COMP 401

Prasun Dewan[1]

Collection Kinds

We saw in an earlier chapter three kinds of collections: histories, databases, and sets. These were implemented using arrays, which are collections in their own right. In this chapter, we will look at and compare a greater variety of collections. We will also contrast identify their logical structures and show how it is different from the Bean logical structures we have seen so far.

Heterogeneous Beans vs. Homogeneous Collections

Recall that a structured type can be decomposed into one or more typed components, and this decomposition can be done based on its representation in memory or the operations it provides to the users of its type. These two decompositions create the physical and logical structure, respectively, of the type. The approach we saw earlier of deriving the physical structure by making each component corresponds to an instance variable works for any type, though we applied it only to Beans. It did require us to know the implementation of the type. The approach for deriving the logical structure did not require us to now its implementation, but required the use of the Bean naming conventions. Thus, it applied only Beans.

Beans are not the only structured types we have seen so far. Arguably, arrays are also structured as they can be logically (and physically) decomposed into their elements. By comparing these two kinds of types, we can gain a better understanding of them. We will focus on logical structures, and mention physical structure only in passing.

In the case of a Bean, the components of different, unrelated types, and these components serve different functions denoted by the names of the components. For example, the X and Angle properties of a Point are of types, int and double, respectively, and serve to characterize different aspects of the Point. In other words, Beans are heterogeneous groupings of components. Moreover, which component of a Bean is accessed is determined at compile time. Because of these two properties, we must write different code to manipulate different components of a Bean:

```
System.out.println("X:" + location.getX() );
System.out.println("Angle:" + location.getAngle() );
```

In the case of an array, on the other hand, the components are all of the same type. If the type is an Object type, then the elements can be instances of different subclasses of these types. But the declared element type describes the common supertype of these elements. In other words, arrays are homogeneous groupings of components. Which component of a collection is accessed can be determined at runtime. These two properties allow us to write the same code, often written in a loop, to access different components of an array:

```
for (int I = 0; < args.length; i++) {

  System.out.println("i:" + args[i] );

}
```

Here we have written a single expression to determine the array element, which can refer to different elements as the variables in the expression change values.

As we will see below, types other than arrays also have these three properties. Any structured type that allows the same code to refer to different components at runtime will be called a collection.

Indexed Collections

Arrays are different from some of the other collections we will see later in that array elements can be explicitly specified using integers called indices, and its elements are ordered by their indices. As we will see later, these two properties do not hold for some other types of collections such as tables, stacks, and queues. These do hold for the StringHistory, StringDatabase, and StringSet types we defined earlier. In the case of arrays, the syntax for providing the index of an element is built into the language:

```
   args[i];
```

As the other three collection types were defined by us, we use method call syntax to refer to array elements:

```
  names.elementAt(i);
```

But the basic concept of indexing the collections still holds for these collections.

Static vs. Dynamic Types

Given an array variable:

```
String[] strings;
```

We can assign to it arrays of different sizes:

```
strings =  {"hello", "world"};
strings = new String [10];
```

This was not the case in several older languages – in particular Pascal. However, even in Java, when the array is created, its size is fixed – 2 in the first array and 10 in the second one, above.

On the other hand, the collections we created in a previous chapter, StringHistory, StringDatabase, and StringSet, can grow:

```
names.addElement(input);
```

Structured types can be classified into dynamic and static types based on whether the number of components of these types can change at runtime, that is, the number of edges emanating from a node can change. Beans and arrays are static – StringHistory, StringDatabase, and StringSet are dynamic. As we saw in the implementation of these types, the physical structure of these types was fixed but their logical structure was dynamic. A dynamic collection can be indexed or not.

The three collections we created are representative of broader classes. We will define a history to be any collection in which an element cannot be changed or deleted, a set to be any collection in which duplicated are not allowed, and a database to be any collection in which elements can be added and duplicates are allowed. These collections can be dynamic or static, and indexed or non-indexed. And of course, they can provide more operations than our illustrative examples do. In particular, a database can allow searching, insertion, and modification of elements.

