Computer Science 211
Data Structures
Mount Holyoke College
Fall 2009

Topic Notes: Associations

Let's consider a very simple example of a data structure called an `Association`.

As the name suggests, an `Association` is a way to associate pairs of objects, one of which is the `key` and one of which is the `value`. Once created, the key cannot be changed, but the value can.

Unlike the structures we have seen so far, this is a "general purpose" structure.

- What should such a structure look like?
- What instance variables will it need?
- What constructors should be provided?
- What methods will it need?

As we have seen, in Java, all objects that are not primitive types are extensions of the class `java.lang.Object`, or just `Object`. This is very convenient when developing data structures, since we can develop our structures to hold references to `Object`s and then use them to store instances of any Java class. This includes `String`s.

Note: very soon we will look at the idea of *generics*, which was introduced into Java a few years ago and to our textbook and the structure package shortly after. But for the moment, we will make use of the fact that all Java objects are instances of class `Object` for our data structures.

The text has an example that uses `Association`s to associate words with their "pig latin" equivalents. We will use Associations in a similar example that associates magic spells with a description of their effect. The name of the spell will be the key, and its effect will be the value.

See Example:
`/home/jteresco/shared/cs211/examples/Spells`

If we do a great job implementing an `Association` for this program, we have a good chance to be able to use it again. This *reusability* is a major focus of this course.

Implementation in Structure

The structure package includes an `Association` class that we can use for this example.

This class is defined as part of `package structure`, meaning it can access `protected` entries of other classes in the structure, and those classes can access `class Association`'s `protected` items.

Pre and Postconditions

The structure package uses an extension to Javadoc that provides two new fields: `@pre` and `@post`. These are *preconditions* and *postconditions* for the methods.

These comments set out a contract for the use of a particular method. For instance, see the following code:

```
/**
    @pre  0 <= index < this.length()
    @post  returns character at "index" position
          (starting count from 0) in this
**/
public char charAt(int index)
{...}
```

The contract expressed by the pre and postconditions is that the implementer promised that the postcondition will be true after executing the method, as long as the user promises that the precondition will be true when it is called. Thus both the caller and the implementer have responsibilities under the contract.

For the above example, the user is required to specify an `index` which is legal for the string (between 0 and `myString.length()` - 1, inclusive), and if the user meets that commitment, the implementation promises to return the character in the `index` position of `myString`.

Assertions

It is useful having these as comments, but often it is much more useful to have them checked at run-time, as if any fail, it is indication of an error in the program.

Moreover, the location of the failure is more likely to provide a pointer to the source of the error than just getting a wrong answer (or system crash).

Thus, there is a class `Assert` in the structure package which contains methods to check these at run-time.

Thus if we were writing the code for a `String`'s `charAt` method above, we could write:

```
public char charAt(int index) {
    Assert.pre(0 <= index && index < length(),
               "Index out of bounds for string");
    ...
    Assert.post(..., "post condition error");
}
```

If either of the boolean conditions is false at the time they are executed then the system will raise an exception (i.e., crash) and print a message telling that a pre/postcondition failed and give the error message.

Sometimes pre and postconditions can't be expressed concisely or efficiently (e.g., how do you say "the array is sorted"), so we may not be able to enforce these in the code using the `Assert` methods.

Another potential pitfall when enforcing pre and postconditions with assertions, is that you may be tempted to call the routine recursively (and get yourself into non-terminating computation) in checking pre or postconditions! In these cases, the comments will have to suffice.

Aside: Eiffel is an object-oriented programming language with built-in support for pre and postconditions. It also has compiler switches which can be turned on and off to determine whether or not pre and postconditions are checked during a program's execution (the default is that only preconditions are checked).

We will expect all methods in the classes you implement here to be decorated with Javadoc-style pre and postconditions, and checks using the `Assert` methods where it makes sense.

Recent versions of Java provide a keyword `assert` that behaves similarly to the structure package's `class Assert`. More on that later in the semester.

Back to Association.java

The actual implementation of the `Association` class is pretty straightforward. A couple of quick notes:

- We require a `key` to construct a new `Association`, but the `value` is optional. If not provided, the `value` part defaults to `null`.
- Two `Association`s are considered equal (by the `equals` method) if their `key`s are the same, regardless of their `value`s.
- We have an *accessor* ("getter") for the `key` (`getKey`) but no *mutator* ("setter"). Once created, the `key` of an `Association` may not be modified.
- For the `value`, we have both an accessor (`getValue`) and a mutator (`setValue`).