In many Java applications that store large amounts of information about large numbers of persons, organizations, commodities, or other entities, it is useful to associate the data structure that contains all of the information about one such entity with a much smaller object called a key or an identifier. Each key uniquely identifies the data structure that is associated with it (which is typically called the value of the key). A key can be used as a shorthand way to refer to the value associated with it, as a search key for recovering the value from a large database, or as a sorting key for putting a collection of such values into some useful order without having actually to copy or rearrange the values themselves. (Typically it is much faster to sort the keys than to sort the values associated with them.)

In today’s lab, we’ll define a generic type for key-value pairs, which are objects that record the associations of values with keys.

1. Open Eclipse, create a new project called generics and in that project create a new package called kvpairs for the classes we develop in this lab.

2. Let’s use the name KeyValuePair for the generic class that we want to define. Because we want the implementation of this class to be the same no matter what classes the keys and the values actually belong to, we’ll use two type parameters in its declaration. Recall that the list of type parameters begins with a less-than sign and ends with a greater-than sign and that adjacent type parameters are separated by commas.

   In Eclipse, use the “New Class” wizard to create the KeyValuePair. The wizard will open an editor containing the boilerplate for the class definition. Add a list of type parameters to the signature of the class.

3. Each KeyValuePair should store its key and its value in private fields. Add declarations for these fields.

4. Define a constructor for KeyValuePair that takes the key and the value as arguments and puts them in the appropriate fields of the newly created KeyValuePair.

5. Provide an accessor method for each of the fields. We’ll assume that KeyValuePair objects, once created, are immutable, and so we won’t provide mutators.

6. Since, by default, the KeyValuePair is derived from java.lang.Object, it inherits the methods that that class supports, including toString, equals, and hashCode. Test these methods to determine whether they give results that seem appropriate to the KeyValuePair class.

   If they do not, override them by defining new methods with the same name inside the class definition and prefixing the definition of each method with the “annotation” @Override, signalling to the compiler that the replacement is intentional.

   Test your new methods to make sure that they give the results you expected. Note that equals and hashCode are connected by an invariant: If alpha.equals(beta) is true, then the same int value must be returned by ‘alpha.hashCode()’ and ‘beta.hashCode()’. Make sure that this invariant still holds if you override either equals or hashCode.

7. The java.util package provides a generic class called LinkedList in which one can store any number of objects and access them sequentially, as in Scheme. But Java’s LinkedList objects are actually Objects, and they have internal states in a way that Scheme lists don’t.

   For instance, there’s no cons function. To create a list 1s with one element (the number 42, for instance) in Java, one would execute the statements
LinkedList<Integer> ls = new LinkedList<Integer>();
ls.addFirst(42);

Unlike cons in Scheme, addFirst changes the state of ls. The constructor gives you an empty list, and then the call to addFirst changes ls from an empty list into a list containing one element.

Note that, in this example, the compiler autoboxes the int 42, wrapping it in an Integer object so that it matches the type specified for values in the linked list.

Now for the exercise: In your test program for the KeyValuePair class, create a linked list of at least five key-value pairs in which the keys are strings and the associated values are the lengths of those strings. The key-value pairs don’t have to be arranged in any particular order.

8. One of the methods provided in the LinkedList class is iterator. This method returns an object of some class that implements the Iterator interface and can be used for traversing the linked list. The Iterator interface specifies that its instances must provide two important methods: next, which takes no arguments and returns a new element from the underlying linked list each time it is invoked, and hasNext, which takes no arguments and determines whether there are any elements of the underlying linked list that have not yet been returned by next.

Thus one common pattern for traversing a LinkedList<Integer> object called ls and processing each of its elements in some way looks like this:

    Iterator<Integer> traverser = ls.iterator();
    while (traverser.hasNext()) {
        Integer element = traverser.next();
        // process element in some way
    }

The exercise, then, is to write and test a static, generic lookup method in your test program that takes as arguments a key and a linked list of key-value pairs and performs a linear search of the linked list. If any of the key-value pairs in the linked list is equal to the key supplied as the first argument to lookup, the lookup procedure should return the value associated with that key. If none of the keys match, it should return null.

9. An association list is a linked list of key-value pairs in which no two of the keys are equal. Write and test a method that takes a linked list of key-value pairs as argument and determines whether it is an association list.