Least Privilege → Collection Variety

Among the three dynamic collections we have seen so far, a StringHistory, StringDatabase, and StringSet, the functionality provided by a StringDatabase is a strict super set of the one provided by a StringHistory. Given a StringHistory, it was useful to create a more sophisticated object to illustrate inheritance. But is there any reason for these two types to co-exist? In other words, given a database, does it make sense to also have a history that provides a subset of the operations in the database?

There are of course situations in which users of the type need only the functions of the restrictive type, as we saw in the example that used StringHistory. If we did not have a type that matched our needs, we would need to use one that provides a superset of the operations we actually need. In our example, we would use a StringDatabase instead of StringHistory. So our question reduces to: Is there a danger in using a type that provides more operations than we need?

The answer is provided by the need to know or least privilege principle we saw in an earlier chapter. We do not want to give users the right to perform operations they do not need. By giving StringDatabase to users who need StringHistory, we violate this principle, and allow them to delete elements, thereby providing ways to compromide the integrity of the object. This compromise may occur accidentally or maliciously, and typically, it is accidents that we must guard against. It is to provide such accidents that we sometimes open a document in read-only mode, which is a mechanism in another context to follow the principle of least privilege.

The least privilege principle is a reason for the great variety in collection types we see here.

Implementation → Collection Variety

At one time, Java come with only one dynamic indexed collection, Vector. It was a collection of Object elements, and had a superset of operations in our StringHistory, shown below:

```
public final int size();
public final Object elementAt(int index);
public final void addElement(Object obj) ;
public final void setElementAt(Object obj, int index);
public final void insertElementAt(Object obj,  int index);
public final boolean removeElement(Object obj);
public final void removeElementAt(int index);
public final int indexOf(Object obj);
```

The names of these operations and the order of their parameters inspired the operations of the StringHistory and StringDatabase types.

Vector was used to define sets, histories, and other kinds of collections of strings, arrays, vectors, and other types. Later, Java designers other implementations of many of the Vector operations, such as ArrayList, and LinkedList, and designed an interface, List, to capture these operations. List offered different names and parameter order for some of the old Vector operations:

```
public final int size();
public final Object get(int index);
public final void add(Object obj) ;
public final void set(int index, Object obj);
public final void insert(int index,  Object obj);
public final boolean remove(Object obj);
public final void remove(int index);
public final int indexOf(Object obj
```

For example, as we see above, they replaced the insertElementAt(Object obj, int index) with insert(int index, Object obj). They then extended the old version of Vector, which did not implement an interface, to also implement List. As a result, Vector provides both the old and new signatures of the operations. Here is some code to illustrate the List Interface and its two implementations, Vector and ArrayList:

```
import java.util.ArrayList;
import java.util.List;
import java.util.Vector;
public class VectorArrayListUser {
  public static void main (String[] args) {
    List names = new Vector();
    List grandSlams = new ArrayList();
    names.add("Nadal");
    grandSlams.add(14);
    names.add("Federer");
    grandSlams.add(17);
```

```
    names.add("Borg");
    grandSlams.add(11);
    names.add("Sampras");
    grandSlams.add(14);
  }
```

Here we have defined two different List variables, to which we have assigned instances of Vector and ArrrayList, respectively.

This history shows that implementation considerations are another reason for the variety in collection types, and also the usefulness of interfaces as a means for uniting different implementations. You will study the pros and cons of these implementations in depth in the data structure course.

The collection types we defined served mainly to sketch the implementation of array-based implementations of dynamic collections. As Java programmers, you will be working with the predefined Collections provided by Java. The code above illustrates how these types are used.

One of the apparent problems with these collections is that they force all elements to be Objects. When we study generics, we will see how to overcome this problem.

Table: Identifying Components by Keys

Let us look more carefully at the code above to illustrate what kind of structure is being simulated by the two Lists. In one list, it adds names of tennis stars, and in the other, number of grand slams won by these stars. More precisely, at index I in the (a) first list is the name of the tennis player, and (b) the second list is the number of grand slams won by the player. Thus, these two lists are simulating, in a rather inefficient manner, a table data structure. Java provides a single interface, Map, to define this structure. Two important operations provided by this type are:

```
// associates key with value, returning last value associated with key
public final Object put (Object key, Object value);
// returns last value associated with key, or null if no association
public final Object get (Object key);
```

The first operation, put, associates an object called, a key, with another object, called a value. The operation can be used to both create the first association of a key with an object, or change an existing association. If the key was previously associated with a value, then the operation returns the previous associated value; otherwise it returns null. The second operation takes a key as an argument. It returns the latest value associated with the key, if it exists, or null otherwise.

