IP Lecture 10: Modularisation and Abstract Datatypes

Joe Pitt-Francis —with thanks to Mike Spivey & Gavin Lowe—

Reading: Chapters 4 (excluding Section 4.5) and 6 (excluding Sections 6.11–6.13) and Section 7.5 of Programming in Scala.

Overview

Imperative Programming Part 2 is about data structures.

We will be seeing some key principles (buzz words) which are used in good programming practice and in object-oriented programming:

- Modularisation: all functionality and data of a "thing" are held in one place (possibly in one file);
- Abstraction: functionality is specified in its general form—free from any specific implementation;
- Encapsulation: implementation details are hidden from the caller.

We will also be seeing invariants again! Whereas before we needed to maintain some meaningful property of a loop (*loop invariant*), now we need to maintain some meaningful property of a data structure (*data structure invariant*).

Modularisation

In order to write large programs, it is important to split the problem up into manageable chunks, or modules. Each module should separate the interface from the implementation.

- The interface describes what the module does.
- The implementation does it.

Somebody who wants to use the module should only have to read the interface.

Procedures and functions as modules

Procedures and functions can be seen as a form of modularisation.

- The interface is the signature (i.e. the types of arguments provided and result returned), together with a precise description of what the code does, e.g. in terms of pre- and post-conditions.
- The implementation is the local variables together with the algorithm used.

Somebody who wants to use the procedure should only have to read the interface. (Does your code satisfy these criteria?)

Modules

A module collects together related data and operations upon that data.

- The interface is the interfaces of the operations provided by the module, perhaps linked to an abstract view of the state;
- The implementation is the internal data, plus the implementations of the operations.

Benefits of using modules:

- The code becomes easier to understand if broken down into manageable chunks.
- We can reuse the module, perhaps in a different program.
- We can improve the implementation of a module, without changing the interface, thereby improving the performance of the program without changing the correct behaviour.

Objects

In an object-oriented language, like Scala, modules are objects and classes.

An object is simply a software artefact with some data, and some operations on that data.

Normally the data is private to the object in question: other parts of the program can access the data only via the operations; we don't want code elsewhere in the program to be able to mess with the internal state of this object.

A class is the set of all objects of a particular type. Equivalently, it's a template for objects.

Operations on objects

If an object `obj` has an operation `op(...)` then we can invoke it with argument `args` using the notation `obj.op(args)`.

The identifier `this` represents the current object, so we can write `this.op(args)` to call the operation `op` defined within the current object. We can abbreviate this to `op(args)` — the notation we've been using so far.

A spelling checker

We'll write a very simple application to check the spelling of words. The user will provide a word on the command line, and the program will say whether or not it is a valid word.

There are lots of files on the Web that give lists of English words[a]. We will use the list `knuth_words`.

However, we want to read the words into a software object, to allow fast access. We therefore want to represent a set of words internally.

```
$ scala SpellCheck invarient
word invarient not found
$ scala SpellCheck invariant
invariant is a valid word
```

[a] Perhaps on your computer in `/usr/share/dict/`
See `https://en.wikipedia.org/wiki/Words_(Unix)`

Sets

Scala has various classes representing sets, both mutable and immutable. We will use mutable sets.

The base of the mutable sets is the trait `scala.collection.mutable.Set`. We can think of a trait as describing what a class should do, but not providing the implementation. Actual implementation classes extend this trait.

New sets (using a default implementation) can be created by, e.g.,[a]

```
val s = scala.collection.mutable.Set(2, 4, 6, 8)
val empty = scala.collection.mutable.Set[Int]()
```

[a] by including `import scala.collection.mutable.Set` at the top of the file, we could omit the "`scala.collection.mutable.`".

Sets

An element `x` can be added to set `s` by `s.add(x)` or `s += x`.

The expression `s.contains(x)` tests whether set `s` contains `x`.

The expression `s.size` gives the number of elements of `s`.

There are many more operations on sets, such as `map` and `filter`.

Sets are mathematical objects: we can therefore write mathematical specifications of their behaviours.

Sets

```
/** A set of values of type A.
  * state:  S : ℙ A
  * init:  S = {}   */

trait Set[A]{
  /** Add elem to the set.
    * post:  S = S_0 ∪ {elem}   */
  def add(elem: A)

  ...
}
```

S_0 represents the value of S at the start of the function call; this is a standard convention.

