IP Lecture 11: Specifying another Abstract Datatype

Joe Pitt-Francis

—with thanks to Mike Spivey & Gavin Lowe—

Abstract datatypes

- Module puts data and functionality of a "thing" in one place.
- Abstract datatype gives the generic (idealised) description.
  - `state` refers to what is represented.
  - `init` gives the initial configuration of `state`.
  - Preconditions and postconditions on operations.
- `trait` gives interface of what a module should do:
  - list of "promises" to be fulfilled by whatever later uses trait;
  - not an implementation and so can't be used in its own right;
  - c.f. an abstract class or Java's *interface*.
- Later we will see that a `class` which `extends` a `trait` must give a concrete implementation to deliver on the promises.

The phone book

We now consider a different application, namely a phone book. We will look at several different implementations. We will want to add people's names and numbers to the book, and then look up a person's number. The program will be in two parts: the phone book itself and the user interface.

Abstractly, the state of the phone book is a mapping (i.e. a finite partial function) from names to phone numbers. Modelling each of these as a string, the abstract state is a mapping

state: $book$: `String` $\rightarrow\!\!\!\!\!+$ `String`

for example[a]

$book = \{$"Gavin" $\rightarrow$ "1234", "Mike" $\rightarrow$ "4567", "Pete" $\rightarrow$ "5443"$\}$.

[a]The "maps-to" arrow "$\rightarrow$" is sometimes written as "$\mapsto$"; I'm using the simpler arrow here for compatibility with Scala notation.

The phone book

We can capture the requirements of the phone book as a trait.

```
/** The state of a phone book, mapping names (Strings) to
  * numbers (also Strings).
  * state:  book : String ⇸ String
  * init:  book = {} */
trait Book{
  ...
}
```

Operations on the phone book

We want to be able to add a particular name and number to the phone book. This corresponds to adding $name \to number$ to $book$:

```
/** Store number against name in the phone book
  * post: book = book₀ ⊕ {name → number} */
def store(name: String, number: String)
```

where $f \oplus g$ is function f overwritten by g, i.e.

$$(f \oplus g)(x) = \begin{cases} g(x) & \text{if } x \in \operatorname{dom} g \\ f(x) & \text{if } x \in \operatorname{dom} f - \operatorname{dom} g \\ \text{undefined} & \text{otherwise} \end{cases}$$

and $\operatorname{dom} f$ is the set of values where f is defined

$$\operatorname{dom} f = \{x \mid (x \to y) \in f\}.$$

Operations on the phone book

We also want to be able to look up a particular name in the phone book. That is, we want to be able to find $book(name)$, if initially $name \in \operatorname{dom} book$.[a]

```
/** The number stored against name
  * pre:  name ∈ dom book
  * post: book = book_0 ∧ returns  book(name) */
def recall(name: String): String
```

The above operation has a non-trivial precondition, so it's useful to have an operation to test if that precondition is true, i.e. if $name$ appears in the book.

```
/** Does name appear in the book?
  * post: book = book_0 ∧ returns  name ∈ dom book */
def isInBook(name: String): Boolean
```

[a] Conventionally, we don't use 0-subscripts in the precondition.

The main program

Later we will produce an object `MapBook` that implements the `book` trait. We can then implement the main program as follows.

```
object PhoneBook{
  def main(args: Array[String]) = {
    val book = MapBook
    var done = false // are we finished yet?
    while(!done){
      val cmd = readLine("Recall (r), store (s) or quit (q)? ")
      cmd match{
        case "r" => ...
        case "s" => ...
        case "q" => done = true
        case _ => println("Please type ‘r’, ‘s’ or ‘q’.")
      } // end of match
    } // end of while
  } // end of main
}
```

The main program (continued)

```
    cmd match{
      case "r" => {
        val name = readLine("Name to recall? ")
        if(book.isInBook(name)) println(book.recall(name))
        else println(name+" not found.")
      }
      case "s" => {
        val name = readLine("Name to store? ")
        val number = readLine("Number to store? ")
        book.store(name,number)
      }
      ...
    }
```

The `scala.collection.mutable.Map` trait

Scala has a trait, `scala.collection.mutable.Map`, which almost exactly meets our requirements. The trait `scala.collection.mutable.Map[A, B]` corresponds to mappings of type `A` $\rightarrow$ `B`.