Java also provides a far more efficient implementation of tables than the one we see above, called HashMap. Again, in the data structure course, you will see study the concepts behind this implementation. Below we see its use to illustrates the semantics of Map:

```
public static void main (String[] args) {
```

```
 Map aMap = new HashMap();
 aMap.put("Nadal", 10); // change this
 aMap.put("Federer", 17);
 aMap.put("Sampras", 14);
 System.out.println(aMap.get("Nadal"));
 System.out.println(aMap.get("nadal"));
 aMap.put("Nadal", 11);
 System.out.println(aMap.get("Nadal"));
 System.out.println(aMap);
}
```

In contrast to the previous implementation, two add() operations in different instances of List are replaced by a single put operation in one Map instance. The following is the output of the code:

```
MapUser [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Oct 2, 2012 10:58:49 AM
10
null
11
{Sampras=14, Federer=17, Nadal=11}
```

As in this example, it is common to use Strings as keys, though as we see from the Map interface, Java allows us to use any object as a key. As we see here, case matters when we use String keys. We also see that the toString() method of a Hashmap nicely prints all the entries in it.

Can we consider a table to also be a collection? Recall a collection is a structured type that allows the same code to refer to different components at runtime. We can regard each (key, value) pair in the table to be a component that is identified by the key. The same get() or put() call can be used to access different components, depending on the actual key expression passed to these operations. Thus, a table is indeed a collection. It is like an indexed collection except that we are not restricted to using integers to identify the elements – we can use any object. A consequence of this generality is that the components are not ordered – we cannot simply increment an index to access the next element. As a result, a more complex mechanism is needed to access the elements of a table. The generality allows tables to, in fact, subsume the functionality of both Beans and indexed collections. In fact, certain research languages (e.g. Snobol) provide tables as the only structured type, as do some research operating system (e.g. EZ).

Stacks and Queues: Implicitly Accessed Elements

In the case of both indexed collections and tables, we can explicitly specify the component to be accessed, using indices and keys, respectively. We can also define collections in which elements are implicitly addressed. Stacks and queues are two important examples.

A stack is a collection that allows both addition and removal of elements. However, unlike a database, it does not allow arbitrary elements to be removed. It is possible to remove only the last element added.

This object mirrors a real world stack, in which a new item is pushed on top of the existing elements, and it is possible to remove only the top element.

The following is an interface describing a string stack.

```
public interface StringStack {
  public boolean isEmpty();
  public String getTop();
  public void push(String element);
  public void pop();
}
```

The push() procedure adds a new element to the collection, the getTop() function returns the top element, and the pop procedure pops the top element, and the isEmpty() operation returns true if the collection is empty. A stack is also called a LIFO(Last In First Out) collection, as the last element to be added is the first one removed.

The following code illustrates LIFO.

```
StringStack stringStack = new AStringStack();
stringStack.push("James Dean");
stringStack.push("Joe Doe");
stringStack.push("Jane Smith");
stringStack.push("John Smith");
System.out.println(stringStack.top());
stringStack.pop();
System.out.println(stringStack.top());
```

The result of executing this code is:

```
John Smith
Jane Smith
```

As JohnSmith was the last element added, and Jane Smith the second last one.

In contrast, a queue is a FIFO (First In First Out) collection, as the first element to be added is the first one removed. Thus, like a stack, a queue allows elements to be removed, but constrains the order in which they are removed. The following is an example of a queue.

```
public interface StringQueue {
  public boolean isEmpty();
  public String getHead();
  public void enqueue(String element);
  public void dequeue();
}
```

The enqueue() procedure adds a new element to the collection, the getHead() function returns the first (oldest) element in the collection, the dequeue() procedure removes the first element, and the isEmpty() operation returns true if the collection is empty.