Sets

```
trait Set[A]{
  ...

  /** Test if this set contains elem.
    * post: returns b s.t. S = S_0 ∧ b = (elem ∈ S)
    * or, more simply
    * post: S = S_0 ∧ returns elem ∈ S */
  def contains(elem: A): Boolean

  /** The size of the set
    * post: S = S_0 ∧ returns #S
  def size: Int

  // ...and about 100 more operations...
}
```

Sets

To store the words of the dictionary, we'll use a particular implementation of `scala.collection.mutable.Set[String]`, namely `scala.collection.mutable.HashSet[String]`. The "`HashSet`" represents that the implementation is based on a hash table; see later.

A new set (initially empty), suitable for holding data of type `A` can be created by `new scala.collection.mutable.HashSet[A]`.

Initialising the dictionary

We can create the dictionary by

```
// The dictionary
val words = new scala.collection.mutable.HashSet[String]
```

We then want to read in words from the file, and add them to `dictionary`. We only want to include words that contain only lower case letters. The following helper function tests whether a particular word should be included:

```
def include(w:String) = w.forall(_.isLower)
```

Initialising the dictionary

Scala provides a class `scala.io.Source`; each `Source` represents a source of textual data. We can create a `Source` associated with a file `fname` by `scala.io.Source.fromFile(fname)`. We can then call the operation `getLines` on this to get a sequence (formally an `Iterator`: see next term) of strings corresponding to the lines of the file (with newline characters removed). We can then iterate over this sequence.

We can initialise the dictionary from the file as follows:

```
/** Initialise dictionary with all words from fname */
def initDict(fname: String) = {
  val allWords = scala.io.Source.fromFile(fname).getLines
  // Should word w be included?
  def include(w: String) = w.forall(_.isLower)
  for(w <- allWords; if include(w)) words += w
}
```

Looking up words in the dictionary

The following function tests whether a given string `w` appears in the dictionary.

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

The main method

```
object SpellCheck0{
  /** The dictionary (arguably should be in separate module) */
  val words = new scala.collection.mutable.HashSet[String]

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

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

  def main(args: Array[String]) = {
    assume(args.length==1, "Needs one argument")
    val w = args(0)
    initDict("knuth_words")
    if(isWord(w)) println(w+" is a valid word")
    else println("word "+w+" not found")
  }
}
```

Modularisation

The implementation mixes the representation of the dictionary with the user interface. It would be better to implement the dictionary in a different module.

- If we wanted to change the implementation —for example to use an ordered array of `String`s, combined with binary search— we would have to search through the program to find the parts to change. This would be better if it were all together in a single module.
- Later we will want a dictionary in a different program. Implementing the dictionary in a different module will allow us to reuse that module.

The `Dictionary` class

We will implement the dictionary as a class `Dictionary`. Recall that a class is a template for objects. This will allow us to create several `Dictionary` objects, perhaps based on different files. We will provide the name of the file when we construct a `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){
  ...
}
```

This will allow us to write, for example,

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

The Dictionary class

```
/** 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) = ... // as before

  // Initialise the dictionary
  initDict(fname)

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

The `Dictionary` class

`words` and `initDict` are marked as `private`, which means they cannot be accessed from outside the class. Code outside the class may use it only via its (public) function `isWord`.

When an object is created, all the code outside `def`s is executed; here, `words` is initialised and `initDict(fname)` called.

The spell checking application

```
/** A simple application to check the spelling of words */
object SpellCheck{
  def main(args: Array[String]) = {
    assume(args.length==1, "Needs one argument")
    val w = args(0)
    val dict = new Dictionary("knuth_words")
    if(dict.isWord(w)) println(w+" is a valid word")
    else println("word "+w+" not found")
  }
}
```

It works. . . and it's educational!

```
$ fsc SpellCheck.scala Dictionary.scala
$ scala SpellCheck invarient
word invarient not found
$ scala SpellCheck invariant
invariant is a valid word
```

Summary

- Modularisation, separating interface from implementation;
- Objects, encapsulating state and operations on that state, and implementing abstract datatypes;
- Traits as specifications of abstract datatypes, e.g. sets;
- `HashSet`;
- Classes as templates for objects.
- Next time: Specifying and implementing `map` datatype.