An object implementing this trait (using the default implementation, based on a hash table) can be created by

```
val m = scala.collection.mutable.Map[A, B]()
```

corresponding to the empty mapping.

The `scala.collection.mutable.Map` trait

Each object `m` that implements the `scala.collection.mutable.Map[A,B]` trait provides the following operations:

- `m.apply(key: A) : B` (or, more concisely, `m(key: A) : B`): "Retrieves the value which is associated with the given key."
- `m.update(key: A, value: B) : Unit` (or, more concisely, `m += ((key, value))` or `m += key -> value`): "Adds a new key/value pair to this map."
- `m.contains(key: A) : Boolean`: "Tests whether this map contains a binding for a key."

and about a zillion other operations.

Aside: many objects that are "function-like" have an operation called `apply`. An application of the form `o.apply(args)` can be abbreviated to `o(args)`. Array indexing is an example of this.

An implementation of `Book` based on `Map`

The following implementation of `Book` is a wrapper around a `Map`.

```
object MapBook extends Book{
  private val map = scala.collection.mutable.Map[String, String]()

  /** Store number against name in the phone book */
  def store(name: String, number: String) = map.update(name, number)

  /** The number stored against name */
  def recall(name: String) : String = {
    assert(map.contains(name)); map(name)
  }

  /** Does name appear in the book? */
  def isInBook(name: String) : Boolean = map.contains(name)
}
```

Note that `map` is annotated with the word `private`. This means that `map` is private to this object, and not accessible from elsewhere.

Why not use a `Map` directly in `PhoneBook`?

We could have written `PhoneBook` to use a `Map` directly. However, such an implementation would mix the representation of the data with the main operation of the program. If we decide we want to change the representation, we would have to make changes in several places in the program. In this case, it would be acceptable — but this is a very small program. With a large program, tracking down all the necessary changes would be very tedious.

It is better to separate out the representation of the data into a separate module. If we subsequently want to change the representation of the data, we only have to re-write that module and make a very small number of changes to the main program.

Classes

Our main `PhoneBook` object uses a single `Book` object.[a] We therefore defined `MapBook` as an `object`, as we only want a single instance. But this wouldn't work if we wanted two different `Book`s, say two `MapBook`s.

A class is a template for objects. If we want several objects of the same type then we need to define an appropriate class, for example

```
class MapBook extends Book{ ... }
```

where the body of the definition is precisely as before.

[a] Defining a single named object rather than a class template is sometimes called the *singleton pattern*. It is easy to make a singleton in Scala—in other languages it becomes more difficult.

Classes

Our main `PhoneBook` object could then create two `ArrayBook` objects by

```
val book1 = new MapBook; val book2 = new MapBook
```

These are independent: each has its own internal state.

An object is an instantiation of one member of a class.

We can call operations on each object, for example:

```
if(book1.isInBook(name)) println(book1.recall(name))
else if(book2.isInBook(name)) println(book2.recall(name))
else println(name+" not found")
```

Testing

Here's a test suite that tests `MapBook` and several other implementations of `Book`.

```
class BookTest extends FunSuite{
  for(book <- List(ArraysBook, MapBook, PairArrayBook, EntryArrayBook)){
    book.store("Gavin", "1234"); book.store("Pete", "5678")
    test(book+": recall of first stored value"){
      assert(book.isInBook("Gavin")); assert(book.recall("Gavin")==="1234")
    }
    test(book+": recall of second stored value"){
      assert(book.isInBook("Pete")); assert(book.recall("Pete")==="5678")
    }
    test(book+": recall of unstored value"){
      assert(!book.isInBook("Mike"))
      intercept[AssertionError]{book.recall("Mike")}
    }
  }
}
```

Summary

- Traits as specifications of abstract datatypes;
- `Map`;
- Classes as templates for objects.
- Next time: Other implementations of `map` datatype.