The result of executing the following code:

```
StringQueue stringQ = new AStringQueue();
stringQ.enqueue("James Dean");
stringQ.enqueue("Joe Doe");
stringQ.enqueue("Jane Smith");
stringQ.enqueue("John Smith");
System.out.println(stringStack.getHead());
stringQ.dequeue();
System.out.println(stringStack.getHead());
```

is:

```
James Dean
Joe Doe
```

as James Dean and Joe Doe were the first and second elements added.

In the implementations above, pop() and dequeue() are procedures. A slight variation of these operations makes them functions with side effects that return the removed element.

Both stacks and queues are collections because the same code can be used to refer to different elements of the object. In both cases, we do not access arbitrary elements of the collections – we read the last and first element, respectively, of the two kinds of collections.

Collection Conventions

Beans represent two ideas –static heterogeneous groupings of components and a set of conventions for allowing humans and tools to extracting these components from the grouping. With collections, we have seen additional kinds of groupings. However, we have not seen a set of conventions for extracting collection components.

Bean conventions came with conventions to define static indexed collections. Other collections are not, unfortunately, not associated with any conventions. This partly due to the fact that many programmers simply use the predefined collections provide by the language, even if they provide operations not necessary for the task. Collection-based tools simply assume that these predefined collections are used. Such tools would not be able to handle collections such as StringHistory defined by the programmer.

This is not a limitation that is acceptable in ObjectEditor, which should be able to display collections we might define to gain experience with their implementation or follow the least privilege principle. Therefore, ObjectEditor defines its own conventions for the various collection kinds covered here. As in

the case of Beans, the details of these conventions have no conceptual value. Some of them are provided here to help you implement your assignments and projects.

ObjectEditor support for collections came when the original Vector was the only type supporting dynamic indexed collections. Therefore, ObjectEditor based one set of naming conventions on Vector, ensuring that Vector is recognized as a collection. A type C is considered a dynamic readonly collection of elements of type E if it provides the following methods:

```
public T elementAt (int index);
public int size();
```

The expected semantics of the operations are what we saw in the definition of StringHistory and Vector – the first operation is expected top return an element of type T an position index, and the second one is expeted to return the number of elements.

If the collection also provides the optional operation:

```
public Any setElementAt(T t, int index);
```

then ObjectEditor allows the collection elements to be edited. This operation is expected to replace the value of the element at position index with the new value t. The type “any” indicates that the return type of the operation does not matter – it can return nothing or the previous value at that index.

Other operations such as addElemnent() are not given any special meaning. These are displayed in the menu. After each such operation, ObjectEditor redisplays the elements of the collection.

A type following these conventions uses the Vector structure pattern, indicated to ObjectEditor using an annotation, as shown below:

```
@StructurePattern(StructurePatternNames.VECTOR_PATTERN)
public interface PointHistory {
        public void addElement (int x, int y);
        public Point elementAt (int index);
        public int size();
}
```

ObjectEditor also supports the List pattern, in which a different syntax is used for accessing an element and modifying it, as shown below:

```
@StructurePattern(StructurePatternNames.LIST_PATTERN)
public interface PointHistory {
        public void addElement (int x, int y);
        public Point get (int index);
        public int size();
}
```

Read vs. Write Methods

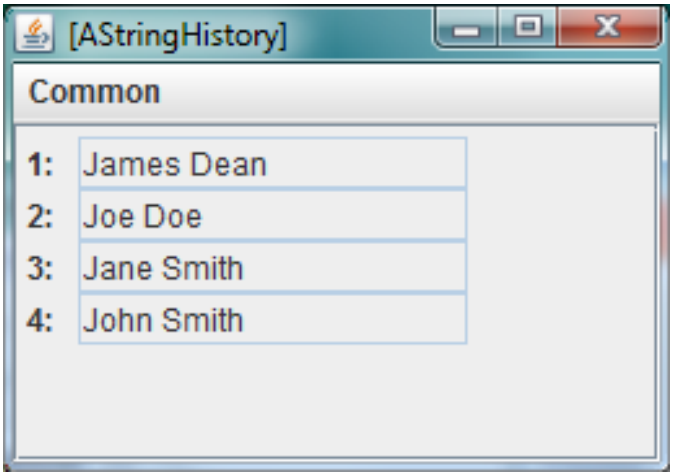
These conventions highlight the importance of providing two kinds of method in a structured object – read and write methods. Read method are those that allow access to the elements. The Bean getters, the elementAt() and size() methods provided by Vector, the get() and size() methods provided by List, are all examples of read methods. Write methods are those that allow modification of the structured type. The Bean setters, the setElementAt() and addElement() methods of Vector, and the set() and add() methods of List are all examples of write methods. An object without write methods is readonly; otherwise it is editable. The model-view-controller idea we will see later is based on these two kinds of methods.

