg); target = args(i) }
    i += 1
  }
  require(target!="", errMsg)
  ...
}
```

The main function

```
def main(args:Array[String]) = {
  ...
  // check lengths agree; initialise dictionary
  require(target.length == start.length,
          "start and target words should be the same length")
  dict = new Dictionary(dictFile)

  // The main stuff
  val optPath = findPath(start, target)
  optPath match{
    case Some(path) => printPath(path)
    case None => println("No path found")
  }

  println("Time taken: "+(System.currentTimeMillis()-t0))
}
```

Solving the puzzle

The program solves Lewis Carroll's example in less than one second, finding the following path:

```
grass, trass, tress, trees, treed, greed, green
```

Breadth-first search

In the Algorithms course, you'll see a variant on this breadth-first search. That algorithm assumes the graph has already been constructed, whereas the algorithm we've seen never explicitly constructs the graph.

Summary

- Programming with abstract datatypes;
- `Queue`, `Option`;
- Breadth-first search.
- Next time: Bit maps and hashing.

Note: For the exam, you do not need to know details of specific classes in the Scala API; however, you'll find this knowledge useful for general programming.

Compiled on February 14, 2019