ObjectEditor UI for Indexed Collections

ObjectEditor can use the conventions mentioned above to create both textual and graphical displays of collections. The following code:

```
public static void main (String[] args) {
  StringHistory stringHistory = new AStringHistory();
  stringHistory.addElement("James Dean");
  stringHistory.addElement("Joe Doe");
  stringHistory.addElement("Jane Smith");
  stringHistory.addElement("John Smith");
  ObjectEditor.edit(stringHistory);
}
```

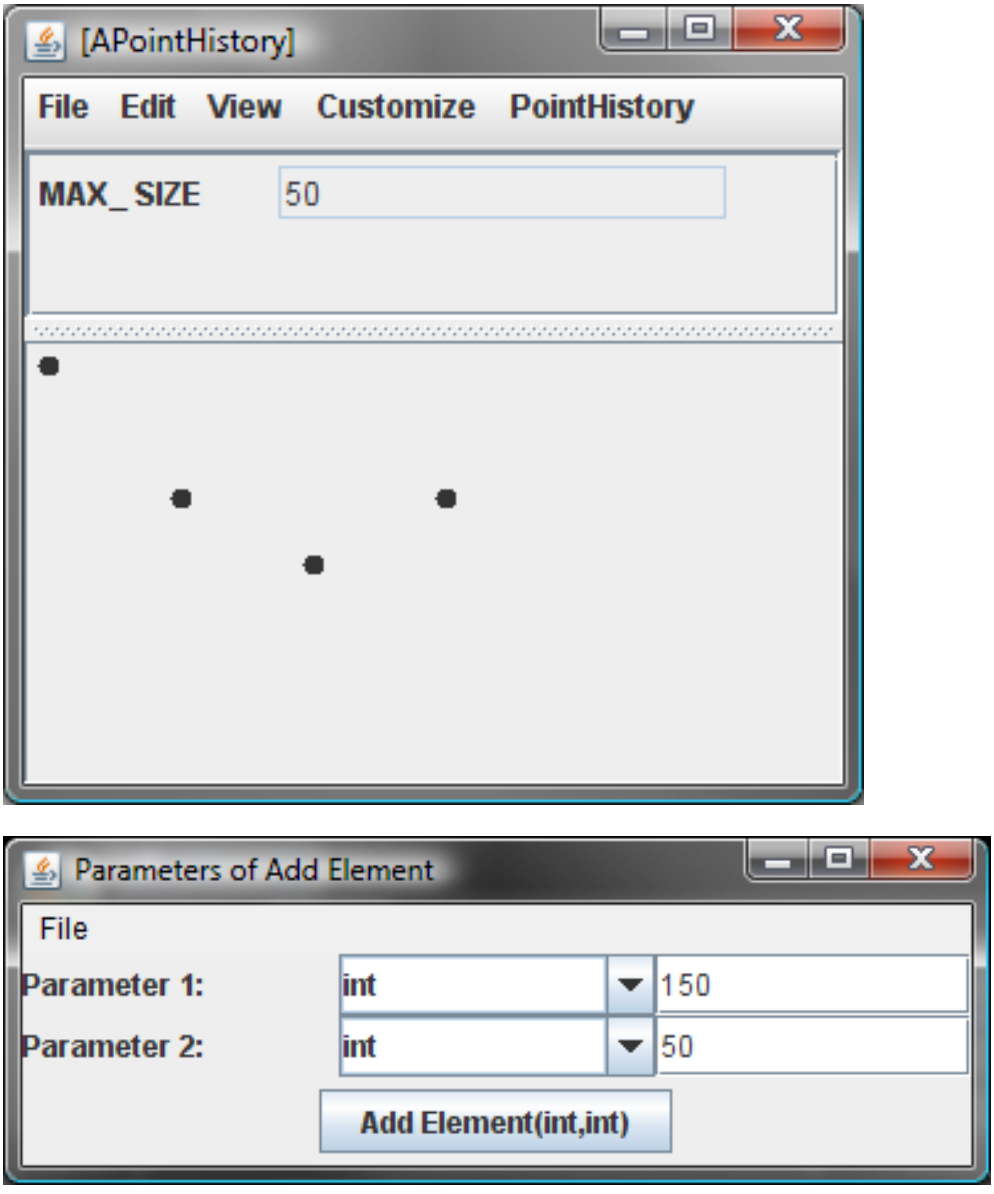
creates the display of Figure ???.



The main window display of a collection is much like the one we saw for beans with no shape components except that indices rather than property names are used as labels of the elements. The

main difference is that the display is dynamic – as new element are added to the collection or removed, they appear and disappear, respectively, in the display.

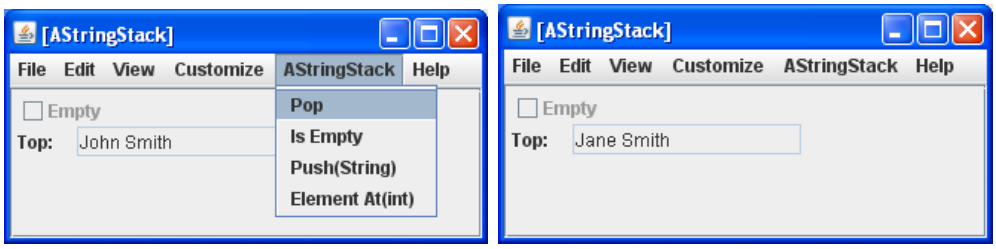
A collection with shape components is also handled similarly to a bean with such components, with the only difference again, the display reacting dynamically to addition and deletion of the components. This is illustrated in figure ???.



Here we see a PointHistory with four elements. The user is invoking the addElement method from the PointHistory menu. When the method is invoked a new Point object corresponding to the Cartesian coordinates, 150 and 50, will be added to the history, which will cause a shape representing the object to be added at these coordinates.

Displaying Stacks and Queues

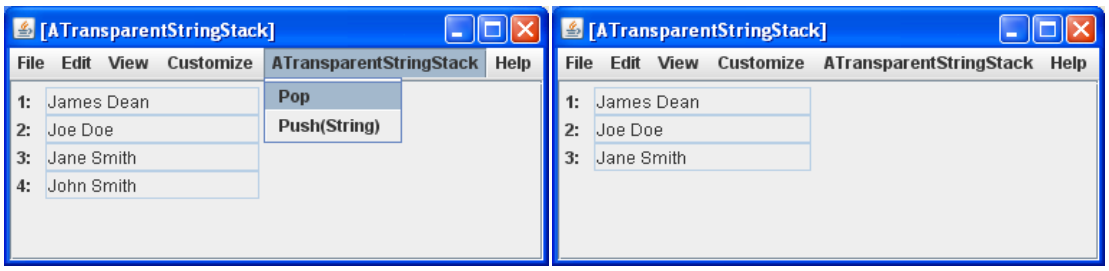
The nature of stack sand queues can be better understood by considering the ObjectEditor display of an instance of StringStack.



Since the stack allows retrieval of only the top element of the collection, ObjectEditor cannot display other elements. However it is possible to create a LIFO collection that allows retrieval of all elements of the collection, as shown by the interface below.

```
public interface TransparentStringStack {
  public int size();
  public String elementAt(int index);
  public void push(String element);
  public void pop();
}
```

Here, instead of providing an operation to examine the top element of the stack, the class provides the read methods provided by the history and database to access all of these elements. We will call such a stack a transparent stack as it allows a view of not only the top element but also the ones below it. Like a traditional stack, it supports LIFO. The following user interface illustrates the nature of LIFO.



In the interface above, to retrieve the top stack element, the stack user must call the elementAt() method with the last stack index, which is computed from the size() of the stack. A slight variation of a transparent stack would define an operation to directly return the top element.

Similarly, we can define a transparent queue, which allows examination of elements in the collection.